

Numerical Methods for Engineers and Scientists Using MATLAB

Second Edition

Ramin S. Esfandiari, PhD

CRC Press

CRC Press
Taylor & Francis Group,
6000 Broken Sound Parkway NW, Suite 300,
Boca Raton, FL 33487-2742

© 2017 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

International Standard Book Number-13: 978-1-4987-7742-1 (Hardback)

Library of Congress Cataloging-in-Publication Data

Names: Esfandiari, Ramin S., author.

Title: Numerical methods for engineers and scientists using MATLAB / Ramin S. Esfandiari.

Description: Second edition. | Boca Raton : a CRC title, part of the Taylor & Francis imprint, a member of the Taylor & Francis Group, the academic division of T&F Informa, plc, [2017]

Identifiers: LCCN 2016039623 | ISBN 9781498777421 (hardback : alk. paper)

Subjects: LCSH: Engineering mathematics. | Numerical analysis.

Classification: LCC TA335 .E843 2017 | DDC 620.00285/53--dc23

LC record available at <https://lccn.loc.gov/2016039623>

Visit the Taylor & Francis Web site at

<http://www.taylorandfrancis.com>

and the CRC Press Web site at

<http://www.crcpress.com>

Contents

Preface.....	xv
Acknowledgments.....	xix
Author.....	xxi
1. Background and Introduction.....	1
Part 1: Background	1
1.1 Differential Equations.....	1
1.1.1 Linear, First-Order ODEs.....	1
1.1.2 Second-Order ODEs with Constant Coefficients	2
1.1.2.1 Homogeneous Solution	2
1.1.2.2 Particular Solution	3
1.1.3 Method of Undetermined Coefficients	3
1.2 Matrix Analysis.....	4
1.2.1 Matrix Operations.....	5
1.2.2 Matrix Transpose.....	5
1.2.3 Special Matrices	6
1.2.4 Determinant of a Matrix.....	6
1.2.5 Properties of Determinant.....	6
1.2.5.1 Cramer's Rule.....	7
1.2.6 Inverse of a Matrix.....	8
1.2.7 Properties of Inverse.....	9
1.2.8 Solving a Linear System of Equations	9
1.3 Matrix Eigenvalue Problem.....	9
1.3.1 Solving the Eigenvalue Problem.....	10
1.3.2 Similarity Transformation.....	11
1.3.3 Matrix Diagonalization.....	11
1.3.4 Eigenvalue Properties of Matrices.....	12
Part 2: Introduction to Numerical Methods.....	12
1.4 Errors and Approximations.....	12
1.4.1 Sources of Computational Error	12
1.4.2 Binary and Hexadecimal Numbers	13
1.4.3 Floating Point and Rounding Errors.....	13
1.4.4 Round-Off: Chopping and Rounding.....	14
1.4.5 Absolute and Relative Errors	15
1.4.6 Error Bound	16
1.4.7 Transmission of Error from a Source to the Final Result.....	16
1.4.8 Subtraction of Nearly Equal Numbers	17
1.5 Iterative Methods	19
1.5.1 Fundamental Iterative Method	20
1.5.2 Rate of Convergence of an Iterative Method.....	21
Problem Set (Chapter 1).....	22
2. Introduction to MATLAB®	27
2.1 MATLAB Built-In Functions	27

2.1.1	Rounding Commands.....	27
2.1.2	Relational Operators.....	28
2.1.3	Format Options	28
2.2	Vectors and Matrices	29
2.2.1	Linspace.....	30
2.2.2	Matrices	30
2.2.3	Determinant, Transpose, and Inverse.....	32
2.2.4	Slash Operators	33
2.2.5	Element-by-Element Operations.....	33
2.2.6	Diagonal Matrices and Diagonals of a Matrix.....	34
2.3	Symbolic Math Toolbox.....	36
2.3.1	Anonymous Functions.....	38
2.3.2	MATLAB Function	38
2.3.3	Differentiation.....	39
2.3.4	Partial Derivatives	40
2.3.5	Integration.....	40
2.4	Program Flow Control.....	41
2.4.1	for Loop	41
2.4.2	The if Command.....	42
2.4.3	while Loop	43
2.5	Displaying Formatted Data	43
2.5.1	Differential Equations.....	44
2.6	Plotting.....	45
2.6.1	subplot.....	45
2.6.2	Plotting Analytical Expressions	46
2.6.3	Multiple Plots.....	46
2.7	User-Defined Functions and Script Files.....	47
2.7.1	Setting Default Values for Input Variables.....	49
2.7.2	Creating Script Files	50
	Problem Set (Chapter 2).....	51

3.	Numerical Solution of Equations of a Single Variable.....	55
3.1	Numerical Solution of Equations.....	55
3.2	Bisection Method	55
3.2.1	MATLAB Built-In Function <code>fzero</code>	60
3.3	Regula Falsi Method (Method of False Position).....	61
3.3.1	Modified Regula Falsi Method	64
3.4	Fixed-Point Method	65
3.4.1	Selection of a Suitable Iteration Function	66
3.4.2	A Note on Convergence	67
3.4.3	Rate of Convergence of the Fixed-Point Iteration.....	71
3.5	Newton's Method (Newton–Raphson Method)	72
3.5.1	Rate of Convergence of Newton's Method	76
3.5.2	A Few Notes on Newton's Method	77
3.5.3	Modified Newton's Method for Roots with Multiplicity 2 or Higher	78
3.6	Secant Method.....	81
3.6.1	Rate of Convergence of Secant Method	83
3.6.2	A Few Notes on Secant Method	83

3.7	Equations with Several Roots.....	83
3.7.1	Finding Roots to the Right of a Specified Point.....	83
3.7.2	Finding Several Roots in an Interval Using <code>fzero</code>	84
	Problem Set (Chapter 3).....	88
4.	Numerical Solution of Systems of Equations	95
4.1	Linear Systems of Equations	95
4.2	Numerical Solution of Linear Systems.....	96
4.3	Gauss Elimination Method.....	96
4.3.1	Choosing the Pivot Row: Partial Pivoting with Row Scaling	98
4.3.2	Permutation Matrices	99
4.3.3	Counting the Number of Operations.....	102
4.3.3.1	Elimination.....	102
4.3.3.2	Back Substitution.....	103
4.3.4	Tridiagonal Systems	103
4.3.4.1	Thomas Method	104
4.3.4.2	MATLAB Built-In Function <code>\</code>	106
4.4	LU Factorization Methods	107
4.4.1	Doolittle Factorization.....	107
4.4.2	Finding L and U Using Steps of Gauss Elimination.....	108
4.4.3	Finding L and U Directly.....	108
4.4.3.1	Doolittle's Method to Solve a Linear System.....	110
4.4.3.2	Operations Count	112
4.4.4	Cholesky Factorization.....	112
4.4.4.1	Cholesky's Method to Solve a Linear System.....	113
4.4.4.2	Operations Count	115
4.4.4.3	MATLAB Built-In Functions <code>lu</code> and <code>chol</code>	115
4.5	Iterative Solution of Linear Systems.....	116
4.5.1	Vector Norms.....	116
4.5.2	Matrix Norms.....	118
4.5.2.1	Compatibility of Vector and Matrix Norms	119
4.5.3	General Iterative Method.....	120
4.5.3.1	Convergence of the General Iterative Method	120
4.5.4	Jacobi Iteration Method	121
4.5.4.1	Convergence of the Jacobi Iteration Method	122
4.5.5	Gauss–Seidel Iteration Method.....	125
4.5.5.1	Convergence of the Gauss–Seidel Iteration Method	127
4.5.6	Indirect Methods versus Direct Methods for Large Systems.....	130
4.6	Ill-Conditioning and Error Analysis.....	131
4.6.1	Condition Number.....	131
4.6.2	Ill-Conditioning	132
4.6.2.1	Indicators of Ill-Conditioning.....	133
4.6.3	Computational Error	133
4.6.3.1	Consequences of Ill-Conditioning.....	135
4.6.4	Effects of Parameter Changes on the Solution	136
4.7	Systems of Nonlinear Equations.....	138
4.7.1	Newton's Method for a System of Nonlinear Equations.....	138
4.7.1.1	Newton's Method for Solving a System of Two Nonlinear Equations	138

4.7.1.2	Newton's Method for Solving a System of n Nonlinear Equations	142
4.7.1.3	Convergence of Newton's Method.....	142
4.7.2	Fixed-Point Iteration Method for a System of Nonlinear Equations	143
4.7.2.1	Convergence of the Fixed-Point Iteration Method.....	143
	Problem Set (Chapter 4).....	146
5.	Curve Fitting and Interpolation	161
5.1	Least-Squares Regression	161
5.2	Linear Regression.....	162
5.2.1	Deciding a "Best" Fit Criterion	163
5.2.2	Linear Least-Squares Regression.....	164
5.3	Linearization of Nonlinear Data.....	167
5.3.1	Exponential Function	167
5.3.2	Power Function	167
5.3.3	Saturation Function	168
5.4	Polynomial Regression.....	172
5.4.1	Quadratic Least-Squares Regression	174
5.4.2	Cubic Least-Squares Regression.....	176
5.4.3	MATLAB Built-In Functions <code>polyfit</code> and <code>polyval</code>	178
5.5	Polynomial Interpolation	179
5.5.1	Lagrange Interpolating Polynomials	180
5.5.2	Drawbacks of Lagrange Interpolation	183
5.5.3	Newton Divided-Difference Interpolating Polynomials	184
5.5.4	Special Case: Equally-Spaced Data	190
5.5.5	Newton Forward-Difference Interpolating Polynomials.....	191
5.6	Spline Interpolation	193
5.6.1	Linear Splines.....	194
5.6.2	Quadratic Splines.....	195
5.6.2.1	Function Values at the Endpoints (2 Equations)	195
5.6.2.2	Function Values at the Interior Knots ($2n - 2$ Equations)	196
5.6.2.3	First Derivatives at the Interior Knots ($n - 1$ Equations)	196
5.6.2.4	Second Derivative at the Left Endpoint is Zero (1 Equation)....	196
5.6.3	Cubic Splines	198
5.6.3.1	Clamped Boundary Conditions	199
5.6.3.2	Free Boundary Conditions.....	199
5.6.4	Construction of Cubic Splines: Clamped Boundary Conditions	199
5.6.5	Construction of Cubic Splines: Free Boundary Conditions.....	204
5.6.6	MATLAB Built-In Functions <code>interp1</code> and <code>spline</code>	205
5.6.7	Boundary Conditions	207
5.6.8	Interactive Curve Fitting and Interpolation in MATLAB	208
5.7	Fourier Approximation and Interpolation	209
5.7.1	Sinusoidal Curve Fitting.....	209
5.7.1.1	Fourier Approximation	210
5.7.1.2	Fourier Interpolation	210
5.7.2	Linear Transformation of Data	210
5.7.3	Discrete Fourier Transform	215
5.7.4	Fast Fourier Transform.....	216
5.7.4.1	Sande-Tukey Algorithm ($N = 2^p$, $p = \text{integer}$)	217

5.7.4.2	Case Study: $N = 2^3 = 8$	218
5.7.4.3	Cooley–Tukey Algorithm ($N = 2^p, p = \text{integer}$).....	219
5.7.5	MATLAB Built-In Function <code>fft</code>	220
5.7.5.1	Interpolation Using <code>fft</code>	220
Problem Set (Chapter 5).....		223
6.	Numerical Differentiation and Integration	249
6.1	Numerical Differentiation	249
6.2	Finite-Difference Formulas for Numerical Differentiation.....	249
6.2.1	Finite-Difference Formulas for the First Derivative	250
6.2.1.1	Two-Point Backward Difference Formula	250
6.2.1.2	Two-Point Forward Difference Formula.....	251
6.2.1.3	Two-Point Central Difference Formula	251
6.2.1.4	Three-Point Backward Difference Formula	252
6.2.1.5	Three-Point Forward Difference Formula.....	253
6.2.2	Finite-Difference Formulas for the Second Derivative.....	254
6.2.2.1	Three-Point Backward Difference Formula	254
6.2.2.2	Three-Point Forward Difference Formula.....	254
6.2.2.3	Three-Point Central Difference Formula	255
6.2.2.4	Summary of Finite-Difference Formulas for First to Fourth Derivatives	256
6.2.3	Estimate Improvement: Richardson’s Extrapolation	256
6.2.4	Richardson’s Extrapolation for Discrete Sets of Data	259
6.2.5	Derivative Estimates for Non-Evenly Spaced Data.....	259
6.2.6	MATLAB Built-In Functions <code>diff</code> and <code>polyder</code>	260
6.3	Numerical Integration: Newton–Cotes Formulas.....	261
6.3.1	Newton–Cotes Formulas	262
6.3.2	Rectangular Rule	262
6.3.2.1	Composite Rectangular Rule.....	262
6.3.3	Error Estimate for Composite Rectangular Rule.....	264
6.3.4	Trapezoidal Rule	266
6.3.4.1	Composite Trapezoidal Rule.....	267
6.3.4.2	Error Estimate for Composite Trapezoidal Rule.....	267
6.3.5	Simpson’s Rules.....	269
6.3.5.1	Simpson’s 1/3 Rule.....	269
6.3.5.2	Composite Simpson’s 1/3 Rule.....	270
6.3.5.3	Error Estimate for Composite Simpson’s 1/3 Rule	270
6.3.5.4	Simpson’s 3/8 Rule	271
6.3.5.5	Composite Simpson’s 3/8 Rule	272
6.3.5.6	Error Estimate for Composite Simpson’s 3/8 Rule	273
6.3.6	MATLAB Built-In Functions <code>quad</code> and <code>trapz</code>	273
6.4	Numerical Integration of Analytical Functions: Romberg Integration, Gaussian Quadrature	275
6.4.1	Romberg Integration	275
6.4.1.1	Richardson’s Extrapolation.....	275
6.4.1.2	Romberg Integration.....	278
6.4.2	Gaussian Quadrature	280
6.5	Improper Integrals.....	285
Problem Set (Chapter 6).....		286

7. Numerical Solution of Initial-Value Problems	301
7.1 Introduction.....	301
7.2 One-Step Methods.....	301
7.3 Euler’s Method.....	302
7.3.1 Error Analysis for Euler’s Method.....	305
7.3.2 Calculation of Local and Global Truncation Errors.....	305
7.3.3 Higher-Order Taylor Methods.....	307
7.4 Runge–Kutta Methods.....	309
7.4.1 Second-Order Runge–Kutta (RK2) Methods.....	310
7.4.1.1 Improved Euler’s Method.....	311
7.4.1.2 Heun’s Method.....	311
7.4.1.3 Ralston’s Method.....	312
7.4.1.4 Graphical Representation of Heun’s Method.....	312
7.4.2 Third-Order Runge–Kutta (RK3) Methods.....	315
7.4.2.1 The Classical RK3 Method.....	315
7.4.2.2 Heun’s RK3 Method.....	315
7.4.3 Fourth-Order Runge–Kutta (RK4) Methods.....	316
7.4.3.1 The Classical RK4 Method.....	317
7.4.4 Higher-Order Runge–Kutta Methods.....	319
7.4.5 Selection of Optimal Step Size: Runge–Kutta Fehlberg (RKF) Method.....	320
7.4.5.1 Adjustment of Step Size.....	321
7.5 Multistep Methods.....	322
7.5.1 Adams–Bashforth Method.....	323
7.5.1.1 Second-Order Adams–Bashforth Formula.....	324
7.5.1.2 Third-Order Adams–Bashforth Formula.....	324
7.5.1.3 Fourth-Order Adams–Bashforth Formula.....	324
7.5.2 Adams–Moulton Method.....	325
7.5.2.1 Second-Order Adams–Moulton Formula.....	326
7.5.2.2 Third-Order Adams–Moulton Formula.....	326
7.5.2.3 Fourth-Order Adams–Moulton Formula.....	326
7.5.3 Predictor–Corrector Methods.....	326
7.5.3.1 Heun’s Predictor–Corrector Method.....	327
7.5.3.2 Adams–Bashforth–Moulton (ABM) Predictor–Corrector Method.....	327
7.6 Systems of Ordinary Differential Equations.....	330
7.6.1 Transformation into a System of First-Order ODEs.....	330
7.6.1.1 State Variables.....	330
7.6.1.2 Notation.....	330
7.6.1.3 State-Variable Equations.....	330
7.6.2 Numerical Solution of a System of First-Order ODEs.....	332
7.6.2.1 Euler’s Method for Systems.....	332
7.6.2.2 Heun’s Method for Systems.....	335
7.6.2.3 Classical RK4 Method for Systems.....	336
7.7 Stability.....	340
7.7.1 Euler’s Method.....	341
7.7.2 Euler’s Implicit Method.....	341
7.8 Stiff Differential Equations.....	343
7.9 MATLAB Built-In Functions for Solving Initial-Value Problems.....	345

7.9.1	Non-Stiff Equations	345
7.9.2	A Single First-Order IVP.....	345
7.9.3	Setting ODE Solver Options.....	347
7.9.4	A System of First-Order IVPs.....	348
7.9.5	Stiff Equations	349
	Problem Set (Chapter 7).....	350
8.	Numerical Solution of Boundary-Value Problems	367
8.1	Second-Order BVP	367
8.2	Boundary Conditions	367
8.3	Higher-Order BVP	368
8.4	Shooting Method.....	368
8.5	Finite-Difference Method.....	374
8.5.1	Boundary-Value Problems with Mixed Boundary Conditions.....	379
8.6	MATLAB Built-In Function <code>bvp4c</code> for Boundary-Value Problems.....	381
8.6.1	Second-Order BVP.....	382
	Problem Set (Chapter 8).....	386
9.	Matrix Eigenvalue Problem.....	393
9.1	Matrix Eigenvalue Problem	393
9.2	Power Method: Estimation of the Dominant Eigenvalue	393
9.2.1	Different Cases of Dominant Eigenvalue.....	395
9.2.2	Algorithm for the Power Method.....	395
9.3	Inverse Power Method: Estimation of the Smallest Eigenvalue.....	398
9.4	Shifted Inverse Power Method: Estimation of the Eigenvalue Nearest a Specified Value.....	399
9.4.1	Notes on the Shifted Inverse Power Method	400
9.5	Shifted Power Method.....	401
9.5.1	Strategy to Estimate All Eigenvalues of a Matrix	401
9.6	MATLAB Built-In Function <code>eig</code>	403
9.7	Deflation Methods	403
9.7.1	Wielandt's Deflation Method	404
9.7.2	Deflation Process.....	405
9.8	Householder Tridiagonalization and QR Factorization Methods	407
9.8.1	Householder's Tridiagonalization Method (Symmetric Matrices)	408
9.8.2	Determination of Symmetric Orthogonal P_k ($k = 1, 2, \dots, n - 2$).....	409
9.8.3	QR Factorization Method	411
9.8.4	Determination of Q_k and R_k Matrices	412
9.8.5	Structure of L_k ($k = 2, 3, \dots, n$).....	412
9.9	MATLAB Built-In Function <code>qr</code>	413
9.10	A Note on the Terminating Condition Used in <code>HouseholderQR</code>	414
9.11	Transformation to Hessenberg Form (Nonsymmetric Matrices).....	417
	Problem Set (Chapter 9).....	418
10.	Numerical Solution of Partial Differential Equations	423
10.1	Introduction	423
10.2	Elliptic Partial Differential Equations.....	424
10.2.1	Dirichlet Problem.....	424

10.2.2	Alternating Direction Implicit (ADI) Methods.....	428
10.2.2.1	Peaceman–Rachford Alternating Direction Implicit (PRADI) Method.....	429
10.2.3	Neumann Problem	433
10.2.3.1	Existence of a Solution for the Neumann Problem	435
10.2.4	Mixed Problem.....	436
10.2.5	More Complex Regions.....	437
10.3	Parabolic Partial Differential Equations.....	440
10.3.1	Finite-Difference Method	440
10.3.1.1	Stability and Convergence of the Finite-Difference Method.....	441
10.3.2	Crank–Nicolson Method.....	443
10.3.2.1	Crank–Nicolson (CN) Method versus Finite-Difference (FD) Method.....	446
10.4	Hyperbolic Partial Differential Equations	448
10.4.1	Starting the Procedure	449
	Problem Set (Chapter 10).....	452
	Index	461

Preface

It has been nearly 4 years since the first edition of *Numerical Methods for Engineers and Scientists Using MATLAB®* was published. During this time, most of the material in the first edition has been rigorously class tested, resulting in many enhancements and modifications to make the new edition even more effective and user-friendly.

As in the first edition, the primary objective of this book is to provide the reader with a broad knowledge of the fundamentals of numerical methods utilized in various disciplines in engineering and science. The powerful software MATLAB is introduced at the outset and is assimilated throughout the book to perform symbolic, graphical, and numerical tasks. The textbook, written at the junior/senior level, methodically covers a wide array of techniques ranging from curve fitting a set of data to numerically solving initial- and boundary-value problems. Each method is accompanied by at least one fully worked-out example, followed by either a user-defined function or a MATLAB script file. MATLAB built-in functions are also presented for each main topic covered.

This book consists of 10 chapters. **Chapter 1** presents the necessary background material and is divided into two parts: (1) differential equations, matrix analysis, and the matrix eigenvalue problem, and (2) computational errors, approximations, iterative methods, and rates of convergence.

Chapter 2 gives an in-depth introduction to the essentials of MATLAB as related to numerical methods. The chapter addresses fundamental features such as built-in functions and commands, formatting options, vector and matrix operations, program flow control, symbolic operations, and plotting capabilities. The reader also learns how to write a user-defined function or a MATLAB script file to perform specific tasks.

Chapters 3 and **4** introduce numerical methods for solving equations. **Chapter 3** focuses on finding roots of equations of a single variable, while **Chapter 4** covers methods for solving linear and nonlinear systems of equations.

Chapter 5 is completely devoted to curve fitting and interpolation techniques, including the fast Fourier transform (FFT). **Chapter 6** covers numerical differentiation and integration methods. **Chapters 7** and **8** present numerical methods for solving initial-value problems and boundary-value problems, respectively.

Chapter 9 covers the numerical solution of the matrix eigenvalue problem, which entails techniques to approximate a few or all eigenvalues of a matrix.

Chapter 10 presents numerical methods for solving elliptic, parabolic, and hyperbolic partial differential equations, specifically those that frequently arise in engineering and science.



Pedagogy of the Book

The book is written in a user-friendly fashion that intends to make the material easy to follow and understand by the reader. The topics are presented systematically using the following format:

- Each newly introduced method is accompanied by at least one fully worked-out example showing all details.
- This is followed by a user-defined function, or a script file, that utilizes the method to perform a desired task.
- The hand-calculated results are then confirmed through the execution of the user-defined function or the script file.
- When available, built-in functions are executed for reconfirmation.
- Plots are regularly generated to shed light on the accuracy and implication of the numerical results.

Exercises

A large set of exercises, of various levels of difficulty, appears at the end of each chapter and can be worked out either using a

-  Hand calculator, or
-  MATLAB.

In many instances, the reader is asked to prepare a user-defined function, or a script file, that implements a specific technique. In many cases, these require simple revisions to those already presented in the chapter.

Ancillary Material

The following will be provided to the instructors adopting the book:

- An instructor's solutions manual (in PDF format), featuring complete solution details of all exercises, prepared by the author.
- A web download containing all user-defined functions used throughout the book, available at <https://www.crcpress.com/Numerical-Methods-for-Engineers-and-Scientists-Using-MATLAB-Second-Edition/Esfandiari/p/book/9781498777421>.

New to This Edition

- **Chapter 2** (Introduction to MATLAB) has been extensively reinforced so that it now covers virtually all features of MATLAB that are consistently used throughout the book.

- Many of the user-defined functions have been revised to become more robust and versatile.
- Several worked-out examples have been either entirely changed or modified to illustrate the important details of the methods under consideration.
- A large proportion of the end-of-chapter exercises have been carefully revamped so that not only their objectives are clear to the reader, but also they better represent a wide spectrum of the ideas presented in each chapter.

Ramin S. Esfandiari, PhD

June 2016

1

Background and Introduction

This chapter is divided into two parts. In Part 1, a review of some essential mathematical concepts as related to differential equations and matrix analysis is presented. In Part 2, fundamentals of numerical methods, such as sources of computational errors, as well as iterations and rates of convergence are introduced. The materials presented here will be fully integrated throughout the book.

Part 1: Background

1.1 Differential Equations

Differential equations are divided into two main groups: ordinary differential equations (ODEs) and partial differential equations (PDEs). An equation involving an unknown function and one or more of its derivatives is called a differential equation. When there is only one independent variable, the equation is called an ODE. If the unknown function is a function of several independent variables, the equation is a PDE. For example, $2\dot{x} + x = e^{-t}$ is an ODE involving the unknown function $x(t)$, its first derivative with respect to t , as well as a given function e^{-t} . Similarly, $t\ddot{x} - x^2\dot{x} = \sin t$ is an ODE relating $x(t)$ and its first and second derivatives with respect to t , as well as the function $\sin t$. The derivative of the highest order of the unknown function with respect to the independent variable is the order of the ODE. For instance, $2\dot{x} + x = e^{-t}$ is of order 1, while $t\ddot{x} - x^2\dot{x} = \sin t$ is of order 2. PDEs are discussed in [Chapter 10](#).

Consider an n th-order ODE in the form

$$a_n x^{(n)} + a_{n-1} x^{(n-1)} + \cdots + a_1 \dot{x} + a_0 x = F(t) \quad (1.1)$$

where $x = x(t)$ and $x^{(n)} = d^n x / dt^n$. If all coefficients a_0, a_1, \dots, a_n are either constants or functions of the independent variable t , then the ODE is linear. Otherwise, it is nonlinear. If $F(t) \equiv 0$, the ODE is homogeneous. Otherwise, it is nonhomogeneous. Therefore $2\dot{x} + x = e^{-t}$ is linear, $t\ddot{x} - x^2\dot{x} = \sin t$ is nonlinear, and both are nonhomogeneous.

1.1.1 Linear, First-Order ODEs

A linear, first-order ODE can be expressed as

$$a_1 \dot{x} + a_0 x = F(t) \quad \xrightarrow{\text{Divide by } a_1} \quad \dot{x} + g(t)x = f(t) \quad (1.2)$$

A general solution for Equation 1.2 is obtained as

$$x(t) = e^{-h} \left[\int e^h f(t) dt + c \right], \quad h(t) = \int g(t) dt, \quad c = \text{const} \quad (1.3)$$

A particular solution is obtained when an initial condition is prescribed. Assuming t_0 is the initial value of t , a first-order initial-value problem (IVP) is described as

$$\dot{x} + g(t)x = f(t), \quad x(t_0) = x_0$$

EXAMPLE 1.1: LINEAR, FIRST-ORDER IVP

Find the particular solution of the following IVP:

$$3\dot{x} + 2x = e^{-t/2}, \quad x(0) = \frac{1}{3}$$

Solution

We first rewrite the ODE in the standard form of Equation 1.2, as $\dot{x} + \frac{2}{3}x = \frac{1}{3}e^{-t/2}$ so that $g(t) = \frac{2}{3}$, $f(t) = \frac{1}{3}e^{-t/2}$. By Equation 1.3, a general solution is obtained as

$$h = \int \frac{2}{3} dt = \frac{2}{3}t, \quad x(t) = e^{-2t/3} \left[\int e^{2t/3} \frac{1}{3} e^{-t/2} dt + c \right] = 2e^{-t/2} + ce^{-2t/3}$$

Applying the initial condition, we find

$$x(0) = 2 + c = \frac{1}{3} \Rightarrow c = -\frac{5}{3}$$

Therefore,

$$x(t) = 2e^{-t/2} - \frac{5}{3}e^{-2t/3}$$

1.1.2 Second-Order ODEs with Constant Coefficients

A second-order ODE in the standard form, with constant coefficients, is expressed as

$$\ddot{x} + a_1\dot{x} + a_0x = f(t), \quad a_1, a_0 = \text{const} \quad (1.4)$$

The corresponding second-order IVP consists of Equation 1.4 accompanied by two initial conditions. A general solution of Equation 1.4 is a superposition of the homogeneous solution $x_h(t)$ and the particular solution $x_p(t)$.

1.1.2.1 Homogeneous Solution

The homogeneous solution is the solution of the homogeneous equation

$$\ddot{x} + a_1\dot{x} + a_0x = 0 \quad (1.5)$$

Assuming a solution in the form $x(t) = e^{\lambda t}$, with λ to be determined, substituting into Equation 1.5, and using the fact that $e^{\lambda t} \neq 0$, we find

$$\lambda^2 + a_1\lambda + a_0 = 0$$

This is known as the characteristic equation. The solution of Equation 1.5 is determined according to the nature of the two roots of the characteristic equation of the ODE. These roots, labeled λ_1 and λ_2 , are called the characteristic values.

1. When $\lambda_1 \neq \lambda_2$ (real), the homogeneous solution is

$$x_h(t) = c_1 e^{\lambda_1 t} + c_2 e^{\lambda_2 t}$$

2. When $\lambda_1 = \lambda_2$, we have

$$x_h(t) = c_1 e^{\lambda_1 t} + c_2 t e^{\lambda_1 t}$$

3. When $\bar{\lambda}_1 = \lambda_2$ (complex conjugates), and $\lambda_1 = \sigma + i\omega$, we find

$$x_h(t) = e^{\sigma t} (c_1 \cos \omega t + c_2 \sin \omega t)$$

EXAMPLE 1.2: HOMOGENEOUS, SECOND-ORDER ODE WITH CONSTANT COEFFICIENTS

Find a general solution of

$$\ddot{x} + 5\dot{x} + 4x = 0$$

Solution

The characteristic equation is formed as $\lambda^2 + 5\lambda + 4 = 0$ so that the characteristic values are $\lambda_1 = -1$, $\lambda_2 = -4$, and

$$x(t) = c_1 e^{-t} + c_2 e^{-4t}$$

1.1.2.2 Particular Solution

The particular solution of Equation 1.4 is determined by the function $f(t)$ and how it is related to the independent functions that constitute the homogeneous solution. The particular solution is obtained by the method of undetermined coefficients. This method is limited in its applications only to cases where $f(t)$ is a polynomial, an exponential function, a sinusoidal function, or any of their combinations.

1.1.3 Method of Undetermined Coefficients

Table 1.1 lists different scenarios and the corresponding recommended $x_p(t)$. These recommended forms are subject to modification in some special cases as follows. If $x_p(t)$ contains a term that coincides with a solution of the homogeneous equation, and that the solution

TABLE 1.1

Method of Undetermined Coefficients

Term in $f(t)$	Recommended $x_p(t)$
$A_n t^n + A_{n-1} t^{n-1} + \dots + A_1 t + A_0$	$K_n t^n + K_{n-1} t^{n-1} + \dots + K_1 t + K_0$
Ae^{at}	Ke^{at}
$A \cos \alpha t$ or $A \sin \alpha t$	$K_1 \cos \alpha t + K_2 \sin \alpha t$
$Ae^{at} \cos \alpha t$ or $Ae^{at} \sin \alpha t$	$e^{at}(K_1 \cos \alpha t + K_2 \sin \alpha t)$

corresponds to a non-repeated characteristic value, then the recommended $x_p(t)$ must be multiplied by t . If the said characteristic value is repeated, then $x_p(t)$ is multiplied by t^2 .

EXAMPLE 1.3: SECOND-ORDER IVP

Solve the following second-order IVP:

$$\ddot{x} + 5\dot{x} + 4x = \frac{1}{4}e^{-t}, \quad x(0) = 0, \quad \dot{x}(0) = -\frac{1}{6}$$

Solution

The homogeneous solution was previously found in Example 1.2, as $x_h(t) = c_1 e^{-t} + c_2 e^{-4t}$. Since $f(t) = \frac{1}{4}e^{-t}$, Table 1.1 recommends $x_p(t) = Ke^{-t}$. However, e^{-t} is one of the independent functions in the homogeneous solution, thus $x_p(t)$ must be modified. Since e^{-t} is associated with a non-repeated characteristic value ($\lambda = -1$), we multiply the recommended $x_p(t)$ by t to obtain $x_p(t) = Kte^{-t}$. Substitution into the ODE, and collecting like terms, yields

$$3Ke^{-t} = \frac{1}{4}e^{-t} \Rightarrow K = \frac{1}{12} \Rightarrow x_p(t) = \frac{1}{12}te^{-t}$$

Therefore, a general solution is formed as $x(t) = c_1 e^{-t} + c_2 e^{-4t} + \frac{1}{12}te^{-t}$. Applying the initial conditions,

$$\begin{aligned} c_1 + c_2 &= 0 \\ -c_1 - 3c_2 + \frac{1}{12} &= -\frac{1}{6} \end{aligned} \quad \begin{array}{l} \text{Solve} \\ \Rightarrow \end{array} \quad \begin{aligned} c_1 &= -\frac{1}{12} \\ c_2 &= \frac{1}{12} \end{aligned}$$

Therefore, $x(t) = \frac{1}{12}(e^{-4t} - e^{-t} + te^{-t})$.

1.2 Matrix Analysis

An n -dimensional vector \mathbf{v} is an ordered set of n scalars, written as

$$\mathbf{v} = \begin{Bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{Bmatrix}$$

where each v_i ($i = 1, 2, \dots, n$) is a component of vector \mathbf{v} . A matrix is a collection of numbers (real or complex) or possibly functions, arranged in a rectangular array and enclosed by square brackets. Each of the elements in a matrix is called an entry of the matrix. The horizontal and vertical levels are the rows and columns of the matrix, respectively. The number of rows and columns of a matrix determine its size. If a matrix \mathbf{A} has m rows and n columns, then its size is $m \times n$. A matrix is called square if the number of its rows and columns are the same. Otherwise, it is rectangular. Matrices are denoted by bold-faced capital letters, such as \mathbf{A} . The abbreviated form of an $m \times n$ matrix is

$$\mathbf{A} = [a_{ij}]_{m \times n}$$

where a_{ij} is known as the (i, j) entry of \mathbf{A} , located at the intersection of the i th row and the j th column of \mathbf{A} . For instance, a_{32} is the entry at the intersection of the third row and the second column of \mathbf{A} . In a square matrix $\mathbf{A}_{n \times n}$, the elements $a_{11}, a_{22}, \dots, a_{nn}$ are the diagonal entries.

Two matrices $\mathbf{A} = [a_{ij}]$ and $\mathbf{B} = [b_{ij}]$ are equal if they have the same size and the same respective entries. A submatrix of \mathbf{A} is generated by deleting some rows and/or columns of \mathbf{A} .

1.2.1 Matrix Operations

The sum of $\mathbf{A} = [a_{ij}]_{m \times n}$ and $\mathbf{B} = [b_{ij}]_{m \times n}$ is

$$\mathbf{C} = [c_{ij}]_{m \times n} = [a_{ij} + b_{ij}]_{m \times n}$$

The product of a scalar k and matrix $\mathbf{A} = [a_{ij}]_{m \times n}$ is

$$k\mathbf{A} = [ka_{ij}]_{m \times n}$$

Consider $\mathbf{A} = [a_{ij}]_{m \times n}$ and $\mathbf{B} = [b_{ij}]_{n \times p}$ so that the number of columns of \mathbf{A} is equal to the number of rows of \mathbf{B} . Then, their product $\mathbf{C} = \mathbf{AB}$ is $m \times p$ whose entries are obtained as

$$c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}, \quad i = 1, 2, \dots, m, \quad j = 1, 2, \dots, p$$

1.2.2 Matrix Transpose

Given $\mathbf{A}_{m \times n}$, its transpose, denoted by \mathbf{A}^T , is an $n \times m$ matrix such that its first row is the first column of \mathbf{A} , its second row is the second column of \mathbf{A} , and so on. Provided all matrix operations are valid,

$$(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$

$$(k\mathbf{A})^T = k\mathbf{A}^T, \quad k = \text{scalar}$$

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

1.2.3 Special Matrices

A square matrix $\mathbf{A} = [a_{ij}]_{n \times n}$ is symmetric if $\mathbf{A}^T = \mathbf{A}$, and skew-symmetric if $\mathbf{A}^T = -\mathbf{A}$. It is upper triangular if $a_{ij} = 0$ for all $i > j$, that is, all entries below the main diagonal are zeros. It is lower triangular if $a_{ij} = 0$ for all $i < j$, that is, all elements above the main diagonal are zeros. It is diagonal if $a_{ij} = 0$ for all $i \neq j$. In the upper and lower triangular matrices, the diagonal elements may be all zeros. However, in a diagonal matrix, at least one diagonal entry must be nonzero. The $n \times n$ identity matrix, denoted by \mathbf{I} , is a diagonal matrix whose every diagonal entry is equal to 1.

1.2.4 Determinant of a Matrix

The determinant of a square matrix $\mathbf{A} = [a_{ij}]_{n \times n}$ is a real scalar denoted by $|\mathbf{A}|$ or $\det(\mathbf{A})$. For $n \geq 2$, the determinant may be calculated using any row or column—with preference given to the row or column with the most zeros. Using the i th row, the determinant is found as

$$|\mathbf{A}| = \sum_{k=1}^n a_{ik} (-1)^{i+k} M_{ik}, \quad i = 1, 2, \dots, n \quad (1.6)$$

In Equation 1.6, M_{ik} is the minor of the entry a_{ik} , defined as the determinant of the $(n-1) \times (n-1)$ submatrix of \mathbf{A} obtained by deleting the i th row and the k th column of \mathbf{A} . The quantity $(-1)^{i+k} M_{ik}$ is the cofactor of a_{ik} and is denoted by C_{ik} . Also note that $(-1)^{i+k}$ is responsible for whether a term is multiplied by +1 or -1. A square matrix is non-singular if its determinant is nonzero. Otherwise, it is called singular.

EXAMPLE 1.4: DETERMINANT

Calculate the determinant of

$$\mathbf{A} = \begin{bmatrix} -1 & -2 & 1 & -3 \\ 2 & 0 & 1 & 4 \\ -1 & 1 & 5 & 2 \\ 3 & -4 & 2 & 3 \end{bmatrix}$$

Solution

We will use the second row since it contains a zero entry.

$$|\mathbf{A}| = -2 \begin{vmatrix} -2 & 1 & -3 \\ -1 & 1 & 2 \\ -4 & 2 & 3 \end{vmatrix} - (-32) \begin{vmatrix} -1 & -2 & -3 \\ -1 & 1 & 2 \\ 3 & -4 & 3 \end{vmatrix} + 4 \begin{vmatrix} -1 & -2 & 1 \\ -1 & 1 & 5 \\ 3 & -4 & 2 \end{vmatrix} = -2(-99) - (-32) + 4(-55) = 10$$

Note that each of the individual 3×3 determinants is calculated via Equation 1.6.

1.2.5 Properties of Determinant

- The determinant of a matrix product is the product of individual determinants:
 $|\mathbf{AB}| = |\mathbf{A}||\mathbf{B}|$.

- The determinant of a matrix and its transpose are the same: $|\mathbf{A}^T| = |\mathbf{A}|$.
- The determinant of a lower triangular, upper triangular, or diagonal matrix is the product of the diagonal entries.
- If any rows or columns of \mathbf{A} are linearly dependent, then $|\mathbf{A}| = 0$.

1.2.5.1 Cramer's Rule

Consider a linear system of n algebraic equations in n unknowns x_1, x_2, \dots, x_n in the form

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases} \quad (1.7)$$

where a_{ij} ($i, j = 1, 2, \dots, n$) and b_i ($i = 1, 2, \dots, n$) are known constants, and a_{ij} 's are the coefficients. Equation 1.7 can be expressed in matrix form, as

$$\mathbf{Ax} = \mathbf{b}$$

with

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}_{n \times n}, \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{Bmatrix}_{n \times 1}, \quad \mathbf{b} = \begin{Bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{Bmatrix}_{n \times 1}$$

Assuming \mathbf{A} is non-singular, each unknown x_k ($k = 1, 2, \dots, n$) is uniquely determined via

$$x_k = \frac{\Delta_k}{\Delta}$$

where determinants Δ and Δ_k are described as

$$\Delta = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}, \quad \Delta_k = \begin{vmatrix} a_{11} & \dots & \text{kth column of } \Delta & \dots & a_{1n} \\ a_{21} & \dots & b_2 & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & \dots & b_n & \dots & a_{nn} \end{vmatrix}$$

EXAMPLE 1.5: CRAMER'S RULE

Solve the following system using Cramer's rule:

$$\begin{bmatrix} 2 & 3 & -1 \\ -1 & 2 & 1 \\ 1 & -3 & -2 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} -3 \\ -6 \\ 9 \end{Bmatrix}$$

Solution

The determinants are calculated as

$$\Delta = \begin{vmatrix} 2 & 3 & -1 \\ -1 & 2 & 1 \\ 1 & -3 & -2 \end{vmatrix} = -6, \quad \Delta_1 = \begin{vmatrix} -3 & 3 & -1 \\ -6 & 2 & 1 \\ 9 & -3 & -2 \end{vmatrix} = -6, \quad \Delta_2 = \begin{vmatrix} 2 & -3 & -1 \\ -1 & -6 & 1 \\ 1 & 9 & -2 \end{vmatrix} = 12,$$
$$\Delta_3 = \begin{vmatrix} 2 & 3 & -3 \\ -1 & 2 & -6 \\ 1 & -3 & 9 \end{vmatrix} = 6$$

The unknowns are then found as

$$x_1 = \frac{\Delta_1}{\Delta} = 1, \quad x_2 = \frac{\Delta_2}{\Delta} = -2, \quad x_3 = \frac{\Delta_3}{\Delta} = -1 \quad \text{Solution vector} \Rightarrow \begin{Bmatrix} 1 \\ -2 \\ -1 \end{Bmatrix}$$

1.2.6 Inverse of a Matrix

The inverse of a square matrix $\mathbf{A}_{n \times n}$ is denoted by \mathbf{A}^{-1} with the property $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ where \mathbf{I} is the $n \times n$ identity matrix. The inverse of \mathbf{A} exists only if \mathbf{A} is non-singular, $|\mathbf{A}| \neq 0$, and is obtained by using the adjoint matrix of \mathbf{A} , denoted by $\text{adj}(\mathbf{A})$.

1.2.6.1 Adjoint Matrix

If $\mathbf{A} = [a_{ij}]_{n \times n}$, then the adjoint of \mathbf{A} is defined as

$$\text{adj}(\mathbf{A}) = \begin{bmatrix} (-1)^{1+1}M_{11} & (-1)^{2+1}M_{21} & \dots & (-1)^{n+1}M_{n1} \\ (-1)^{1+2}M_{12} & (-1)^{2+2}M_{22} & \dots & (-1)^{n+2}M_{n2} \\ \dots & \dots & \dots & \dots \\ (-1)^{1+n}M_{1n} & (-1)^{2+n}M_{2n} & \dots & (-1)^{n+n}M_{nn} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{21} & \dots & C_{n1} \\ C_{12} & C_{22} & \dots & C_{n2} \\ \dots & \dots & \dots & \dots \\ C_{1n} & C_{2n} & \dots & C_{nn} \end{bmatrix} \quad (1.8)$$

where M_{ij} is the minor of a_{ij} and $C_{ij} = (-1)^{i+j}M_{ij}$ is the cofactor of a_{ij} . Note that each minor M_{ij} (or cofactor C_{ij}) occupies the (j, i) position in the adjoint matrix. Then,

$$\mathbf{A}^{-1} = \frac{1}{|\mathbf{A}|} \text{adj}(\mathbf{A}) \quad (1.9)$$

EXAMPLE 1.6: INVERSE

Find the inverse of

$$\mathbf{A} = \begin{bmatrix} 3 & 1 & 0 \\ 1 & -1 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

Solution

We first calculate $|\mathbf{A}| = -8$. Following the strategy outlined in Equation 1.8, the adjoint matrix of \mathbf{A} is obtained as

$$\text{adj}(\mathbf{A}) = \begin{bmatrix} -3 & -1 & 2 \\ 1 & 3 & -6 \\ 2 & -2 & -4 \end{bmatrix}$$

Finally, by Equation 1.9, we have

$$\mathbf{A}^{-1} = \frac{1}{-8} \begin{bmatrix} -3 & -1 & 2 \\ 1 & 3 & -6 \\ 2 & -2 & -4 \end{bmatrix} = \begin{bmatrix} 0.3750 & 0.1250 & -0.2500 \\ -0.1250 & -0.3750 & 0.7500 \\ -0.2500 & 0.2500 & 0.5000 \end{bmatrix}$$

1.2.7 Properties of Inverse

- $(\mathbf{A}^{-1})^{-1} = \mathbf{A}$
- $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$
- $(\mathbf{A}^{-1})^p = (\mathbf{A}^p)^{-1}$, $p = \text{integer} > 0$
- $|\mathbf{A}^{-1}| = 1/|\mathbf{A}|$
- $(\mathbf{A}^{-1})^T = (\mathbf{A}^T)^{-1}$
- Inverse of a symmetric matrix is symmetric.
- Inverse of a diagonal matrix is diagonal whose entries are the reciprocals of the entries of the original matrix.

1.2.8 Solving a Linear System of Equations

A linear system of equations $\mathbf{Ax} = \mathbf{b}$, where \mathbf{A} is non-singular, can be solved as

$$\mathbf{Ax} = \mathbf{b} \quad \begin{array}{l} \text{Pre-multiply} \\ \Rightarrow \\ \text{both sides by } \mathbf{A}^{-1} \end{array} \quad \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

1.3 Matrix Eigenvalue Problem

Consider an $n \times n$ matrix \mathbf{A} , a scalar λ (generally complex), and a nonzero $n \times 1$ vector \mathbf{v} . The eigenvalue problem associated with matrix \mathbf{A} is defined as

$$\mathbf{Av} = \lambda\mathbf{v}, \quad \mathbf{v} \neq \mathbf{0} \tag{1.10}$$

where λ is an eigenvalue of \mathbf{A} , and \mathbf{v} is the eigenvector of \mathbf{A} corresponding to λ . Note that an *eigenvector cannot be a zero vector*.

1.3.1 Solving the Eigenvalue Problem

Rewriting Equation 1.10, we have

$$\mathbf{A}\mathbf{v} - \lambda\mathbf{v} = \mathbf{0} \quad \begin{array}{c} \text{Factor } \mathbf{v} \\ \Rightarrow \\ \text{from the right side} \end{array} \quad [\mathbf{A} - \lambda\mathbf{I}]\mathbf{v} = \mathbf{0} \quad (1.11)$$

where the identity matrix \mathbf{I} has been inserted to make the two terms in brackets size compatible. Equation 1.11 has a non-trivial (nonzero vector) solution if and only if the coefficient matrix is singular, that is,

$$|\mathbf{A} - \lambda\mathbf{I}| = 0 \quad (1.12)$$

This gives the characteristic equation of matrix \mathbf{A} . Since \mathbf{A} is $n \times n$, Equation 1.12 has n roots, which are the eigenvalues of \mathbf{A} . The corresponding eigenvector for each λ is obtained by solving Equation 1.11. Since $\mathbf{A} - \lambda\mathbf{I}$ is singular, it has at least one row dependent on other rows. Therefore, for each λ , Equation 1.11 has infinitely many solutions. A basis of solutions will then represent all eigenvectors associated with λ .

EXAMPLE 1.7: EIGENVALUE PROBLEM

Find the eigenvalues and eigenvectors of

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Solution

The characteristic equation yields the eigenvalues:

$$|\mathbf{A} - \lambda\mathbf{I}| = \begin{vmatrix} 1-\lambda & 0 & 1 \\ 0 & 1-\lambda & 0 \\ 1 & 0 & 1-\lambda \end{vmatrix} = \lambda(\lambda-1)(\lambda-2) = 0 \Rightarrow \lambda = 0, 1, 2$$

Solving Equation 1.11 with $\lambda_1 = 0$, we have

$$[\mathbf{A} - \lambda_1\mathbf{I}]\mathbf{v}_1 = \mathbf{0} \Rightarrow \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \mathbf{v}_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Let the three components of \mathbf{v}_1 be a, b, c . Then, the above system yields $b = 0$ and $a + c = 0$. This implies there is a free variable, which can be either a or c . Letting $a = 1$ leads to $c = -1$, and consequently the eigenvector associated with $\lambda_1 = 0$ is determined as

$$\mathbf{v}_1 = \begin{Bmatrix} 1 \\ 0 \\ -1 \end{Bmatrix}$$

Similarly, the eigenvectors associated with the other two eigenvalues ($\lambda_2 = 1, \lambda_3 = 2$) will be obtained as

$$\mathbf{v}_2 = \begin{Bmatrix} 0 \\ 1 \\ 0 \end{Bmatrix}, \quad \mathbf{v}_3 = \begin{Bmatrix} 1 \\ 0 \\ 1 \end{Bmatrix}$$

1.3.2 Similarity Transformation

Consider a matrix $\mathbf{A}_{n \times n}$ and a non-singular matrix $\mathbf{S}_{n \times n}$ and suppose

$$\mathbf{S}^{-1}\mathbf{A}\mathbf{S} = \mathbf{B} \tag{1.13}$$

We say \mathbf{B} has been obtained through a similarity transformation of \mathbf{A} , and that matrices \mathbf{A} and \mathbf{B} are similar. Similar matrices have the same set of eigenvalues. That is, eigenvalues are preserved under a similarity transformation.

1.3.3 Matrix Diagonalization

Suppose matrix $\mathbf{A}_{n \times n}$ has eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$ and linearly independent eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$. Then, the modal matrix $\mathbf{P} = [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \dots \quad \mathbf{v}_n]_{n \times n}$ diagonalizes \mathbf{A} by means of a similarity transformation:

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{P} = \mathbf{D} = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \dots & \\ & & & \lambda_n \end{bmatrix} \tag{1.14}$$

EXAMPLE 1.8: MATRIX DIAGONALIZATION

Consider the matrix in Example 1.7. The modal matrix is formed as

$$\mathbf{P} = [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \mathbf{v}_3] = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

Subsequently,

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{P} = \begin{bmatrix} \boxed{0} & 0 & 0 \\ 0 & \boxed{1} & 0 \\ 0 & 0 & \boxed{2} \end{bmatrix} = \begin{bmatrix} \boxed{\lambda_1} & & \\ & \boxed{\lambda_2} & \\ & & \boxed{\lambda_3} \end{bmatrix}$$

1.3.4 Eigenvalue Properties of Matrices

- The determinant of a matrix is the product of its eigenvalues.
- Eigenvalues of lower triangular, upper triangular, and diagonal matrices are the diagonal entries of the matrix.
- Similar matrices have the same set of eigenvalues.
- Eigenvalues of a symmetric matrix are all real.
- Every eigenvalue of an orthogonal matrix ($\mathbf{A}^{-1} = \mathbf{A}^T$) has an absolute value of 1.

Part 2: Introduction to Numerical Methods

1.4 Errors and Approximations

Numerical methods are procedures that allow for efficient solution of a mathematically formulated problem in a finite number of steps to within an arbitrary precision. Although scientific calculators can handle simple problems, computers are needed in most cases. Numerical methods commonly consist of a set of guidelines to perform predetermined mathematical (algebraic and logical) operations leading to an approximate solution of a specific problem. Such set of guidelines is known as an algorithm.

1.4.1 Sources of Computational Error

While investigating the accuracy of the results of a certain numerical method, two key questions arise: (1) what are the possible sources of error, and (2) to what degree do these errors affect the ultimate result? In numerical computations, there exist three possible sources of error:

1. Error in the initial model
2. Truncation error
3. Round-off error

The first source occurs in the initial model of the problem. These include, for example, when simplifying assumptions are made in the derivation of a physical system model, or using approximate values such as 2.7183 and 3.1416 for mathematical numbers such as e and π , respectively, and 9.81 (or 32.2) for g , the gravitational acceleration.

The second source is due to truncation, which occurs whenever an expression is approximated by some type of a mathematical method. As an example, suppose we use the Maclaurin series representation of the sine function

$$\sin \alpha = \sum_{n=odd}^{\infty} \frac{(-1)^{(n-1)/2}}{n!} \alpha^n = \alpha - \frac{1}{3!} \alpha^3 + \frac{1}{5!} \alpha^5 - \dots + \frac{(-1)^{(m-1)/2}}{m!} \alpha^m + E_m$$

where E_m is the tail end of the expansion, neglected in the process, and known as the truncation error.

The third type of computational error is caused by the computer during the process of translating a decimal number to a binary number. This is because unlike humans who use the decimal number system (in base 10), computers mostly use the binary number system (in base 2 or base 16). In doing so, the inputted number is first converted to base 2, arithmetic is done in base 2, and the outcome is converted back to base 10.

1.4.2 Binary and Hexadecimal Numbers

For ordinary purposes, base 10 is used to represent numbers. For example, the number 147 is expressed as

$$147 = [1 \times 10^2 + 4 \times 10^1 + 7 \times 10^0]_{10}$$

where the subscript is usually omitted when the base is 10. This is known as decimal notation. The so-called normalized decimal form of a number is

$$\pm 0.d_1 d_2 \dots d_m \times 10^p, \quad 1 \leq d_1 \leq 9, \quad 0 \leq d_2, d_3, \dots, d_m \leq 9 \quad (1.15)$$

The form in Equation 1.15 is also known as the floating-point form, to be explained shortly. On the other hand, most computers use the binary system (in base 2). For instance, the number 147 is expressed in base 2 as follows. First, we readily verify that

$$147 = [1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0]_2$$

Then, in base 2, we have

$$147 = (10010011)_2$$

We refer to a binary digit as a bit. This last expression represents a binary number. Similarly, the same number can be expressed in base 16, as

$$147 = [9 \times 16^1 + 3 \times 16^0]_{16} \quad \stackrel{\text{In base 16}}{\Rightarrow} \quad 147 = (93)_{16}$$

This last expression represents a hexadecimal number. While the binary system consists of only two digits, 0 and 1, there are 16 digits in the hexadecimal system; 0, 1, 2, ..., 9, A, B, ..., F, where A–F represent 10–15. We then sense that the hexadecimal system is a natural extension of the binary system. Since $2^4 = 16$, for every group of four bits, there is one hexadecimal digit. Examples include $C = (1100)_2$, $3 = (0011)_2$, and so on.

1.4.3 Floating Point and Rounding Errors

Because only a limited number of digits can be stored in computer memory, a number must be represented in a manner that uses a somewhat fixed number of digits. Digital computers mostly represent a number in one of two ways: fixed point and floating point. In a fixed-point setting, a fixed number of decimal places are used for the representation of numbers. For instance, in a system using 4 decimal places, we encounter numbers like

-2.0000, 131.0174, 0.1234. On the other hand, in a floating-point setting, a fixed number of significant digits* are used for representation. For instance, if four significant digits are used, then we will encounter numbers such as†

$$0.2501 \times 10^{-2}, \quad -0.7012 \times 10^5$$

Note that these two numbers fit the general form given by Equation 1.15. In the floating-point representation of a number, one position is used to identify its sign, a prescribed number of bits to represent its fractional part, known as the mantissa, and another prescribed number of bits for its exponential part, known as the characteristic. Computers that use 32 bits for single-precision representation of numbers, use 1 bit for the sign, 24 bits for the mantissa, and 8 bits for the exponent. Typical computers can handle wide ranges of exponents. As one example, the IEEE‡ floating-point standard range is between -38 and +38. Outside of this range, the result is an underflow if the number is smaller than the minimum and an overflow if the number is larger than the maximum.

1.4.4 Round-Off: Chopping and Rounding

Consider a positive real number N expressed as

$$N = 0.d_1d_2 \dots d_md_{m+1} \dots \times 10^p$$

The floating-point form of N , denoted by $FL(N)$, in the form of Equation 1.15, is obtained by terminating its fractional part at m decimal digits. There are two ways to do this. The first method is called chopping, and involves chopping off the digits to the right of d_m to get

$$FL(N) = 0.d_1d_2 \dots d_m \times 10^p$$

The second method is known as rounding, and involves adding $5 \times 10^{p-(m+1)}$ to N and then chopping. In this process, if $d_{m+1} < 5$, then all that happens is that the first m digits are retained. This is known as rounding down. If $d_{m+1} \geq 5$, then $FL(N)$ is obtained by adding one to d_m . This is called rounding up. It is clear that when a number is replaced with its floating-point form, whether through rounding down or up, an error results. This error is called round-off error.

EXAMPLE 1.9: CHOPPING AND ROUNDING

Consider $e = 2.71828182 \dots = 0.271828182 \dots \times 10^1$. If we use 5-digit chopping ($m = 5$), the floating-point form is $FL(e) = 0.27182 \times 10^1 = 2.7182$. We next use rounding. Since the digit immediately to the right of d_5 is $d_6 = 8 > 5$, we add 1 to d_5 to obtain

$$FL(e) = 0.27183 \times 10^1 = 2.7183$$

* Note that significant digits are concerned with the first nonzero digit and the ones to its right. For example, 4.0127 and 0.088659 both have five significant digits.

† Also expressed in scientific notation, as $0.2501E - 2$ and $-0.7012E + 5$.

‡ Institute of Electrical and Electronics Engineers.

so that we have rounded up. The same result is obtained by following the strategy of adding $5 \times 10^{p-(m+1)}$ to e and chopping. Note that $p = 1$ and $m = 5$, so that $5 \times 10^{p-(m+1)} = 5 \times 10^{-5} = 0.00005$. Adding this to e , we have

$$e + 0.00005 = 2.71828182 \dots + 0.00005 = 2.71833 \dots = 0.271833 \dots \times 10^1$$

Five-digit chopping yields $FL(e) = 0.27183 \times 10^1 = 2.7183$, which agrees with the result of rounding up.

1.4.5 Absolute and Relative Errors

In the beginning of this section, we discussed the three possible sources of error in computations. Regardless of what the source may be, computations generally yield approximations as their output. This output may be an approximation to a true solution of an equation, or an approximation of a true value of some quantity. Errors are commonly measured in one of two ways: absolute error and relative error. If \tilde{x} is an approximation to a quantity whose true value is x , the absolute error is defined as

$$e_{\text{abs}} = x - \tilde{x} \tag{1.16}$$

On the other hand, the true relative error is given by

$$e_{\text{rel}} = \frac{\text{Absolute error}}{\text{True value}} = \frac{e_{\text{abs}}}{x} = \frac{x - \tilde{x}}{x}, \quad x \neq 0 \tag{1.17}$$

Note that if the true value happens to be zero, the relative error is regarded as undefined. The relative error is generally of more significance than the absolute error, as we will discuss in Example 1.10. And because of that, whenever possible, we will present bounds for the relative error in computations.

EXAMPLE 1.10: ABSOLUTE AND RELATIVE ERRORS

Consider two different computations. In the first one, an estimate $\tilde{x}_1 = 0.003$ is obtained for the true value $x_1 = 0.004$. In the second one, $\tilde{x}_2 = 1238$ for $x_2 = 1258$. Therefore, the absolute errors are

$$(e_{\text{abs}})_1 = x_1 - \tilde{x}_1 = 0.001, \quad (e_{\text{abs}})_2 = x_2 - \tilde{x}_2 = 20$$

The corresponding relative errors are

$$(e_{\text{rel}})_1 = \frac{(e_{\text{abs}})_1}{x_1} = \frac{0.001}{0.004} = 0.25, \quad (e_{\text{rel}})_2 = \frac{(e_{\text{abs}})_2}{x_2} = \frac{20}{1258} = 0.0159$$

We notice that the absolute errors of 0.001 and 20 can be rather misleading, judging by their magnitudes. In other words, the fact that 0.001 is much smaller than 20 does not make the first error a smaller error relative to its corresponding computation. In fact, looking at the relative errors, we see that 0.001 is associated with a 25% error, while 20 corresponds to 1.59% error, much smaller than the first. Because they convey a more specific type of information, relative errors are considered more significant than absolute errors.

1.4.6 Error Bound

It is customary to use the absolute value of e_{abs} so that only the upper bound needs to be obtained, since the lower bound is clearly zero. We say that α is an upper bound for the absolute error if

$$|e_{\text{abs}}| = |x - \tilde{x}| \leq \alpha$$

Note that α does not provide an estimate for $|x - \tilde{x}|$, and is simply a bound. Similarly, we say that β is an upper bound for the relative error if

$$|e_{\text{rel}}| = \frac{|x - \tilde{x}|}{|x|} \leq \beta, \quad x \neq 0$$

EXAMPLE 1.11: ERROR BOUND

Find two upper bounds for the relative errors caused by the 5-digit chopping and rounding of e in Example 1.9.

Solution

Using the results of Example 1.9, we have

$$|e_{\text{rel}}|_{\text{Chopping}} = \frac{|e - \text{FL}(e)|}{|e|} = \frac{0.000008182 \dots \times 10^1}{0.271828182 \dots \times 10^1} = \frac{0.8182 \dots}{0.271828182 \dots} \times 10^{-5} \leq 10^{-4}$$

Here, we have used the fact that the numerator is less than 1, while the denominator is greater than 0.1. It can be shown that in the general case, an m -digit chopping results in an upper bound relative error of 10^{1-m} . For the 5-digit rounding, we have

$$|e_{\text{rel}}|_{\text{Rounding}} = \frac{|e - \text{FL}(e)|}{|e|} = \frac{0.000001818 \dots \times 10^1}{0.271828182 \dots \times 10^1} = \frac{0.1818 \dots}{0.271828182 \dots} \times 10^{-5} \leq 0.5 \times 10^{-4}$$

where we used the fact that the numerator is less than 0.5 and the denominator is greater than 0.1. In general, an m -digit rounding corresponds to an upper bound relative error of $0.5 \times 10^{1-m}$.

1.4.7 Transmission of Error from a Source to the Final Result

Now that we have learned about the sources of error, we need to find out about the degree to which these errors affect the outcome of a computation. Depending on whether addition (and/or subtraction) or multiplication (and/or division) is considered, definite conclusions may be drawn.

Theorem 1.1: Transmission of Error

Suppose in a certain computation the approximate values \tilde{x}_1 and \tilde{x}_2 have been generated for true values x_1 and x_2 , respectively, with absolute and relative errors $(e_{\text{abs}})_i$ and $(e_{\text{rel}})_i$, $i = 1, 2$, and

$$|(e_{\text{abs}})_1| \leq \alpha_1, \quad |(e_{\text{abs}})_2| \leq \alpha_2, \quad |(e_{\text{rel}})_1| \leq \beta_1, \quad |(e_{\text{rel}})_2| \leq \beta_2$$

1. The upper bound for the absolute error e_{abs} in addition and subtraction is the sum of the upper bounds of the absolute errors associated with the quantities involved. That is,

$$|e_{\text{abs}}| = |(x_1 \pm x_2) - (\tilde{x}_1 \pm \tilde{x}_2)| \leq \alpha_1 + \alpha_2$$

2. The upper bound for the relative error e_{rel} in multiplication and division is approximately equal to the sum of the upper bounds of the relative errors associated with the quantities involved. That is,

$$\text{Multiplication} \quad |e_{\text{rel}}| = \left| \frac{x_1 x_2 - \tilde{x}_1 \tilde{x}_2}{x_1 x_2} \right| \leq \beta_1 + \beta_2 \quad (1.18)$$

$$\text{Division} \quad |e_{\text{rel}}| = \left| \frac{x_1/x_2 - \tilde{x}_1/\tilde{x}_2}{x_1/x_2} \right| \leq \beta_1 + \beta_2 \quad (1.19)$$

Proof

1. We have

$$|e_{\text{abs}}| = |(x_1 \pm x_2) - (\tilde{x}_1 \pm \tilde{x}_2)| = |(x_1 - \tilde{x}_1) \pm (x_2 - \tilde{x}_2)| \leq |x_1 - \tilde{x}_1| + |x_2 - \tilde{x}_2| \leq \alpha_1 + \alpha_2$$

2. We will prove Equation 1.18. Noting that $(e_{\text{abs}})_i = x_i - \tilde{x}_i$ for $i = 1, 2$, we have $\tilde{x}_i = x_i - (e_{\text{abs}})_i$. Insertion into the left side of Equation 1.18 yields

$$|e_{\text{rel}}| = \left| \frac{x_1 x_2 - \tilde{x}_1 \tilde{x}_2}{x_1 x_2} \right| = \left| \frac{x_1 x_2 - [x_1 - (e_{\text{abs}})_1][x_2 - (e_{\text{abs}})_2]}{x_1 x_2} \right| = \left| \frac{-(e_{\text{abs}})_1 (e_{\text{abs}})_2 + (e_{\text{abs}})_2 x_1 + (e_{\text{abs}})_1 x_2}{x_1 x_2} \right|$$

But $(e_{\text{abs}})_1 (e_{\text{abs}})_2$ can be assumed negligible relative to the other two terms in the numerator. As a result,

$$|e_{\text{rel}}| \cong \left| \frac{(e_{\text{abs}})_2 x_1 + (e_{\text{abs}})_1 x_2}{x_1 x_2} \right| = \left| \frac{(e_{\text{abs}})_1}{x_1} + \frac{(e_{\text{abs}})_2}{x_2} \right| \leq \left| \frac{(e_{\text{abs}})_1}{x_1} \right| + \left| \frac{(e_{\text{abs}})_2}{x_2} \right| \leq \beta_1 + \beta_2$$

as asserted. ■

1.4.8 Subtraction of Nearly Equal Numbers

There are two particular instances leading to unacceptable inaccuracies: division by a number that is very small in magnitude, and subtraction of nearly equal numbers. Naturally, if this type of subtraction takes place in the denominator of a fraction, the latter gives rise

to the former. Consider two numbers N_1 and N_2 having the same first k decimal digits in their floating-point forms, that is,

$$\text{FL}(N_1) = 0.d_1d_2 \dots d_k a_{k+1} \dots a_m \times 10^p$$

$$\text{FL}(N_2) = 0.d_1d_2 \dots d_k b_{k+1} \dots b_m \times 10^p$$

The larger the value of k , the more “nearly equal” the two numbers are considered to be. Subtraction yields

$$\text{FL}(\text{FL}(N_1) - \text{FL}(N_2)) = 0.c_{k+1} \dots c_m \times 10^{p-k}$$

where c_{k+1}, \dots, c_m are constant digits. From this expression we see that there are only $m-k$ significant digits in the representation of the difference. In comparison with the m significant digits available in the original representations of the two numbers, some significant digits have been lost in the process. This is precisely what contributes to the round-off error, which will then be propagated throughout the subsequent computations. This can often be remedied by a simple reformulation of the problem, as illustrated in the following example.

EXAMPLE 1.12: THE QUADRATIC FORMULA WHEN $b^2 \gg 4ac$

Consider $x^2 + 52x + 3 = 0$ with approximate roots $x_1 = -0.05775645785$, $x_2 = -51.94224354$. Recall that the quadratic formula generally provides the solution of $ax^2 + bx + c = 0$, as

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

But in our example, $b^2 \gg 4ac$ so that $\sqrt{b^2 - 4ac} \cong b$. This means that in the calculation of x_1 we are subtracting nearly equal numbers in the numerator. Now, let us use a 4-digit rounding for floating-point representation. Then,

$$\text{FL}(x_1) = \frac{-52.00 + \sqrt{(52.00)^2 - 4(1.000)(3.000)}}{2(1.000)} = \frac{-52.00 + 51.88}{2.000} = -0.0600$$

and

$$\text{FL}(x_2) = \frac{-52.00 - \sqrt{(52.00)^2 - 4(1.000)(3.000)}}{2(1.000)} = -51.94$$

The corresponding relative errors, in magnitude, are computed as

$$|e_{\text{rel}}|_{x_1} = \frac{|x_1 - \text{FL}(x_1)|}{|x_1|} \cong 0.0388 \quad \text{or} \quad 3.88\%$$

$$|e_{\text{rel}}|_{x_2} = \frac{|x_2 - \text{FL}(x_2)|}{|x_2|} \cong 0.0043 \quad \text{or} \quad 0.43\%$$

Thus, the error associated with x_1 is rather large compared to that for x_2 . We anticipated this because in the calculation of x_2 nearly equal numbers are added, causing no

concern. As mentioned above, reformulation of the problem often remedies the situation. Also note that the roots of $ax^2 + bx + c = 0$ satisfy $x_1x_2 = c/a$. We will retain the value of $FL(x_2)$ and calculate $FL(x_1)$ via

$$FL(x_1) = \frac{c}{aFL(x_2)} = \frac{3.000}{(1.000)(-51.94)} = -0.05775$$

The resulting relative error is

$$|e_{rel}|_{x_1} = \frac{|x_1 - FL(x_1)|}{|x_1|} \cong 0.00011 \quad \text{or} \quad 0.011\%$$

which shows a dramatic improvement compared to the result of the first trial.

1.5 Iterative Methods

Numerical methods generally consist of a set of directions to perform predetermined algebraic and logical mathematical operations leading to an approximate solution of a specific problem. These sets of directions are known as algorithms. In order to effectively describe a certain algorithm, we will use a code. Based on the programming language or the software package used, a code can easily be modified to accomplish the task at hand. A code consists of a set of inputs, the required operations, and a list of outputs. It is standard practice to use two types of punctuation symbols in an algorithm: the period (.) proclaims that the current step is terminated, and the semicolon (;) indicates that the step is still in progress. An algorithm is stable if a small change in the initial data will correspond to a small change in the final result. Otherwise, it is unstable.

An iterative method is a process that starts with an initial guess and computes successive approximations of the solution of a problem until a reasonably accurate approximation is obtained. As we will demonstrate throughout the book, iterative methods are used to find roots of algebraic equations, solutions of systems of algebraic equations, solutions of differential equations, and much more. An important issue in an iterative scheme is the manner in which it is terminated. There are two ways to stop a procedure: (1) when a *terminating condition* is satisfied, or (2) when the maximum number of iterations is exceeded. In principle, the terminating condition should check to see whether an approximation calculated in a step is within a prescribed tolerance of the true value. In practice, however, the true value is not available. As a result, one practical form of a terminating condition is whether the difference between two successively generated quantities by the iterative method is within a prescribed tolerance. The ideal scenario is when an algorithm meets the terminating condition, and at a reasonably fast rate. If it does not, then the total number of iterations performed should not exceed a prescribed maximum number of iterations.

EXAMPLE 1.13: AN ALGORITHM AND ITS CODE

Approximate e^{-2} to seven significant digits with a tolerance of $\varepsilon = 10^{-6}$.

Solution

Retaining the first $n + 1$ terms in the Maclaurin series of $f(x) = e^x$ yields the n th-degree Taylor's polynomial

$$T_n(x) = \sum_{i=0}^n \frac{1}{i!} x^i \quad (1.20)$$

We want to evaluate e^{-2} by determining the least value of n in Equation 1.20 such that

$$\left| e^{-2} - T_n(-2) \right| < \varepsilon \quad (1.21)$$

Equation 1.21 is the terminating condition. To seven significant digits, the true value is $e^{-2} = 0.1353353$. Let us set the maximum number of iterations as $N = 20$, so that the program is likely to fail if the number of iterations exceeds 20 and the terminating condition is not met. As soon as an approximate value within the given tolerance is reached, the terminating condition is satisfied and the program is terminated. Then the outputs are n and the corresponding value for e^{-2} . We write the algorithm listed in [Table 1.2](#). It turns out that 14 iterations are needed before the terminating condition is satisfied, that is, $n = 13$. The approximate value for e^{-2} is 0.1353351 with an absolute error of $0.2 \times 10^{-6} < \varepsilon$.

1.5.1 Fundamental Iterative Method

A fundamental iterative method is the one that uses repeated substitutions. Suppose that a function $g(x)$ and a starting value x_0 are known. Let us generate a sequence of values x_1, x_2, \dots via an iteration defined by

$$x_{n+1} = g(x_n), \quad n = 0, 1, 2, \dots, \quad x_0 \text{ is known} \quad (1.22)$$

There are a few possible scenarios. The iteration may exhibit convergence, either at a fast rate or a slow rate. It is also possible that the iteration does not converge at all. Again, its

TABLE 1.2

Algorithm in Example 1.13

Input	$x = 2, \varepsilon = 10^{-6}, N = 20$
Output	An approximate value of e^{-2} accurate to within ε , or a message of "failure"
Step 1	Set $n = 0$ Tval = e^{-x} True value Term = 1 Psum = 0 Initiate partial sum Sgn = 1 Initiate alternating signs
Step 2	While $n \leq N$, do Step 3–Step 5
Step 3	Psum = Psum + Sgn*Term/ $n!$
Step 4	If $ \text{Psum} - \text{Tval} < \varepsilon$, then Output(n) Terminating condition Stop
Step 5	$n = n + 1$ Update n Sgn = -Sgn Alternate sign Term = Term*x Update Term
Step 6	Output(failure) Stop
End	

divergence may happen at a slow or a fast rate. These all depend on critical factors such as the nature of the function $g(x)$ and the starting value, x_0 . We will analyze these in detail in Chapter 3.

EXAMPLE 1.14: ITERATION BY REPEATED SUBSTITUTIONS

Consider the sequence described by $x_n = (\frac{1}{3})^n, n = 0, 1, 2, \dots$. In order to generate the same sequence of elements using iteration by repeated substitutions, we need to reformulate it to agree with Equation 1.22. To that end, we propose

$$x_{n+1} = (\frac{1}{3})x_n, \quad n = 0, 1, 2, \dots, \quad x_0 = 1$$

This way, the sequence starts with $x_0 = 1$, which matches the first element of the original sequence. Next, we calculate

$$x_1 = \frac{1}{3}x_0 = \frac{1}{3}, \quad x_2 = \frac{1}{3}x_1 = \frac{1}{9}, \quad x_3 = \frac{1}{3}x_2 = \frac{1}{27}, \dots$$

which agree with the respective elements in the original sequence. Therefore,

$$x_{n+1} = g(x_n), \quad n = 0, 1, 2, \dots, \quad x_0 = 1$$

where $g(x) = \frac{1}{3}x$.

1.5.2 Rate of Convergence of an Iterative Method

Consider a sequence $\{x_n\}$ that converges to x . The error at the n th iteration is then defined as

$$e_n = x - x_n, \quad n = 0, 1, 2, \dots$$

If there exists a number R and a constant $K \neq 0$ such that

$$\lim_{n \rightarrow \infty} \frac{|e_{n+1}|}{|e_n|^R} = K \tag{1.23}$$

then we say that R is the rate of convergence of the sequence. There are two types of convergence that we encounter more often than others: linear and quadratic. A convergence is linear if $R = 1$, that is,

$$\lim_{n \rightarrow \infty} \frac{|e_{n+1}|}{|e_n|} = K \neq 0 \tag{1.24}$$

A convergence is said to be quadratic if $R = 2$, that is,

$$\lim_{n \rightarrow \infty} \frac{|e_{n+1}|}{|e_n|^2} = K \neq 0 \tag{1.25}$$

Rate of convergence is not always an integer. We will see in Section 3.6, for instance, that the secant method has a rate of $\frac{1}{2}(1 + \sqrt{5}) \cong 1.618$.

EXAMPLE 1.15: RATE OF CONVERGENCE

Determine the rate of convergence for the sequence in Example 1.14.

Solution

Since $x_n = \left(\frac{1}{3}\right)^n \rightarrow 0$ as $n \rightarrow \infty$, the limit is $x = 0$. With that, the error at the n th iteration is

$$e_n = x - x_n = 0 - \left(\frac{1}{3}\right)^n = -\left(\frac{1}{3}\right)^n$$

We will first examine $R = 1$, that is, Equation 1.24:

$$\lim_{n \rightarrow \infty} \frac{|e_{n+1}|}{|e_n|} = \lim_{n \rightarrow \infty} \frac{\left|-\left(\frac{1}{3}\right)^{n+1}\right|}{\left|-\left(\frac{1}{3}\right)^n\right|} = \frac{1}{3} \neq 0$$

Therefore, $R = 1$ works and convergence is linear. Once a value of R satisfies the condition in Equation 1.23, no other values need be inspected.

PROBLEM SET (CHAPTER 1)

Differential Equations (Section 1.1)

 In Problems 1 through 8, solve each IVP.

1. $\dot{y} + \frac{1}{3}y = t$, $y(0) = -1$
2. $\dot{y} + ty = t$, $y(0) = \frac{1}{2}$
3. $\frac{1}{2}\dot{y} + y = 0$, $y(1) = 1$
4. $\dot{y} + y = e^{-2t}$, $y(0) = 1$
5. $\ddot{y} + 2\dot{y} + 2y = 0$, $y(0) = 0$, $\dot{y}(0) = 1$
6. $\ddot{y} + 2\dot{y} + y = e^{-t}$, $y(0) = 1$, $\dot{y}(0) = 0$
7. $\ddot{y} + 2\dot{y} = \sin t$, $y(0) = 0$, $\dot{y}(0) = 1$
8. $\ddot{y} + 2\dot{y} = t$, $y(0) = 1$, $\dot{y}(0) = 1$

Matrix Analysis (Section 1.2)

 In Problems 9 through 12, calculate the determinant of the given matrix.

9. $\mathbf{A} = \begin{bmatrix} 1 & 5 & 1 \\ 3 & 0 & 2 \\ -4 & 2 & 6 \end{bmatrix}$

10. $\mathbf{A} = \begin{bmatrix} 8 & 2 & -1 \\ 1 & 0 & 4 \\ -3 & 4 & 5 \end{bmatrix}$

$$11. \mathbf{A} = \begin{bmatrix} -1 & 3 & 1 & 2 \\ 1 & -1 & 4 & 3 \\ 1 & 0 & 1 & 0 \\ 2 & 3 & 4 & -5 \end{bmatrix}$$

$$12. \mathbf{A} = \begin{bmatrix} 0 & -6 & 1 & 0 \\ -1 & 2 & 4 & -3 \\ 2 & 0 & 1 & 1 \\ 4 & 5 & -3 & 1 \end{bmatrix}$$


 In Problems 13 through 16, solve each system using Cramer's rule.

$$13. \mathbf{Ax} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} 2 & 1 & 0 & -1 \\ 1 & 3 & -1 & 3 \\ 0 & 1 & -3 & 2 \\ 2 & 0 & 1 & 4 \end{bmatrix}, \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} -3 \\ 13 \\ 5 \\ 11 \end{Bmatrix}$$

$$14. \mathbf{Ax} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} -1 & 0 & 4 & 2 \\ 5 & 1 & 3 & -1 \\ 1 & 0 & 2 & 2 \\ -3 & 2 & 0 & -2 \end{bmatrix}, \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} 3 \\ 22 \\ 5 \\ -7 \end{Bmatrix}$$

$$15. \begin{cases} x_1 + x_2 - 4x_3 = -1 \\ -2x_1 + x_2 + 3x_3 = 0 \\ x_2 + 5x_3 = 6 \end{cases}$$

$$16. \begin{cases} 4x_1 + x_2 - 3x_3 = -13 \\ -x_1 + 2x_2 + 6x_3 = 13 \\ x_1 + 7x_3 = 4 \end{cases}$$

 In Problems 17 through 20, find the inverse of each matrix.

$$17. \mathbf{A} = \begin{bmatrix} 4 & 0 & 1 \\ 0 & 3 & -2 \\ -1 & -2 & -1 \end{bmatrix}$$

$$18. \mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

$$19. \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 4 & 3 & -2 \end{bmatrix}$$

$$20. \mathbf{A} = \begin{bmatrix} \alpha & 0 & -1 \\ 0 & \alpha+1 & 2 \\ 1 & 0 & \alpha+2 \end{bmatrix}, \quad \alpha = \text{parameter}$$

Matrix Eigenvalue Problem (Section 1.3)

 In Problems 21 through 24, find the eigenvalues and the corresponding eigenvectors of each matrix.


$$21. \mathbf{A} = \begin{bmatrix} -3 & 0 \\ -2 & 1 \end{bmatrix}$$

$$22. \mathbf{A} = \begin{bmatrix} 2 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$23. \mathbf{A} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & -3 \\ 0 & 0 & -1 \end{bmatrix}$$

$$24. \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 2 & 3 & 3 \end{bmatrix}$$

25.  Prove that a singular matrix has at least one zero eigenvalue.

 In Problems 26 through 28, diagonalize each matrix by using an appropriate modal matrix.

$$26. \mathbf{A} = \begin{bmatrix} -2 & -1 & -1 \\ 3 & 2 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

$$27. \mathbf{A} = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

$$28. \mathbf{A} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & -3 \\ 0 & 0 & -1 \end{bmatrix}$$

Errors and Approximations (Section 1.4)

 In Problems 29 through 32, convert each decimal number to a binary number.

29. 67

30. 234

31. 45.25

32. 1127

✎ In Problems 33 through 36, convert each decimal number to a hexadecimal number.

33. 596

34. 1327

35. 23.1875

36. 364.5

✎ In Problems 37 through 40, convert each hexadecimal number to a binary number.

37. $(2B5.4)_{16}$

38. $(143)_{16}$

39. $(3D.2)_{16}$

40. $(12F.11)_{16}$

✎ In Problems 41 through 45, write the floating-point form of each decimal number by m -digit rounding for the given value of m .

41. -0.00031676 ($m = 4$)

42. 11.893 ($m = 4$)

43. 200.346 ($m = 5$)

44. -1203.423 ($m = 6$)

45. 22318 ($m = 4$)

46. ✎ Suppose m -digit chopping is used to find the floating-point form of

$$N = 0.d_1d_2 \dots d_md_{m+1} \dots \times 10^p$$

show that

$$|e_{\text{rel}}|_{\text{Chopping}} = \frac{|N - \text{FL}(N)|}{|N|} \leq 10^{1-m}$$

47. ✎ Suppose in Problem 46 we use m -digit rounding. Show that

$$|e_{\text{rel}}|_{\text{Rounding}} = \frac{|N - \text{FL}(N)|}{|N|} \leq 0.5 \times 10^{1-m}$$

Iterative Methods (Section 1.5)

48. ✎ Consider the sequence described by $x_n = \frac{1}{1+n}$, $n = 0, 1, 2, \dots$

- Find a suitable function $g(x)$ so that the sequence can be generated by means of repeated substitution in the form $x_{n+1} = g(x_n)$, $n = 0, 1, 2, \dots$
- Determine the rate of convergence of the sequence to its limit.

2

Introduction to MATLAB

This chapter presents features and capabilities of MATLAB pertinent to numerical methods. These range from vector and matrix operations and symbolic calculations to plotting options for functions and sets of data. Several MATLAB built-in functions and their applications will also be introduced. The chapter concludes with guidelines to prepare user-defined functions and script files to perform specific tasks.

2.1 MATLAB Built-In Functions

MATLAB has a large number of built-in elementary functions, each accompanied by a brief but sufficient description through the help command. For example,

```
>> help sqrt
sqrt    Square root.
        sqrt(X) is the square root of the elements of X. Complex
        results are produced if X is not positive.

        See also sqrtm, realsqrt, hypot.

        Reference page in Help browser
        doc sqrt

>> (1+sqrt(5))/2      % Calculate the golden ratio

ans =

    1.6180
```

The outcome of a calculation can be stored under a variable name, and suppressed by using a semicolon at the end of the statement:

```
>> g_ratio = (1+sqrt(5))/2;
```

If the variable is denoted by “ a ”, other elementary functions include $\text{abs}(a)$ for $|a|$, $\text{sin}(a)$ for $\sin(a)$, $\text{log}(a)$ for $\ln a$, $\text{log}_{10}(a)$ for $\log_{10}(a)$, $\text{exp}(a)$ for e^a , and many more. Descriptions of these functions are available through the help command.

2.1.1 Rounding Commands

MATLAB has four built-in functions that round decimal numbers to the nearest integer via different rounding techniques. These are listed in [Table 2.1](#).

TABLE 2.1

MATLAB Rounding Functions

MATLAB Function	Example
round(a) Round to the nearest integer	round(1.65) = 2, round(-4.7) = -5
fix(a) Round toward zero	fix(1.65) = 1, fix(-4.7) = -4
ceil(a) Round up toward infinity	ceil(1.65) = 2, ceil(-4.7) = -4
floor(a) Round down toward minus infinity	floor(1.65) = 1, floor(-4.7) = -5

2.1.2 Relational Operators

Table 2.2 gives a list of the relational and logical operators used in MATLAB.

2.1.3 Format Options

The `format` built-in function offers several options for displaying output. The preferred option can be chosen by selecting the following in the pull-down menu: File → Preferences → Command Window. A few of the format options are listed in Table 2.3.

TABLE 2.2

MATLAB Relational Operators

Mathematical Symbol	MATLAB Symbol
=	==
≠	~=
<	<
>	>
≤	<=
≥	>=
AND	& or &&
OR	or
NOT	~

TABLE 2.3

MATLAB Format Options

Format Option	Description	Example: 73/7
format short (default)	Fixed point with 4 decimal digits	10.4286
format long	Fixed point with 14 decimal digits	10.428571428571429
format short e	Scientific notation with 4 decimal digits	1.0429e+001
format long e	Scientific notation with 14 decimal digits	1.042857142857143e+001
format bank	Fixed point with 2 decimal digits	10.43

2.2 Vectors and Matrices

Vectors can be created in several ways in MATLAB. The row vector $\mathbf{v} = [1 \ 4 \ 6 \ 7 \ 10]$ is created as

```
>> v = [1 4 6 7 10]
```

```
v =  
    1     4     6     7    10
```

Commas may be used instead of spaces between elements. For column vectors, the elements must be separated by semicolons.

```
>> v = [1;4;6;7;10]
```

```
v =  
    1  
    4  
    6  
    7  
   10
```

The length of a vector is determined by using the `length` command:

```
>> length(v)
```

```
ans =  
     5
```

The size of a vector is determined by the `size` command. For the last (column) vector defined above, we find

```
>> size(v)
```

```
ans =  
     5     1
```

Arrays of numbers with equal spacing can be created more effectively. For example, a row vector whose first element is 2, its last element is 17, with a spacing of 3 is created as

```
>> v = [2:3:17]      or      >> v = 2:3:17
```

```
v =  
     2     5     8    11    14    17
```

To create a column vector with the same properties

```
>> v = [2:3:17]'
```

```
v =  
     2  
     5  
     8  
    11  
    14  
    17
```

Any component of a vector can be easily retrieved. For example, the third component of the above vector is retrieved by typing

```
>> v(3)
ans =
     8
```

A group of components may be retrieved as well. For example, the last three components of the row vector defined earlier are recovered as

```
>> v = 2:3:17;
>> v(end-2:end)
ans =
    11     14     17
```

2.2.1 Linspace

Another way to create vectors with equally spaced elements is by using the `linspace` command.

```
>> x = linspace(1,5,6)    % 6 equally-spaced points between 1 and 5
x =
    1.0000    1.8000    2.6000    3.4000    4.2000    5.0000
```

The default value for the number of points is 100. Therefore, if we use `x = linspace(1,5)`, then 100 equally spaced points will be generated between 1 and 5.

2.2.2 Matrices

A matrix can be created by using brackets enclosing all of its elements, rows separated by a semicolon.

```
>> A = [1 -2 3;-3 0 1;5 1 4]
A =
     1     -2     3
    -3     0     1
     5     1     4
```

An entry can be accessed by using the row and column number of the location of that entry.

```
>> A(3,2)    % Entry at the intersection of the 3rd row and 2nd column
ans =
     1
```

An entire row or column of a matrix is accessed by using a colon.

```
>> Row_2 = A(2,:)      % 2nd row of A

Row_2 =

    -3     0     1

>> Col_3 = A(:,3)    % 3rd column of A

Col_3 =

     3
     1
     4
```

To replace an entire column of matrix A by a given vector v , we proceed as follows:

```
>> v = [1;0;1];
>> A_new = A;        % Pre-allocate the new matrix
>> A_new(:,2) = v    % Replace the 2nd column with v

A_new =

     1     1     3
    -3     0     1
     5     1     4
```

The $m \times n$ zero matrix is created by using `zeros(m,n)`; for instance, the 3×2 zero matrix:

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

A =

     0     0
     0     0
     0     0
```

The $m \times n$ zero matrix is commonly used for pre-allocation of matrices to save memory space. In the matrix A defined above, any entry can be altered while others remain unchanged.

```
>> A(2,1) = -3; A(3,2) = -1

A =

     0     0
    -3     0
     0    -1
```

Size of a matrix is determined by using the `size` command:

```
>> size(A)

ans =

     3     2
```

The $n \times n$ identity matrix is created by `eye(n)`:

```
>> I = eye(3)
```

```
I =
```

```
    1    0    0
    0    1    0
    0    0    1
```

Matrix operations (Section 1.2) can be easily performed in MATLAB. If the sizes are not compatible, or the operations are not defined, MATLAB returns an error message to that effect.

```
>> A = [1 2;2 -2;4 0]; B = [-1 3;2 1]; % A is 3-by-2, B is 2-by-2
>> C = A*B % Operation is valid
```

```
C =
```

```
     3     5
    -6     4
    -4    12
```

2.2.3 Determinant, Transpose, and Inverse

The determinant of an $n \times n$ matrix is calculated by the `det` command.

```
>> A = [1 2 -3;0 2 1;1 2 5]; det(A)
```

```
ans =
```

```
    16
```

The transpose of a matrix is found as

```
>> At = A.'
```

```
At =
```

```
     1     0     1
     2     2     2
    -3     1     5
```

The inverse of a (non-singular) matrix is calculated by the `inv` command

```
>> Ai = inv(A)
```

```
Ai =
```

```
    0.5000   -1.0000    0.5000
    0.0625    0.5000   -0.0625
   -0.1250         0    0.1250
```

2.2.4 Slash Operators

There are two slash operators in MATLAB: backslash (\) and slash (/).

```
>> help \
```

```
\ Backslash or left matrix divide.
```

```
A\B is the matrix division of A into B, which is roughly the same as INV(A)*B , except it is computed in a different way. If A is an N-by-N matrix and B is a column vector with N components, or a matrix with several such columns, then X = A\B is the solution to the equation A*X = B. A warning message is printed if A is badly scaled or nearly singular. A\EYE(SIZE(A)) produces the inverse of A.
```

The backslash (\) operator is employed for solving a linear system of algebraic equations $\mathbf{Ax} = \mathbf{b}$, whose solution vector is obtained as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. However, instead of performing $\mathbf{x} = \text{inv}(\mathbf{A}) * \mathbf{b}$, it is most efficient to find it as follows:

```
>> A = [1 -1 2; 2 0 3; 1 -2 1]; b = [2; 8; -3];  
>> x = A\b
```

```
x =  
    1  
    3  
    2
```

The description of the slash (/) operator is given below.

```
>> help /
```

```
/ Slash or right matrix divide.
```

```
A/B is the matrix division of B into A, which is roughly the same as A*INV(B) , except it is computed in a different way. More precisely, A/B = (B'\A')'. See MLDIVIDE for details.
```

2.2.5 Element-by-Element Operations

Element-by-element operations are summarized in [Table 2.4](#). These are used when operations are performed on each element of a vector or matrix.

TABLE 2.4

Element-by-Element Operations

MATLAB Symbol	Description
.*	Multiplication
./	(right) Division
.^	Exponentiation

For example, suppose we want to raise each element of a vector to power of 2.

```
>> x = 0:2:10
```

```
x =  
    0     2     4     6     8    10
```

```
>> x.^2    % If we use x^2 instead, an error is returned by MATLAB
```

```
ans =  
    0     4    16    36    64   100
```

Now consider $(1 + x)/(2 + x)$ where vector x is as defined above. This fraction is to be evaluated for each value of x :

```
>> (1.+x)./(2.+x)
```

```
ans =  
    0.5000    0.7500    0.8333    0.8750    0.9000    0.9167
```

If two arrays are involved in the element-by-element operation, they must be of the same size.

```
>> v = [1;2;3];  
>> w = [2;3;4];  
>> v.*w
```

```
ans =  
     2  
     6  
    12
```

2.2.6 Diagonal Matrices and Diagonals of a Matrix

If \mathbf{A} is an $n \times n$ matrix, then $\text{diag}(\mathbf{A})$ creates an $n \times 1$ vector whose components are the (main) diagonal entries of \mathbf{A} . To construct a diagonal matrix whose main diagonal matches that of \mathbf{A} , we use $\text{diag}(\text{diag}(\mathbf{A}))$:

```
>> A = [-1 0 1 3;1 2 1 -4;0 2 4 1;1 0 -2 5]
```

```
A =  
   -1     0     1     3  
     1     2     1    -4  
     0     2     4     1  
     1     0    -2     5
```

```
>> diag(A)    % Returns a vector of the diagonal entries of A
```

```
ans =
    -1
     2
     4
     5

>> D = diag(diag(A))    % Constructs a diagonal matrix with diagonal entries of A

D =
    -1     0     0     0
     0     2     0     0
     0     0     4     0
     0     0     0     5
```

The command `diag(A,1)` creates a vector consisting of the entries of `A` that are one level higher than the main diagonal. Of course, the dimension of this vector is one less than the dimension of `A` itself. Then `diag(diag(A,1),1)` creates a matrix (size of `A`) whose entries one level higher than the main diagonal are the components of the vector `diag(A,1)`.

```
>> diag(A,1)

ans =
     0
     1
     1

>> diag(diag(A,1),1)

ans =
     0     0     0     0
     0     0     1     0
     0     0     0     1
     0     0     0     0
```

Similarly, `diag(A,-1)` creates a vector whose components are the entries of `A` that are one level lower than the main diagonal. Subsequently, `diag(diag(A,-1),-1)` generates a matrix (size of `A`) whose entries one level lower than the main diagonal are the components of the vector `diag(A,-1)`.

```
>> diag(A,-1)

ans =
     1
     2
    -2

>> diag(diag(A,-1),-1)

ans =
     0     0     0     0
     1     0     0     0
     0     2     0     0
     0     0    -2     0
```

Other commands such as `diag(A,2)`, `diag(A,-2)`, and so on can be used for similar purposes. The command `triu(A)` returns the upper-triangular version of `A`, that is, matrix `A` with all entries below the main diagonal set to zero.

```
>> triu(A)
```

```
ans =
```

```
-1    0    1    3
  0    2    1   -4
  0    0    4    1
  0    0    0    5
```

Similarly, `tril(A)` returns the lower-triangular version of `A`, that is, matrix `A` with all entries above the main diagonal set to zero.

```
>> tril(A)
```

```
ans =
```

```
-1    0    0    0
  1    2    0    0
  0    2    4    0
  1    0   -2    5
```

2.3 Symbolic Math Toolbox

The Symbolic Math toolbox allows for manipulation of symbols to perform operations symbolically. Symbolic variables are created by using the `syms` command. Consider, for example, the function $g = 4.81 \sin(a/3) + 3e^{-b/c}$ where $c = 2.1$. This function can be defined symbolically as follows:

```
>> syms a b
```

```
>> c = 2.1;
```

```
>> g = 4.81*sin(a/3)+3*exp(-b/c)
```

```
g =
```

```
3/exp((10*b)/21) + (481*sin(a/3))/100    % Value of c has been substituted!
```

In symbolic expressions, numbers are always converted to the ratio of two integers, as it is observed here as well. For decimal representation of numbers, we use the `vpa` (variable precision arithmetic) command. The syntax is

```
>> vpa(g,n)
```

where `n` is the number of desired digits. For example, if four digits are desired in our current example, then


```
>> g4 = vpa(4.81*sin(a/3)+3*exp(-b/c),4)
```

```
g4 =
```

```
4.81*sin(0.3333*a) + 3.0/exp(0.4762*b)
```

To evaluate the symbolic function g for specified values of a and b , we use the `subs` command which replaces all variables in the symbolic expression g with values obtained from the MATLAB workspace. For instance, to evaluate g when $a=1$ and $b=2$,

```
>> a = 1; b = 2; subs(g)
```

```
ans =
```

```
3*exp(-20/21) + (481*sin(1/3))/100
```

The command `double` may then be used to convert to double precision

```
>> double(ans)
```

```
ans =
```

```
2.7313
```

The function $g = 4.81 \sin(a/3) + 3e^{-b/c}$ in the current example may also be defined symbolically via

```
>> g = sym('4.81*sin(a/3)+3*exp(-b/c)')
```

```
g =
```

```
4.81*sin(a/3) + 3*exp(-b/c)
```

Note that a , b , and c do not need to be declared symbols, as this is handled automatically by `sym` in the definition of g . Also, assignment of a specific value (such as $c=2.1$) to a variable will not be taken into account when using `sym`. Instead, we can use the `subs` command at this stage:

```
>> c = 2.1; g = subs(g)
```

```
g =
```

```
3*exp(-(10*b)/21) + 4.81*sin(a/3)
```

This agrees with what we saw at the outset of this discussion. The symbolic function g can be evaluated for a list of specific parameter values as follows:

```
>> a = 1; b = 2; double(subs(g))
```

```
ans =
```

```
2.7313 % Agrees with previous result
```

2.3.1 Anonymous Functions

An anonymous function offers a way to create a function for simple expressions without creating an M file. Anonymous functions can only contain one expression and cannot return more than one output variable. They can either be created in the Command Window or as a script. The generic form is

```
My_function = @(arguments)(expression)
```

As an example, let us create an anonymous function (in the Command Window) to evaluate $R = \sqrt{1 + e^{-bx/2}}$ when $b = 1$ and $x = 2$.

```
>> R = @(b,x)(sqrt(1+exp(-b*x/2)));
```

This creates $R(b, x)$, which is then evaluated for specific values of b and x . For example,

```
>> R(1,2)
```

```
ans =
```

```
1.1696
```

An anonymous function can be used in another anonymous function. For example, to evaluate $L = \ln\sqrt{1 + e^{-bx/2}}$,

```
>> R = @(b,x)(sqrt(1+exp(-b*x/2)));
```

```
>> L = @(b,x)(log(R(b,x)));
```

```
>> L(1,2)
```

```
ans =
```

```
0.1566
```

2.3.2 MATLAB Function

The built-in `matlabFunction` allows us to generate a MATLAB file or anonymous function from `sym` object. The generic form $G = \text{matlabFunction}(F)$ converts the symbolic expression or function F to a MATLAB function with the handle G .

Let us consider the example involving the evaluation of $R = \sqrt{1 + e^{-bx/2}}$ when $b = 1$ and $x = 2$.

```
>> syms b x
```

```
>> R = matlabFunction(sqrt(1+exp(-b*x/2)))
```

```
R =
```

```
@(b,x)sqrt(exp(b.*x.*(-1.0./2.0))+1.0) % Inputs are arranged in alphabetical order
```

```
>> R(1,2)
```

```
ans =
```

```
1.1696
```

As expected, this agrees with the earlier result using the anonymous function. Note that if the desired order of variables is not specified by the user, MATLAB will list them in alphabetical order. In the above example, omitting the list of variables would still result in $R(b, x)$. If, however, $R(x, b)$ is desired, the 'vars' option is utilized as follows:

```
>> R = matlabFunction(sqrt(1+exp(-b*x/2)), 'vars', [x b])
```

```
R =
```

```
@(x,b)sqrt(exp(b.*x./2)+1)
```

In the previous example, where the function was defined as $R(b, x)$, suppose b is a scalar and x is a vector. Let $b = 1$ and $x = [1 \ 2 \ 3]$. Then,

```
>> b = 1; x = [1 2 3];
```

```
>> R(b,x)
```

```
ans =
```

```
1.2675 1.1696 1.1060
```

Three outputs are returned, one for each component in the vector x . Note that, since the second component of x happens to be 2, the second returned output matches what we got earlier for the case of $b = 1$ and $x = 2$.

2.3.3 Differentiation

In order to find the derivative of a function with respect to any of its variables, the function must be defined symbolically. For example, consider $f(t) = t^3 - \sin t$, a function of a single variable. To determine df/dt , we proceed as follows:

```
>> f = sym('t^3-sin(t)');
```

```
>> dfdt = diff(f)
```

```
dfdt =
```

```
3*t^2 - cos(t)
```

The second derivative d^2f/dt^2 is found as

```
>> dfdt2 = diff(f,2)
```

```
dfdt2 =
```

```
6*t + sin(t)
```

The symbolic derivatives can be converted to MATLAB functions for convenient evaluation. For example, to evaluate df/dt when $t = 1.26$,

```
>> f = sym('t^3-sin(t)');
```

```
% Define function symbolically
```

```
>> fd = matlabFunction(diff(f));
```

```
% Convert the derivative to a MATLAB function
```

```
>> fd(1.26) % Evaluate the derivative
ans =
4.4570
```

2.3.4 Partial Derivatives

The `diff` command can also handle partial differentiation. Consider $h(x, y) = 2x + y^2$, a function of two variables. The first partial derivatives of h with respect to its variables x and y are found as follows:

```
>> h = sym('2*x+y^2');
>> hx = diff(h, 'x')
hx =
2
>> hy = diff(h, 'y')
hy =
2*y
```

To find the second partial derivative with respect to y , the `diff` command is applied to the first partial:

```
>> hy2 = diff(hy, 'y')
hy2 =
2
```

2.3.5 Integration

Indefinite and definite integrals are calculated symbolically via the `int` command. For example, the indefinite integral $\int (2t + \cos 3t) dt$ is calculated as

```
>> f = sym('2*t+cos(3*t)');
>> int(f)
ans =
sin(3*t)/3 + t^2
```

The definite integral $\int_1^3 (at - e^{t/2}) dt$, where a is a parameter, is calculated as follows:

```
>> g = sym('a*t-exp(t/2)');
>> syms t
>> I = int(g,t,1,3) % t is the integration variable, and 1 and 3 are limits of integration
I =
4*a - 2*exp(1/2)*(exp(1) - 1)
```

To evaluate the integral when $a = 1$, we proceed as follows:

```
>> a = 1; double(subs(I))  
  
ans =  
  
-1.665935599275873
```

Note that the default integration variable here is t . Thus, in the above example, it could have been omitted to yield the correct result:

```
>> int(g,1,3)  
  
ans =  
  
4*a - 2*exp(1/2)*(exp(1) - 1)
```

2.4 Program Flow Control

Program flow can be controlled with the following three commands: `for`, `if`, and `while`.

2.4.1 for Loop

A `for/end` loop repeats a statement, or a group of statements, a specific number of times. Its generic form is

```
for i = first:increment:last,  
    statements  
end
```

The loop is executed as follows. The index i assumes its first value, all statements in the subsequent lines are executed with $i = \text{first}$, then the program goes back to the `for` command and i assumes the value $i = \text{first} + \text{increment}$ and the process continues until the very last run corresponding to $i = \text{last}$.

As an example, suppose we want to generate a 5×5 matrix A with diagonal entries all equal to 1, and upper diagonal entries all equal to 2, while all other entries are zero.

```
A = zeros(5,5);    % Pre-allocate  
for i = 1:5,  
    A(i,i) = 1;    % Diagonal entries  
end  
for i = 1:4,  
    A(i,i+1) = 2; % Upper diagonal entries  
end
```

Execution of this script returns

```
>> A
A =
    1    2    0    0    0
    0    1    2    0    0
    0    0    1    2    0
    0    0    0    1    2
    0    0    0    0    1
```

2.4.2 The if Command

The most general form of the if command is

```
if condition 1
    set of expressions 1
else if condition 2
    set of expressions 2
else
    set of expressions 3
end
```

The simplest form of a conditional statement is the if/end structure. For example,

```
syms x
f = matlabFunction(log(x/3)); x = 1;
if f(x) ~= 0,
error('x is not a root')
end
```

Execution of this script returns

```
x is not a root
```

The if/else/end structure allows for choosing one group of expressions from two groups. The most complete form of the conditional statement is the if/elseif/else/end structure. Let us create the same 5×5 matrix **A** as above, this time employing the if/elseif/else/end structure.

```
A = zeros(5,5);
for i = 1:5,
    for j = 1:5,
        if j == i+1,
            A(i,j) = 2;
        elseif j == i,
            A(i,j) = 1;
        end
    end
end
```

Note that each for statement is accompanied by an end statement. Execution of this script returns

```
>> A
A =
    1     2     0     0     0
    0     1     2     0     0
    0     0     1     2     0
    0     0     0     1     2
    0     0     0     0     1
```

2.4.3 while Loop

A `while`/`end` loop repeats a statement, or a group of statements, until a specific condition is met. Its generic form is

```
while condition
    statements
end
```

We will generate the same 5×5 matrix **A** as before, this time with the aid of the `while` loop.

```
A = eye(5); i = 1;
while i < 5,
    A(i,i+1) = 2;
    i = i+1;
end
```

Executing this script returns the same matrix as above.

```
>> A
A =
    1     2     0     0     0
    0     1     2     0     0
    0     0     1     2     0
    0     0     0     1     2
    0     0     0     0     1
```

2.5 Displaying Formatted Data

Formatted data can be displayed by using either `disp` or `fprintf`. An example of how the `disp` command is used is

```
>> v = [1.2 -9.7 2.8];
>> disp(v)

    1.2000    -9.7000     2.8000
```

Formatted data may be better controlled via `fprintf`. Let us consider the script below. A function $f(x) = x \cos x + 1$ is defined. For $k = 1, 2, 3, 4, 5$ we want to calculate each $c = \left(\frac{1}{2}\right)^k$ as

well as the corresponding function value $f(c)$. The output is to be displayed in tabulated form containing the values of k , c , and $f(c)$ for each $k = 1, 2, 3, 4, 5$.

```
syms x
f = matlabFunction(x*cos(x)+1);
disp(' k      c      f(c)')
for k = 1:5,
    c = (1/2)^k;
    fc = f(c);
    fprintf(' %2i      %6.4f      %6.4f\n',k,c,fc)
end
```

Execution of this script returns

k	c	f(c)
1	0.5000	1.4388
2	0.2500	1.2422
3	0.1250	1.1240
4	0.0625	1.0624
5	0.0313	1.0312

The `disp` command simply displays all contents inside the single quotes. The `fprintf` command is used inside `for` loop. For each k in the loop, `fprintf` writes the value of k , the calculated value of c , as well as $f(c)$. The format `%2i` means integer of length 2, which is being used for displaying the value of k . In `%6.4f`, the letter `f` represents the fixed-point format, 6 is the length, and the number 4 is the number of digits to the right of the decimal. Finally, `\n` means new line. A more detailed description is available through the `help` command.

2.5.1 Differential Equations

Differential equations and initial-value problems can be solved by the `dsolve` function. For example, the solution of the differential equation $y' + (x+1)y = 0$ is obtained as

```
>> y = dsolve('Dy+(x+1)*y=0','x')
```

```
y =
```

```
C4/exp((x + 1)^2/2)      % C4 is some constant
```

Note that the default independent variable in `dsolve` is t . Since in our example the independent variable is x , we needed to specify that in single quotes. The initial-value problem $\ddot{x} + 2\dot{x} + 2x = e^{-t}$, $x(0) = 0$, $\dot{x}(0) = 1$ is solved as

```
>> x = dsolve('D2x+2*Dx+2*x=exp(-t)','x(0)=0, Dx(0)=1')
```

```
x =
```

```
1/exp(t) - cos(t)/exp(t) + sin(t)/exp(t)
```


2.6 Plotting

Plotting a vector of values versus another vector of values is done by using the `plot` command. For example to plot the function $x(t) = e^{-t}(\cos t + \sin t)$ over the interval $[0, 5]$ using 100 points:

```
>> t = linspace(0,5);    % 100 values for 0   t   5
>> x = exp(-t).*(cos(t)+sin(t));    % Corresponding 100 values for x
>> plot(t,x)    % Figure 2.1
```

The Figure Window can be used to edit the figure. These include adding grid, adjusting thickness of lines and curves, adding text and legend, axes titles, and much more.

2.6.1 subplot

The built-in function `subplot` is designed to create multiple figures in tiled positions. Suppose we want to plot the function $z(x, t) = e^{-x}\sin(t + 2x)$ versus $0 \leq x \leq 5$ for four values of $t = 0, 1, 2, 3$. Let us generate the four plots and arrange them in a 2×2 formation.

```
x = linspace(0,5); t = 0:1:3;
for i = 1:4,
    for j = 1:100,
        z(j,i) = exp(-x(j))*sin(t(i)+2*x(j));    % Generate 100 values of z for each t
    end
end
```

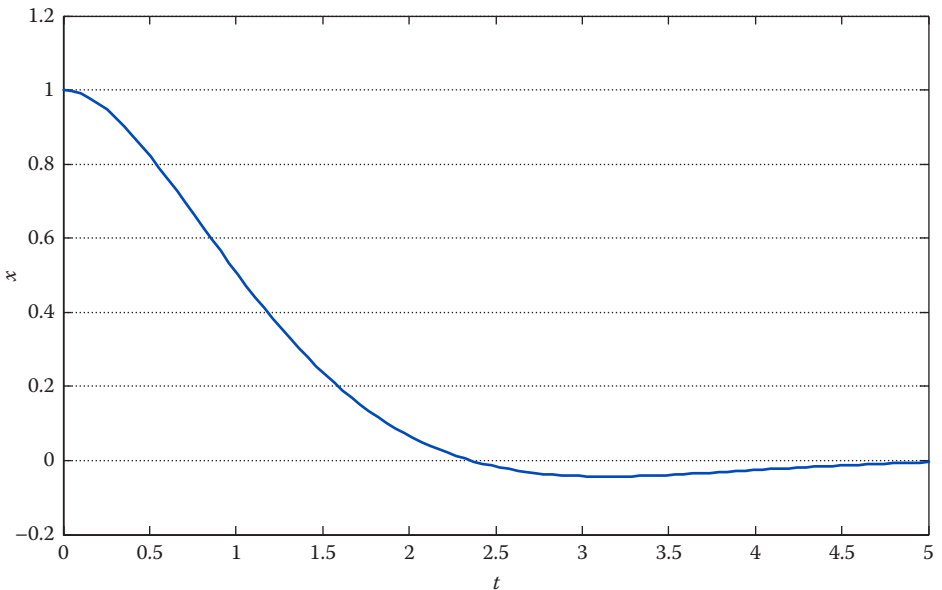


FIGURE 2.1

Plot of a function versus its variable.

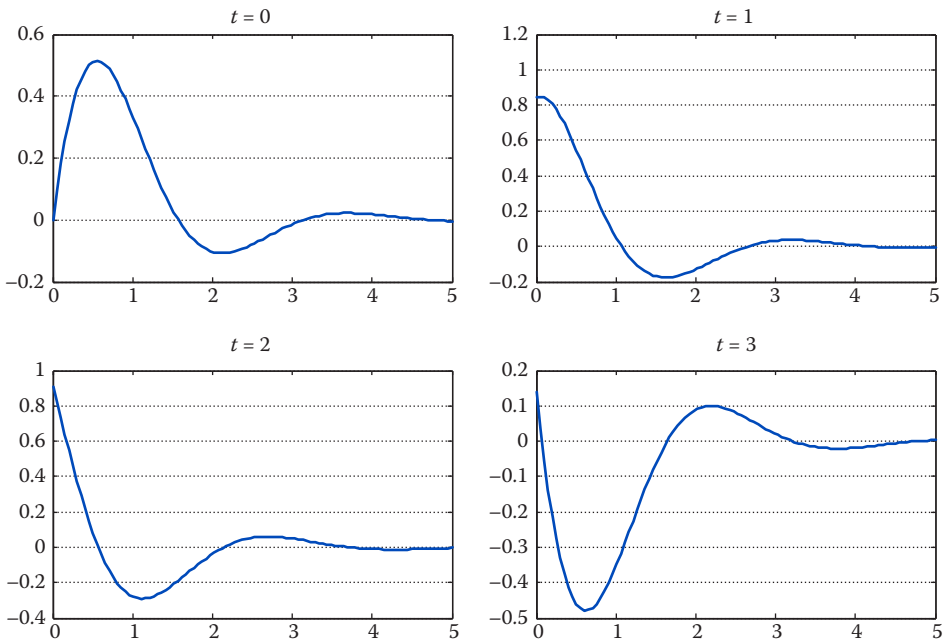


FIGURE 2.2
Subplot in a 2×2 formation.

```
% Initiate Figure 2.2
subplot(2,2,1), plot(x,z(:,1)), title('t = 0')
subplot(2,2,2), plot(x,z(:,2)), title('t = 1')
subplot(2,2,3), plot(x,z(:,3)), title('t = 2')
subplot(2,2,4), plot(x,z(:,4)), title('t = 3')
```

2.6.2 Plotting Analytical Expressions

An alternative way to plot a function is to use the `ezplot` command, which plots the function without requiring data generation. As an example, consider the function $x(t) = e^{-t}(\cos t + \sin t)$ that we previously plotted over the interval $[0, 5]$. The plot in [Figure 2.1](#) can be regenerated using `ezplot` as follows:

```
>> x = sym('exp(-t)*(cos(t)+sin(t))');
>> ezplot(x,[0,5]) % Figure 2.1
```

2.6.3 Multiple Plots

Multiple plots can also be created using `ezplot`. Suppose the two functions $y_1 = 0.7e^{-2t/3} \sin 2t$ and $y_2 = e^{-t/3} \sin 3t$ are to be plotted versus $0 \leq t \leq 5$ in the same graph.

```
>> y1 = sym('0.7*exp(-2*t/3)*sin(2*t)');
>> y2 = sym('exp(-t/3)*sin(3*t)');
>> ezplot(y1,[0,5]) % Initiate Figure 2.3
>> hold on
>> ezplot(y2,[0,5]) % Complete Figure 2.3
```

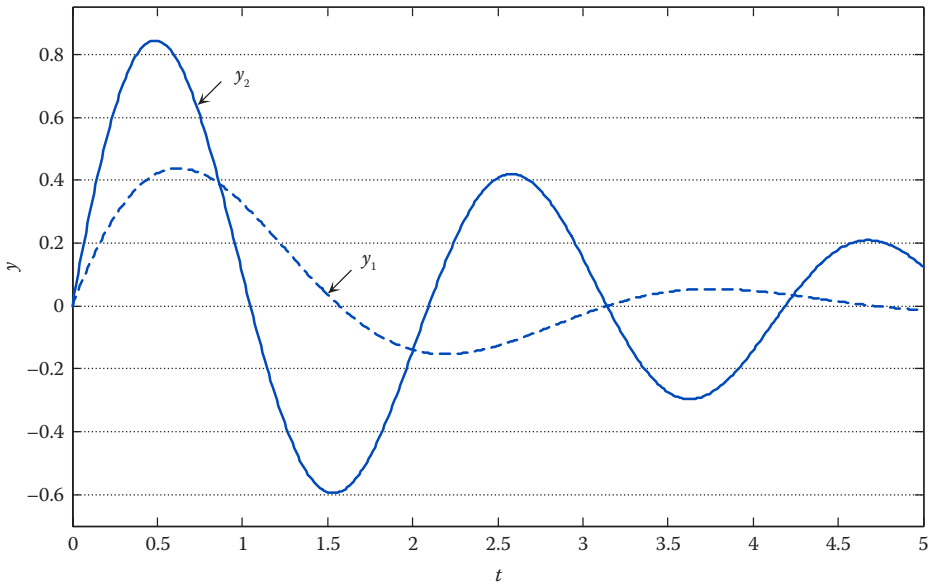


FIGURE 2.3
Multiple plots.

Executing the preceding script generates a figure which does not exactly match [Figure 2.3](#). To enable the interactive plot editing mode in the MATLAB figure window, click the Edit Plot button (⌘) or select Tools > Edit Plot from the main menu. If you enable plot editing mode in the MATLAB figure window, you can perform point-and-click editing of your graph. In this mode, you can modify the appearance of a graphics object by double-clicking the object and changing the values of its properties.

2.7 User-Defined Functions and Script Files

User-defined M file functions and scripts may be created, saved, and edited in MATLAB using the `edit` command. For example, suppose we want to create a function called `Circ` that returns the area of a circle with a given radius. The function can be saved in a folder on the MATLAB path or in the current directory. The current directory can be viewed and/or changed using the drop down menu at the top of the MATLAB command window. Once the current directory has been properly selected, type

```
>> edit Circ
```

A new window (Editor Window) will be opened where the function can be created. The generic structure of a function is

```
function [output variables] = FunctionName(input variables)
% Comments
Expressions/statements
Calculation of all output variables
```

Our user-defined function `Circ` is specifically created as follows.

```
function A = Circ(r)
%
% Circ calculates the area of a circle of a given radius.
%
%   A = Circ(r), where
%
%       r is the radius of the circle,
%
%       A is the area.
%
A = pi*r^2;
```

To compute the area of a circle with radius 1.3, we simply execute

```
>> A = Circ(1.3)
```

```
A =
```

```
5.3093
```

Often, functions with multiple outputs are desired. For example, suppose our function `Circ` is to return two outputs: area of the circle and the perimeter of the circle. We create this as follows.

```
function [A, P] = Circ(r)
%
% Circ calculates the area and the perimeter of a circle of a given radius.
%
%   [A, P] = Circ(r), where
%
%       r is the radius of the circle,
%
%       A is the area,
%       P is the perimeter.
%
A = pi*r^2; P = 2*pi*r;
```

Executing this function for the case of radius 1.3, we find

```
>> [A, P] = Circ(1.3)
```

```
A =
```

```
5.3093    % Of course, this agrees with last result
```

```
P =
```

```
8.1681
```

2.7.1 Setting Default Values for Input Variables

Sometimes, default values are declared for one or more of the input variables of a function. As an example, consider a function `y_int` that returns the y -intercept of a straight line that passes through a specified point with a given slope. Suppose the slope is 1 unless specified otherwise; that is, the default value for slope is 1. If the specified point has coordinates (x_0, y_0) and the slope is m , then the y -intercept is found as $y = y_0 - mx_0$. Based on this we write the function as follows.

```
function y = y_int(x0,y0,m)
%
%   y_int finds the y-intercept of a line passing through a point (x0,y0)
%   with slope m.
%
%   y = y_int(x0,y0,m), where
%
%   x0, y0 are the coordinates of the given point,
%   m is the slope of the line (default 1),
%
%   y is the y-intercept of the line.
%
if nargin < 3 || isempty(m), m = 1; end
y = y0 - m*x0;
```

The MATLAB command `nargin` (number of function input arguments) is used for the purpose of setting default values for one or more of the input variables. The `if` statement here ensures that if the number of input arguments is less than 3, or that the third input argument is empty, then the default value of 1 will be used for m . As an example, to find the y -intercept of the line going through the point $(-1, 2)$ with slope 1, either one of the following two statements may be executed:

```
>> y = y_int(-1,2)           % Number of input arguments is less than 3
y =
    3
```

OR

```
>> y = y_int(-1,2,[])       % The third argument is empty
y =
    3
```

In many cases, at least one of the input variables of a user-defined function happens to be either a MATLAB function or an anonymous function. Consider the following example. Suppose we want to create a user-defined function with the function call `[r, k] = My_func(f, x0, tol, kmax)` where f is an anonymous function, x_0 is a given initial value, tol is the tolerance (with default value $1e-4$), and $kmax$ is the maximum number of steps (with default value 20) to be performed. The function calculates $x_1 = f(x_0)$, followed by $x_2 = f(x_1)$, $x_3 = f(x_2)$, and so on. Operations stop as soon as the distance between two successive elements generated in this manner is less than tol . The outputs of the function are the last element generated when the tolerance condition is met, as well as the number of steps required to achieve that.

```

function [r, k] = My_func(f,x0,tol,kmax)

if nargin < 3 || isempty(tol), tol = 1e-4; end
if nargin < 4 || isempty(kmax), kmax = 20; end

x = zeros(kmax); % Pre-allocate
x(1) = x0;       % Define the first element in the array
for k = 1:kmax,
    x(k+1) = f(x(k));
    if abs(x(k+1)-x(k)) < tol,    % Check tolerance condition
        break
    end
    r = x(k+1); % Set the output as the very last element generated
end

```

Let us now use `My_func` for $f(x) = 3^{-x}$ with $x_0=0$, $tol=1e-3$, and $kmax=20$.

```

>> f = @(x)(3^(-x));
>> [r, k] = My_func(f,0,1e-3)    % kmax uses default value of 20

r =

    0.5473

k =

    15

```

It turns out that the number of steps is 15, which is lower than the maximum of 20. If the returned value for k happens to be 20, further inspection must be conducted. It is possible that exactly 20 steps were needed to meet the desired condition. It is also possible that the maximum number of steps has been exhausted without having the tolerance met. That is why the function needs to be reexecuted with a larger value of $kmax$ than the default (in this case, 20) to gather more precise information.

The user-defined function `My_func` has two outputs: r and k . We may retrieve the value of r only by executing

```
>> r = My_func(f,x0,tol,kmax)
```

This is possible because r is the first output. To have access to the value of k , however, we must execute

```
>> [r, k] = My_func(f,x0,tol,kmax)
```

2.7.2 Creating Script Files

A script file comprises a list of commands as if they were typed at the command line. Script files can be created in the MATLAB Editor, and saved as an M file. For example, typing

```
>> edit My_script
```

opens the Editor Window, where the script can be created and saved under the name `My_script`. It is recommended that a script start with the functions `clear` and `clc`. The first one clears all the previously generated variables, and the second one clears the Command Window. Suppose we type the following lines in the Editor Window:

```
clear
clc
x = 2; N = 10;
a = cos(x)*N^2;
```

While in the Editor Window, select “Run My_script.m” under the Debug pull-down menu. This will execute the lines in the script file and return the Command Prompt. Simply type a at the prompt to see the result.

```
>> My_script
>> a

a =

    -41.6147
```

This can also be done by highlighting the contents and selecting “Evaluate Selection.” An obvious advantage of creating a script file is that it allows us to simply make changes to the contents without having to retype all the commands.

PROBLEM SET (CHAPTER 2)

All calculations must be performed in MATLAB.

- Evaluate the function $g(x, y) = \frac{1}{2}e^{-2x/3} \tan(y + 1)$ for $x = 0.3, y = -0.7$
 - Using the subs command.
 - By conversion into a MATLAB function.
- Evaluate the function $h(x, y) = \cos\left(\frac{1}{3}x - 1\right)\sin\left(y + \frac{1}{2}\right)$ for $x = \frac{3}{4}, y = 1$ using
 - The subs command.
 - An anonymous function.
- Evaluate the vector function $f(x, y) = \begin{cases} x - 1 \\ 2y + x \end{cases}$ for $x = 2, y = \frac{2}{3}$ using
 - The subs command.
 - An anonymous function.
- Evaluate the matrix function $f(x, y) = \begin{bmatrix} 1 - 2x & x + y \\ 0 & \cos y \end{bmatrix}$ for $x = 1, y = -1$
 - Using the subs command.
 - By conversion into a MATLAB function.
- Consider $g(t) = t \sin\left(\frac{1}{2}t\right) + \ln(t - 1)$. Evaluate dg/dt at $t = \frac{4}{3}$
 - Using the subs command.
 - By conversion into a MATLAB function.
- Consider $h(x) = 3^{x-2} \sin x + \frac{2}{3}e^{1-2x}$. Evaluate dh/dx at $x = -0.3$
 - Using the subs command.
 - By conversion into a MATLAB function.
- Evaluate $\left[x^2 + e^{-a(x+1)}\right]^{1/3}$ when $a = -1, x = 3$ using an anonymous function in another anonymous function.

8. Evaluate $\sqrt{x + \ln|1 - e^{(a+2)x/3}|}$ when $a = -3$, $x = 1$ using an anonymous function in another anonymous function.

In Problems 9 through 12 write a script file that employs any combination of the *flow control commands* to generate the given matrix.

$$9. \mathbf{A} = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 0 & 0 \\ 2 & 0 & 3 & 0 & -1 & 0 \\ 0 & 2 & 0 & 4 & 0 & -1 \\ 0 & 0 & 2 & 0 & 5 & 0 \\ 0 & 0 & 0 & 2 & 0 & 6 \end{bmatrix}$$

$$10. \mathbf{A} = \begin{bmatrix} 4 & 1 & -2 & 3 & 0 & 0 \\ 0 & 4 & -1 & 2 & 3 & 0 \\ 0 & 0 & 4 & 1 & -2 & 3 \\ 0 & 0 & 0 & 4 & -1 & 2 \\ 0 & 0 & 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

$$11. \mathbf{B} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -2 & 2 & 0 & 0 & 0 \\ -1 & 0 & 3 & 3 & 0 & 0 \\ 0 & 1 & 0 & -4 & 4 & 0 \\ 0 & 0 & -1 & 0 & 5 & 5 \\ 0 & 0 & 0 & 1 & 0 & -6 \end{bmatrix}$$

$$12. \mathbf{B} = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -2 & 0 \\ 4 & 0 & 2 & 0 & -3 \\ 0 & 5 & 0 & -3 & 0 \\ 0 & 0 & 6 & 0 & 4 \end{bmatrix}$$

13. Using any combination of commands `diag`, `triu`, and `tril`, construct matrix \mathbf{B} from \mathbf{A} .

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & -1 & 2 \\ 3 & 0 & 4 & 1 \\ 1 & 5 & -1 & 3 \\ 0 & 2 & 6 & 1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 1 & 5 & 0 & 0 \\ 0 & 2 & 6 & 0 \end{bmatrix}$$

14. Using any combination of commands `diag`, `triu`, and `tril`, construct matrix **B** from **A**.

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & -1 & 2 \\ 3 & 0 & 4 & 1 \\ 1 & 5 & -1 & 3 \\ 0 & 2 & 6 & 1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 1 & -1 & 2 \\ 3 & 0 & 4 & 1 \\ 0 & 5 & 0 & 3 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

15. Plot $\int_1^t e^{-2x} \sin x dx$ versus $-1 \leq t \leq 1$, add grid and label.
16. Plot $\int_0^t (x+t)^2 e^{-(t-x)} dx$ versus $-2 \leq t \leq 1$, add grid and label.
17. Plot $y_1 = \frac{1}{3} e^{-t} \sin(t\sqrt{2})$ and $y_2 = e^{-t/2}$ versus $0 \leq t \leq 5$ in the same graph. Add grid, and label.
18. Generate 100 points for each of the two functions in Problem 17 and plot versus $0 \leq t \leq 5$ in the same graph. Add grid, and label.
19. Evaluate $\int_0^{\infty} \frac{\sin \omega}{\omega} d\omega$.
20. Plot $u(x, t) = \cos(1.7x) \sin(3.2t)$ versus $0 \leq x \leq 5$ for four values of $t = 1, 1.5, 2, 2.5$ in a 2×2 tile. Add grid and title.
21. Plot $u(x, t) = (1 - \sin x)e^{-(t+1)}$ versus $0 \leq x \leq 5$ for two values of $t = 1, 3$ in a 1×2 tile. Add grid and title.
22. Given that $f(x) = e^{-2x} + \cos(x + 1)$, plot $f'(x)$ versus $0 \leq x \leq 8$.
23. Write a user-defined function with function call `val = f_eval(f, a, b)` where f is an anonymous function, and a and b are constants such that $a < b$. The function calculates the midpoint m of the interval $[a, b]$ and returns the value of $f(a) + \frac{1}{2} f(m) + f(b)$. Execute `f_eval` for $f = e^{-x/3}$, $a = -4$, $b = 2$.
24. Write a user-defined function with function call `m = mid_seq(a, b, tol)` where a and b are constants such that $a < b$, and `tol` is a specified tolerance. The function first calculates the midpoint m_1 of the interval $[a, b]$, then the midpoint m_2 of $[a, m_1]$, then the midpoint m_3 of $[a, m_2]$, and so on. The process terminates when two successive midpoints are within `tol` of each other. Allow a maximum of 20 iterations. The output of the function is the sequence m_1, m_2, m_3, \dots . Execute the function for $a = -4$, $b = 10$, $\text{tol} = 10^{-3}$.
25. Write a user-defined function with function call `C = temp_conv(F)` where F is temperature in Fahrenheit, and C is the corresponding temperature in Celsius. Execute the function for $F = 87$.
26. Write a user-defined function with function call `P = partial_eval(f, a)` where f is a function defined symbolically, and a is a constant. The function returns the value of $f' + f''$ at $x = a$. Execute the function for $f = 3x^2 - e^{x/3}$, and $a = 1$.
27. Write a user-defined function with function call `P = partial_eval2(f, g, a)` where f and g are functions defined symbolically, and a is a constant. The function

returns the value of $f' + g'$ at $x = a$. Execute the function for $f = x^2 + e^{-x}$, $g = \sin(0.3x)$, and $a = 0.8$.

28. Write a user-defined function with function call `[r, k] = root_finder(f, x0, kmax, tol)` where f is an anonymous function, x_0 is a specified value, k_{\max} is the maximum number of iterations, and tol is a specified tolerance. The function sets $x_1 = x_0$, calculates $|f(x_1)|$, and if it is less than the tolerance, then x_1 approximates the root r . If not, it will increment x_1 by 0.01 to obtain x_2 , repeat the procedure, and so on. The process terminates as soon as $|f(x_k)| < \text{tol}$ for some k . The outputs of the function are the approximate root and the number of iterations it took to find it. Execute the function for $f(x) = x^2 - 3.3x + 2.1$, $x_0 = 0.5$, $k_{\max} = 50$ $\text{tol} = 10^{-2}$.
29. Repeat Problem 28 for $f(x) = 3 + \ln(2x - 1) - e^x$, $x_0 = 1$, $k_{\max} = 25$ $\text{tol} = 10^{-2}$.
30. Write a user-defined function with function call `[opt, k] = opt_finder(fp, x0, kmax, tol)` where fp is the derivative (as a MATLAB function) of a given function f , x_0 is a specified value, k_{\max} is the maximum number of iterations, and tol is a specified tolerance. The function sets $x_1 = x_0$, calculates $|\text{fp}(x_1)|$, and if it is less than the tolerance, then x_1 approximates the critical point opt at which the derivative is near zero. If not, it will increment x_1 by 0.1 to obtain x_2 , repeat the procedure, and so on. The process terminates as soon as $|\text{fp}(x_k)| < \text{tol}$ for some k . The outputs are the approximate optimal point and the number of iterations it took to find it. Execute the function for $f(x) = x + (x - 2)^2$, $x_0 = 1$, $k_{\max} = 50$ $\text{tol} = 10^{-3}$.

3

Numerical Solution of Equations of a Single Variable

This chapter focuses on numerical solution of equations of a single variable, which appear in the general form

$$f(x) = 0 \tag{3.1}$$

Graphically, a solution (or root) of $f(x) = 0$ refers to the point of intersection of $f(x)$ and the x -axis. Therefore, depending on the nature of the graph of $f(x)$ in relation to the x -axis, Equation 3.1 may have a unique solution, multiple solutions, or no solution. A root of an equation can sometimes be determined analytically resulting in an exact solution in closed form. For instance, the equation $e^{3x} - 2 = 0$ can be solved analytically to obtain a unique solution $x = \frac{1}{3} \ln 2$. In most situations, however, this is not possible and the root(s) must be found numerically. As an example, consider the equation $2 - x + \cos x = 0$. [Figure 3.1](#) shows that this equation has one solution only, which may be approximated to within a desired accuracy with the aid of a numerical method.

3.1 Numerical Solution of Equations

As described in [Figure 3.2](#), numerical methods for solving an equation are divided into three main categories: bracketing methods, open methods, and using the built-in MATLAB function `fzero`.

Bracketing methods require that an interval containing the root first be identified. Referring to [Figure 3.3](#), this means an interval $[a, b]$ with the property that $f(a)f(b) < 0$ so that a root lies in $[a, b]$. The length of the interval is then reduced in succession until a desired accuracy is satisfied. Exactly how this interval gets narrowed in each step depends on the specific method used. It is readily seen that bracketing methods always converge to the root. Open methods require an initial estimate of the solution, somewhat close to the intended root. Subsequently, more accurate estimates are successively generated by a specific method; [Figure 3.4](#). Open methods are more efficient than bracketing methods, but do not always generate a sequence that converges to the root. The built-in function `fzero` finds the root of a function f near a specified point, or in a specified interval $[a, b]$ such that $f(a)f(b) < 0$.

3.2 Bisection Method

The bisection method is the simplest bracketing method to find a root of $f(x) = 0$. It is assumed that $f(x)$ is continuous on an interval $[a, b]$ and has a root there so that $f(a)$ and $f(b)$

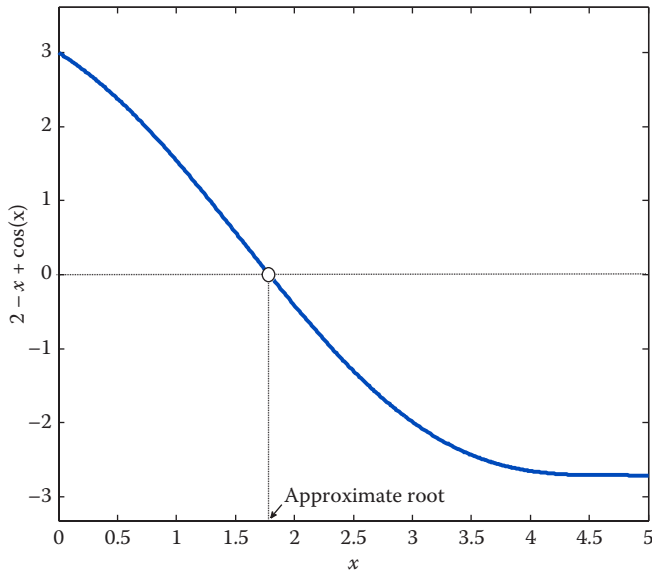


FIGURE 3.1

Approximate solution of $2 - x + \cos x = 0$.

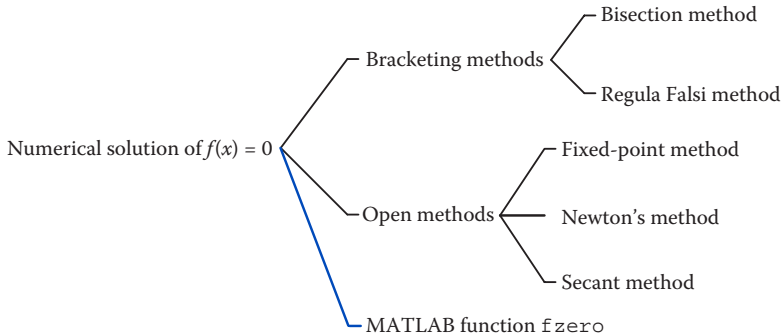


FIGURE 3.2

Classification of methods to solve an equation of one variable.

have opposite signs, hence $f(a)f(b) < 0$. The procedure goes as follows: Locate the midpoint of $[a, b]$, that is, $c_1 = \frac{1}{2}(a + b)$, [Figure 3.5](#). If $f(a)$ and $f(c_1)$ have opposite signs, the interval $[a, c_1]$ contains the root and will be retained for further analysis; that is, the left end is retained while the right end is adjusted. If $f(b)$ and $f(c_1)$ have opposite signs, we continue with $[c_1, b]$; that is, the right end is kept while the left end is adjusted. In [Figure 3.5](#) it so happens that the interval $[c_1, b]$ brackets the root and is retained. Since the right endpoint is unchanged, we update the interval $[a, b]$ by resetting the left endpoint $a = c_1$. The process is repeated until the length of the most recent interval $[a, b]$ satisfies the desired accuracy.

The initial interval $[a, b]$ has length $b - a$. Beyond that, the first generated interval has length $\frac{1}{2}(b - a)$, the next interval $\frac{1}{4}(b - a)$, and so on. Thus, the n -th interval constructed

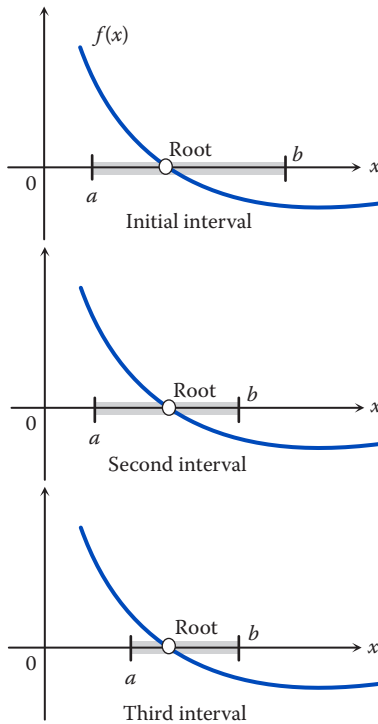


FIGURE 3.3

Philosophy of bracketing methods.

in this manner has length $(b - a)/2^{n-1}$, and because it brackets the root, the absolute error associated with the n th iteration satisfies

$$|e_n| \leq \frac{b-a}{2^{n-1}} \quad (b > a)$$

This upper bound is usually larger than the actual error at the n th step. If the bisection method is used to approximate the root of $f(x) = 0$ within a prescribed tolerance $\epsilon > 0$, then it can be shown that the number N of iterations needed to meet the tolerance condition satisfies

$$N > \frac{\ln(b-a) - \ln \epsilon}{\ln 2} \quad (3.2)$$

The user-defined function `Bisection` shown below generates a sequence of values (midpoints) that ultimately converges to the true solution. The iterations terminate when $\frac{1}{2}(b - a) < \epsilon$, where ϵ is a prescribed tolerance. The output of the function is the last generated estimate of the root at the time the tolerance was met. It also returns a table that comprises iteration counter, interval endpoints, and interval midpoint per iteration, as well as the value of $\frac{1}{2}(b - a)$ to see when the terminating condition is satisfied.

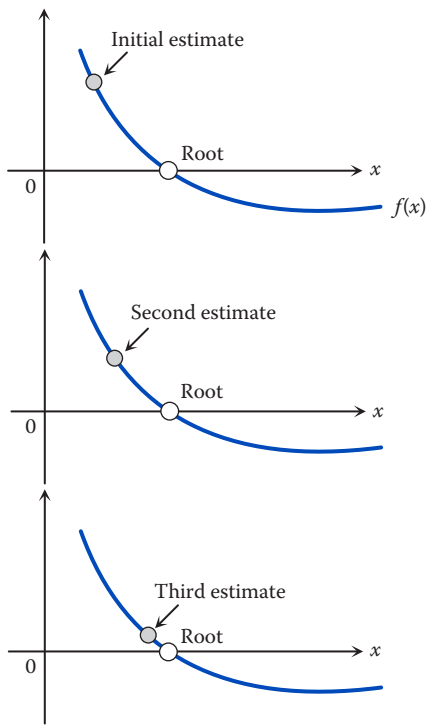


FIGURE 3.4
Philosophy of open methods.

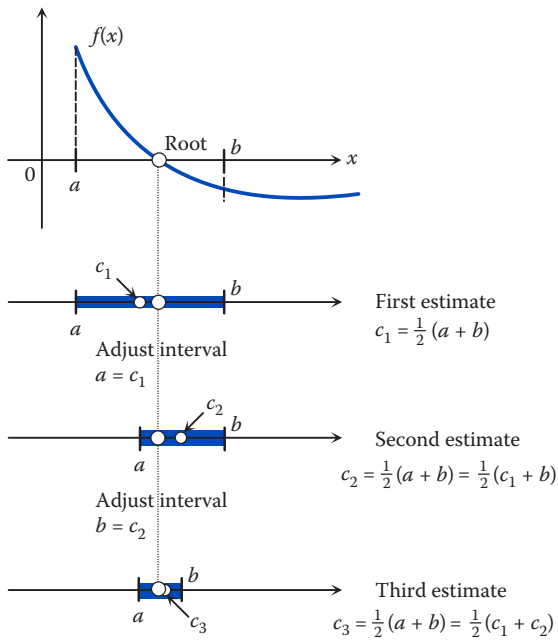


FIGURE 3.5
Bisection method: three iterations shown.

```

function c = Bisection(f, a, b, kmax, tol)
%
% Bisection uses the bisection method to find a root of f(x) = 0
% in the interval [a,b].
%
% c = Bisection(f, a, b, kmax, tol), where
%
% f is an anonymous function representing f(x),
% a and b are the endpoints of interval [a,b],
% kmax is the maximum number of iterations (default 20),
% tol is the scalar tolerance for convergence (default 1e-4),
%
% c is the approximate root of f(x) = 0.
%
if nargin < 5 || isempty(tol), tol = 1e-4; end
if nargin < 4 || isempty(kmax), kmax = 20; end
if f(a)*f(b) > 0
    c = 'failure';
    return
end
disp(' k      a      b      c      (b-a)/2')
for k = 1:kmax,
    c = (a+b)/2;    % Find the first midpoint
    if f(c) == 0,  % Stop if a root has been found
        return
    end
    fprintf('%3i    %11.6f%11.6f%11.6f%11.6f\n',k,a,b,c,(b-a)/2)
    if (b-a)/2 < tol,    % Stop if tolerance is met
        return
    end
    if f(b)*f(c) > 0    % Check sign changes
        b = c;        % Adjust the endpoint of interval
    else a = c;
    end
end
end

```

EXAMPLE 3.1: BISECTION METHOD

The equation $x \cos x + 1 = 0$ has a root in the interval $[-2, 4]$, as shown in [Figure 3.6](#):

```

>> f = @(x)(x*cos(x)+1);
>> ezplot(f,[-2,4])

```

Define $f(x) = x \cos x + 1$ so that $f(-2) > 0$ and $f(4) < 0$. We will perform two steps of the bisection method as follows. The first midpoint is found as $c_1 = \frac{1}{2}(-2 + 4) = 1$. Since the function value at this point is $f(c_1) = f(1) > 0$, the root must be in $[1, 4]$. This means the left end is adjusted as $a = c_1 = 1$ while $b = 4$ remains unchanged. The next midpoint is then calculated as $c_2 = \frac{1}{2}(1 + 4) = 2.5$. Since $f(c_2) = f(2.5) < 0$, the root must lie in $[1, 2.5]$. This process continues until a desired accuracy is achieved. In particular, if we execute the user-defined function `Bisection` with $\varepsilon = 10^{-2}$ and maximum 20 iterations, the following results are obtained.

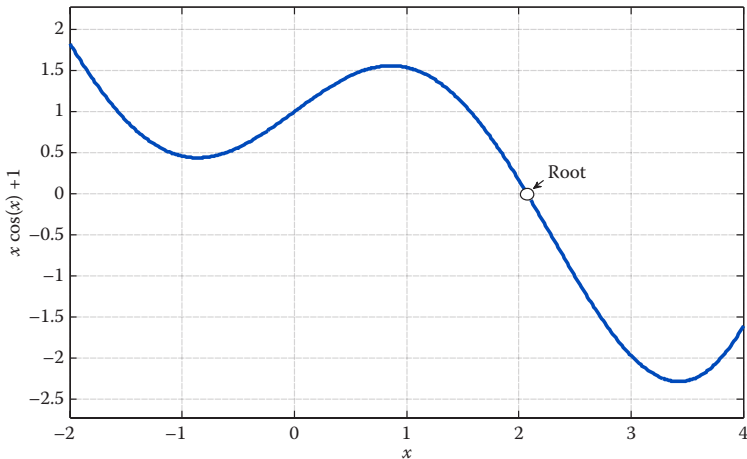


FIGURE 3.6
Location of the root of $x \cos x + 1 - 0$ in $[-2, 4]$.

```
>> c = Bisection(f, -2, 4, [], 1e-2)
```

k	a	b	c	(b-a)/2
1	-2.000000	4.000000	1.000000	3.000000
2	1.000000	4.000000	2.500000	1.500000
3	1.000000	2.500000	1.750000	0.750000
4	1.750000	2.500000	2.125000	0.375000
5	1.750000	2.125000	1.937500	0.187500
6	1.937500	2.125000	2.031250	0.093750
7	2.031250	2.125000	2.078125	0.046875
8	2.031250	2.078125	2.054688	0.023438
9	2.054688	2.078125	2.066406	0.011719
10	2.066406	2.078125	2.072266	0.005859

```
c =  
2.0723
```

The data in the first three rows confirm the hand calculations. Iterations stopped when $\frac{1}{2}(b-a) = 0.005859 < \epsilon = 0.01$. Note that by Equation 3.2, we have

$$N > \frac{\ln(b-a) - \ln \epsilon}{\ln 2} \stackrel{a=-2, b=4}{\underset{\epsilon=0.01}{=}} \frac{\ln 6 - \ln 0.01}{\ln 2} = 9.23$$

which means at least 10 iterations are required for convergence. This is in agreement with the findings here, as we saw that tolerance was met after 10 iterations. The accuracy of the solution estimate will improve if a smaller tolerance is imposed.

3.2.1 MATLAB Built-In Function `fzero`

The `fzero` function in MATLAB finds the roots of $f(x) = 0$ for a real function $f(x)$.

`FZERO` Scalar nonlinear zero finding.

`X = FZERO(FUN,X0)` tries to find a zero of the function `FUN` near `X0`, if `X0` is a scalar.

The `fzero` function uses a combination of the bisection, secant, and inverse quadratic interpolation methods. If we know two points where the function value differs in sign, we can specify this starting interval using a two-element vector for `x0`. This algorithm is guaranteed to return a solution. If we specify a scalar starting point `x0`, then `fzero` initially searches for an interval around this point where the function changes sign. If an interval is found, then `fzero` returns a value near where the function changes sign. If no interval is found, `fzero` returns a NaN value.

The built-in function `fzero` can be used to confirm the approximate root in Example 3.1. This can be done in one of two ways. As a first option, we may specify a point near which the root must be found. For instance, selecting `x0 = 1` leads to

```
>> fzero(f,1)
ans =
    2.0739
```

As a second option, we may identify two points where the function value differs in sign. For instance, choosing the interval `[1, 3]` leads to

```
>> fzero(f,[1,3])
ans =
    2.0739
```

The accuracy of the approximate root (2.0723) returned by the user-defined function `Bisection` can be improved by choosing a smaller tolerance. For example, the reader can verify that executing `Bisection` with `tol = 1e-8` returns a root estimate (2.0739) that agrees with `fzero` to at least 4 decimal places, but requires 30 iterations.

3.3 Regula Falsi Method (Method of False Position)

The regula falsi method is another bracketing method to find a root of $f(x) = 0$. Once again, it is assumed that $f(x)$ is continuous on an interval $[a, b]$ and has a root there so that $f(a)$ and $f(b)$ have opposite signs, $f(a)f(b) < 0$. The technique is geometrical in nature and described as follows. Let $[a_1, b_1] = [a, b]$. Connect points $A:(a_1, f(a_1))$ and $B:(b_1, f(b_1))$ by a straight line as in [Figure 3.7](#) and let c_1 be its x -intercept. If $f(a_1)f(c_1) < 0$, then $[a_1, c_1]$ brackets the root. Otherwise, the root is in $[c_1, b_1]$. In [Figure 3.7](#) it just so happens that $[a_1, c_1]$ brackets the root. This means the left end is unchanged, while the right end is adjusted to c_1 . Therefore, the interval that is used in the next iteration is $[a_2, b_2]$ where $a_2 = a_1$ and $b_2 = c_1$. Continuing this process generates a sequence c_2, c_3, \dots that eventually converges to the root. In the case shown in [Figure 3.7](#) the curve of $f(x)$ is concave up and the left end of the interval remains fixed at least through the first three iterations shown. This issue will be addressed shortly.

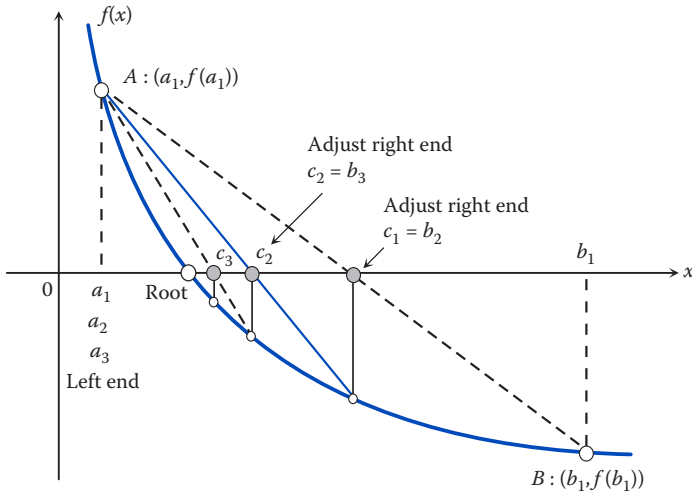


FIGURE 3.7

Method of false position (regula falsi).

Analytically, the procedure is illustrated as follows. The equation of the line connecting points A and B is

$$y - f(b_1) = \frac{f(b_1) - f(a_1)}{b_1 - a_1}(x - b_1)$$

To find the x -intercept, set $y = 0$ and solve for $x = c_1$:

$$c_1 = b_1 - \frac{b_1 - a_1}{f(b_1) - f(a_1)} f(b_1) \stackrel{\text{Simplify}}{=} \frac{a_1 f(b_1) - b_1 f(a_1)}{f(b_1) - f(a_1)}$$

Generalizing this result, the sequence of points that converges to the root is generated via

$$c_n = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}, \quad n = 1, 2, 3, \dots \quad (3.3)$$

The user-defined function `RegulaFalsi` generates a sequence of elements that eventually converges to the root of $f(x) = 0$. The iterations stop when two consecutive x -intercepts are within an acceptable distance from one another. That is, the terminating condition is $|c_{n+1} - c_n| < \epsilon$, where ϵ is the imposed tolerance. The outputs are the approximate root and the number of iterations required to meet the tolerance. The function also returns a table comprised of the intervals containing the root in all iterations performed.

```
function [r, k] = RegulaFalsi(f, a, b, kmax, tol)
%
% RegulaFalsi uses the regula falsi method to approximate a root of f(x) = 0
% in the interval [a,b].
```

```

%
% [r, k] = RegulaFalsi(f, a, b, kmax, tol), where
%
% f is an anonymous function representing f(x),
% a and b are the limits of interval [a,b],
% kmax is the maximum number of iterations (default 20),
% tol is the scalar tolerance for convergence (default 1e-4),
%
% r is the approximate root of f(x) = 0,
% k is the number of iterations needed for convergence.
%
if nargin < 5 || isempty(tol), tol = 1e-4; end
if nargin < 4 || isempty(kmax), kmax = 20; end
c = zeros(1,kmax); % Pre-allocate
if f(a)*f(b) > 0
    r = 'failure';
    return
end
disp(' k      a          b')
for k = 1:kmax,
    c(k) = (a*f(b)-b*f(a))/(f(b)-f(a)); % Find the x-intercept
    if f(c(k)) == 0 % Stop if a root has been found
        return
    end
    fprintf('%2i      %11.6f%11.6f\n',k,a,b)
    if f(b)*f(c(k)) > 0 % Check sign changes
        b = c(k); % Adjust the endpoint of interval
    else a = c(k);
    end
    c(k+1) = (a*f(b)-b*f(a))/(f(b)-f(a)); % Find the next x-intercept
    if abs(c(k+1)-c(k)) < tol, % Stop if tolerance is met
        r = c(k+1);
        return
    end
end
end

```

EXAMPLE 3.2: REGULA FALSI METHOD

Reconsider the equation $x \cos x + 1 = 0$ and the interval $[-2, 4]$ that contains its root. Letting $f(x) = x \cos x + 1$, we have $f(-2) > 0$ and $f(4) < 0$. We will perform two steps of the regula falsi method. First, set $[a_1, b_1] = [-2, 4]$. Then,

$$c_1 = \frac{a_1 f(b_1) - b_1 f(a_1)}{f(b_1) - f(a_1)} = 1.189493$$

Since $f(c_1) > 0$, the root must lie in $[c_1, b_1]$ so that the left endpoint is adjusted to $a_2 = c_1$ and the right end remains unchanged, $b_2 = b_1$. Therefore, the updated interval is $[a_2, b_2] = [1.189493, 4]$. Next,

$$c_2 = \frac{a_2 f(b_2) - b_2 f(a_2)}{f(b_2) - f(a_2)} = 2.515720$$

This process continues until a desired accuracy is achieved. In particular, if we execute the user-defined function `RegulaFalsi` with $\epsilon = 10^{-2}$ and maximum 20 iterations, the following results are obtained.

```
>> f = @(x)(x*cos(x)+1);
>> [r, k] = RegulaFalsi(f, -2, 4, [], 1e-2)

k          a          b
1      -2.000000    4.000000
2      1.189493    4.000000
3      1.189493    2.515720
4      1.960504    2.515720

r =
    2.0738

k =
     4
```

The boxed entries confirm the hand calculations. We observe that, for the same tolerance ($1e-2$), the root estimate returned by `RegulaFalsi` is reached faster and is more accurate than the one returned by `Bisection`. Also note that the functions `Bisection` and `RegulaFalsi` use different criteria to terminate the respective iterations.

3.3.1 Modified Regula Falsi Method

In many cases, the curve representing $f(x)$ happens to be concave up or concave down. In these situations, when regula falsi is employed, one of the endpoints of the interval remains the same through all iterations, while the other endpoint advances in the direction of the root. For instance, in [Figure 3.7](#), the function is concave up, the left endpoint remains unchanged, and the right endpoint moves toward the root. The regula falsi method can be modified such that both ends of the interval move toward the root, thus improving the rate of convergence. Among many proposed modifications, there is one that is presented here. Reconsider the scenario portrayed in [Figure 3.7](#) now shown in [Figure 3.8](#). If endpoint a

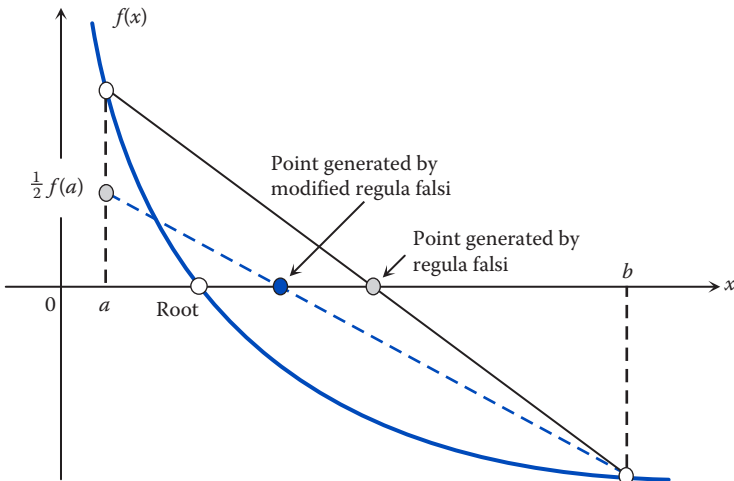


FIGURE 3.8
Modified regula falsi method.

remains stagnant after, say, three consecutive iterations, the usual straight line is replaced with one that is less steep, going through the point at $\frac{1}{2}f(a)$ instead of $f(a)$, which causes the x -intercept to be closer to the actual root. It is possible that this still does not force the endpoint a to move toward the root. In that event, if endpoint a remains the same after three more iterations, the modified line will be replaced with yet a less steep line going through $\frac{1}{4}f(a)$, and so on; See Problem Set.

3.4 Fixed-Point Method

The fixed-point method is an open method to find a root of $f(x) = 0$. The idea is to rewrite $f(x) = 0$ as $x = g(x)$ for a suitable $g(x)$, which is called an iteration function. A point of intersection of $y = g(x)$ and $y = x$ is called a fixed point of $g(x)$. A fixed point of $g(x)$ is also a root of the original equation $f(x) = 0$. As an example, consider $e^{-x/2} - x = 0$ and its root as shown in Figure 3.9. The equation is rewritten as $x = e^{-x/2}$ so that $g(x) = e^{-x/2}$ is an iteration function. It is observed that $g(x)$ has only one fixed point, which is the only root of the original equation. It should be noted that for a given equation $f(x) = 0$ there usually exist more than one iteration function. For instance, $e^{-x/2} - x = 0$ can also be rewritten as $x = -2 \ln x$ so that $g(x) = -2 \ln x$.

A fixed point of $g(x)$ is found numerically via the fixed-point iteration:

$$x_{n+1} = g(x_n), \quad n = 1, 2, 3, \dots, \quad x_1 = \text{initial guess} \quad (3.4)$$

The procedure begins with an initial guess x_1 near the fixed point. The next point x_2 is found as $x_2 = g(x_1)$, followed by $x_3 = g(x_2)$, and so on. This continues until convergence

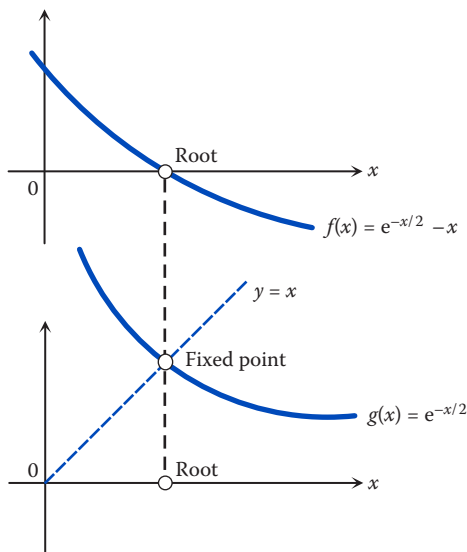


FIGURE 3.9

Root of an equation interpreted as a fixed point of an iteration function.

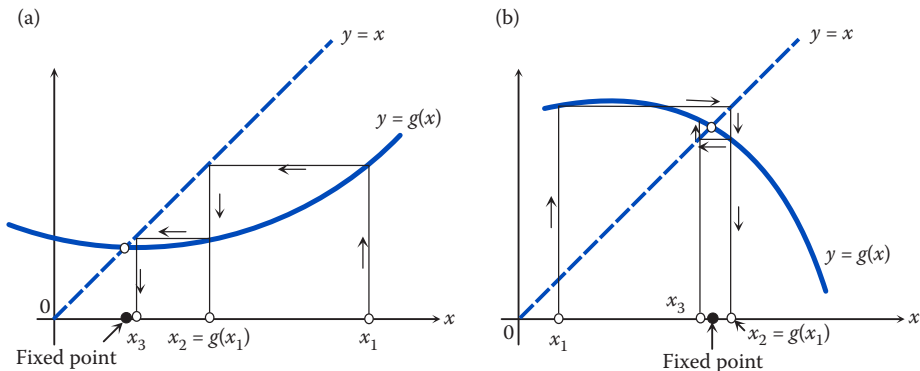


FIGURE 3.10

Fixed-point iteration: (a) monotone convergence, and (b) oscillatory convergence.

is observed, that is, until two successive points are within a prescribed distance of one another, or

$$|x_{n+1} - x_n| < \varepsilon$$

Two types of convergence can be exhibited by the fixed-point iteration: monotone and oscillatory, as illustrated in Figure 3.10. In a monotone convergence, the elements of the generated sequence converge to the fixed point from one side, while in an oscillatory convergence, the elements bounce from one side of the fixed point to the other as they approach it.

3.4.1 Selection of a Suitable Iteration Function

As mentioned above, there is usually more than one way to rewrite a given equation $f(x) = 0$ as $x = g(x)$. An iteration function $g(x)$ must be suitably selected so that when used in Equation 3.4, the iterations converge to the fixed point. In some cases, more than one of the possible forms can be successfully used. Sometimes, none of the forms is suitable, which means that the root cannot be found by the fixed-point method. When there are multiple roots, one possible form may be used to find one root, while another form leads to another root. As demonstrated in Theorem 3.1, there is a way to decide whether a fixed-point iteration converges or not for a specific choice of iteration function.

Theorem 3.1: Convergence of Fixed-Point Iteration

Let $r \in I$ be a fixed point of $g(x)$. Assume that $g(x)$ has a continuous derivative in interval I , and $|g'(x)| \leq K < 1$ for all $x \in I$. Then, for any initial point $x_1 \in I$, the fixed-point iteration in Equation 3.4 generates a sequence $\{x_n\}$ that converges to r . Furthermore, if $e_1 = x_1 - r$ and $e_n = x_n - r$ denote the initial error and the error at the n th iteration, we have

$$|e_n| \leq K^n |e_1| \tag{3.5}$$

Proof

Suppose $x \in I$. Then, by the mean value theorem (MVT) for derivatives, there exists a point $\xi \in (x, r)$ such that

$$g(x) - g(r) = g'(\xi)(x - r)$$

Next, let us consider the left side of Equation 3.5. Noting that $r = g(r)$ and $x_n = g(x_{n-1})$, we have

$$\begin{aligned} |e_n| &= |x_n - r| = |g(x_{n-1}) - g(r)| \stackrel{\text{MVT}}{=} |g'(\xi)| |x_{n-1} - r| \\ &\leq K |x_{n-1} - r| = K |g(x_{n-2}) - g(r)| \\ &\stackrel{\text{MVT}}{=} K |g'(\eta)| |x_{n-2} - r| \\ &\leq K^2 |x_{n-2} - r| \leq \dots \leq K^n |x_1 - r| = K^n |e_1| \end{aligned}$$

Since $K < 1$ by assumption, $|e_n| = |x_n - r| \rightarrow 0$ as $n \rightarrow \infty$. That completes the proof. ■

3.4.2 A Note on Convergence

Following Theorem 3.1, if $|g'(x)| < 1$ near a fixed point of $g(x)$, convergence is guaranteed. In other words, *if in a neighborhood of a root, the curve representing $g(x)$ is less steep than the line $y = x$, then the fixed-point iteration converges to that root.* Note that this is a sufficient, and not necessary, condition for convergence.

The user-defined function `FixedPoint` uses an initial point x_1 and generates a sequence of elements $\{x_n\}$ that eventually converges to the fixed point of $g(x)$. The iterations stop when two consecutive elements are sufficiently close to one another, that is, $|x_{n+1} - x_n| < \varepsilon$, where ε is the tolerance. The outputs are the approximate value of the fixed point and the number of iterations needed to meet the tolerance.

```
function [r, n] = FixedPoint(g, x1, kmax, tol)
%
% FixedPoint uses the fixed-point method to approximate a fixed point
% of g(x).
%
% [r, n] = FixedPoint(g, x1, kmax, tol), where
%
% g is an anonymous function representing g(x),
% x1 is the initial point,
% kmax is the maximum number of iterations (default 20),
% tol is the scalar tolerance for convergence (default 1e-4),
%
% r is the approximate fixed point of g(x),
% n is the number of iterations needed for convergence.
%
if nargin < 4 || isempty(tol), tol = 1e-4; end
if nargin < 3 || isempty(kmax), kmax = 20; end
x = zeros(1, kmax);
x(1) = x1;
for n = 1:kmax,
```

```

x(n+1) = g(x(n));
if abs(x(n+1) - x(n)) < tol
    r = x(n+1);
    return
end
end
end
r = 'failure'; % Failure to converge after kmax iterations

```

EXAMPLE 3.3: FIXED-POINT METHOD

The objective is to find the roots of $x - 2^{-x} = 0$ using the fixed-point method. Rewrite the equation as $x = 2^{-x}$ so that $g(x) = 2^{-x}$. The (only) fixed point can be roughly located as in Figure 3.11.

```

>> g = @(x)(2^(-x));
>> ezplot(g,[0,2])
>> hold on
>> syms x
>> ezplot(x,[0,2])    % Figure 3.11

```

Before applying the fixed-point iteration, we need to check the condition of convergence, Theorem 3.1, as follows:

$$|g'(x)| = |-2^{-x} \ln 2| = 2^{-x} \ln 2 < 1 \Rightarrow x > 0.5288$$

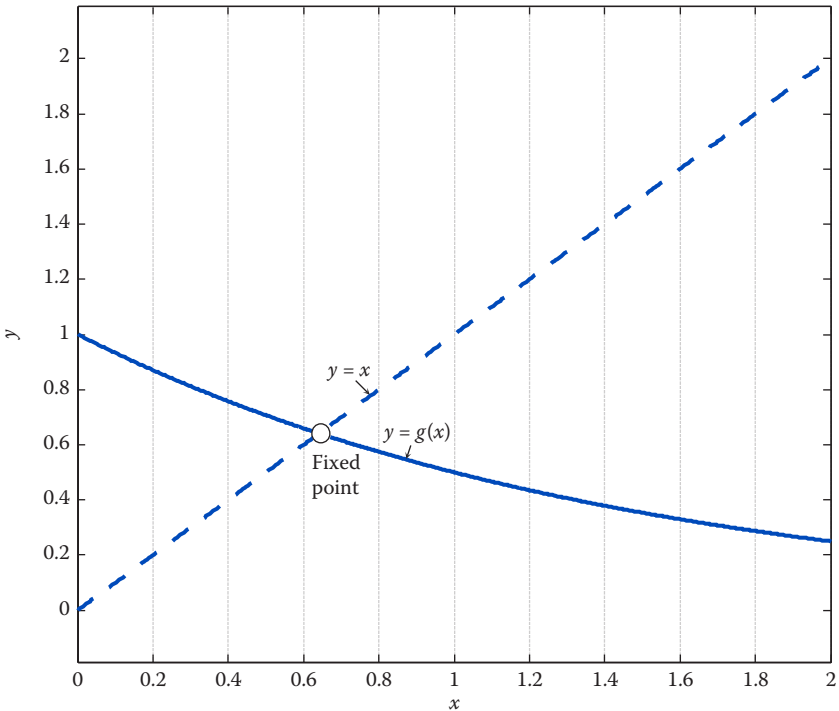


FIGURE 3.11

Location of the fixed point of $g(x) = 2^{-x}$.

This means if the fixed point is in an interval comprised of values of x larger than 0.5288, the fixed-point iteration is guaranteed to converge. Figure 3.11 confirms that this condition is indeed satisfied.

We will execute the user-defined function `FixedPoint` with $x_1 = 0$ and default values for `kmax` and `tol`.

```
>> [r, n] = FixedPoint(g, 0)
```

```
r =
```

```
0.6412
```

```
n =
```

```
13
```

Therefore, the fixed point of $g(x) = 2^{-x}$, which is the root of $x - 2^{-x} = 0$, is found after 13 iterations. The reader may verify that the convergence is oscillatory. In fact, the sequence of elements generated by the iteration is

```
0.0000    1.0000    0.5000    0.7071    0.6125    0.6540    0.6355    0.6437
0.6401    0.6417    0.6410    0.6413    0.6411    0.6412
```

EXAMPLE 3.4: SELECTION OF A SUITABLE ITERATION FUNCTION

Consider the quadratic equation $x^2 - 4x + 1 = 0$. As stated earlier, there is more than one way to construct an iteration function $g(x)$. For instance, two such forms are

$$g_1(x) = \frac{1}{4}(x^2 + 1), \quad g_2(x) = 4 - \frac{1}{x}$$

Let us first consider $g_1(x)$, Figure 3.12. The portion of the curve of $g_1(x)$ below point A is less steep than $y = x$. Starting at any arbitrary point in that region, we see that the iteration always converges to the smaller root B . On the other hand, above point A , the curve is steeper than the line $y = x$. Starting at any point there, the iteration will diverge. Thus, only the smaller of the two roots can be approximated if $g_1(x)$ is used. Now, referring to Figure 3.13, the curve of $g_2(x)$ is much less steep near A than it is near B . And, it appears that starting at any point above or below A (also above B), the iteration converges to the larger root.

Let us inspect the condition of convergence as stated in Theorem 3.1. In relation to $g_1(x)$, we have

$$|g'_1(x)| = \left| \frac{1}{2}x \right| < 1 \Rightarrow |x| < 2 \Rightarrow -2 < x < 2$$

The fixed point B falls inside this interval, and starting at around $x_1 = 2$ (Figure 3.12), the sequence did converge toward B . When we started at around $x_1 = 4$, however, the sequence showed divergence. In relation to $g_2(x)$,

$$|g'_2(x)| = \left| \frac{1}{x^2} \right| < 1 \Rightarrow |x^2| > 1 \Rightarrow x < -1 \text{ or } x > 1$$

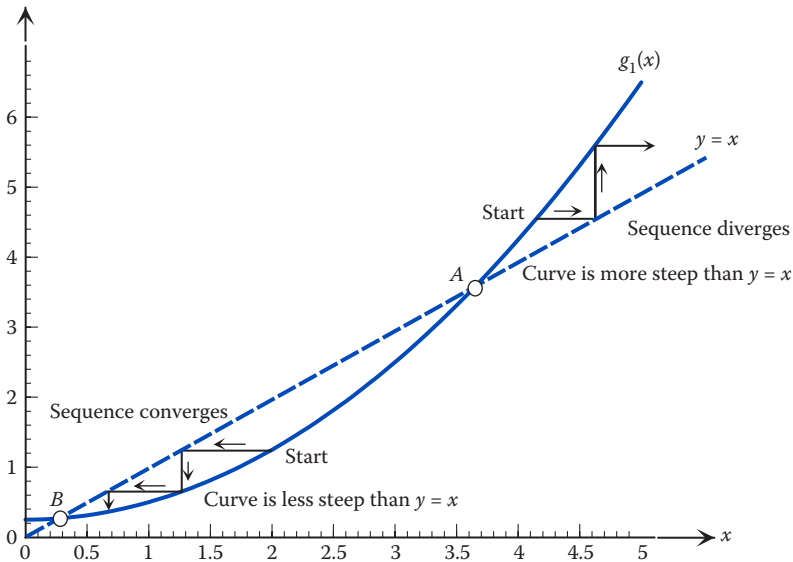


FIGURE 3.12
Fixed points of $g_1(x) = \frac{1}{4}(x^2 + 1)$.

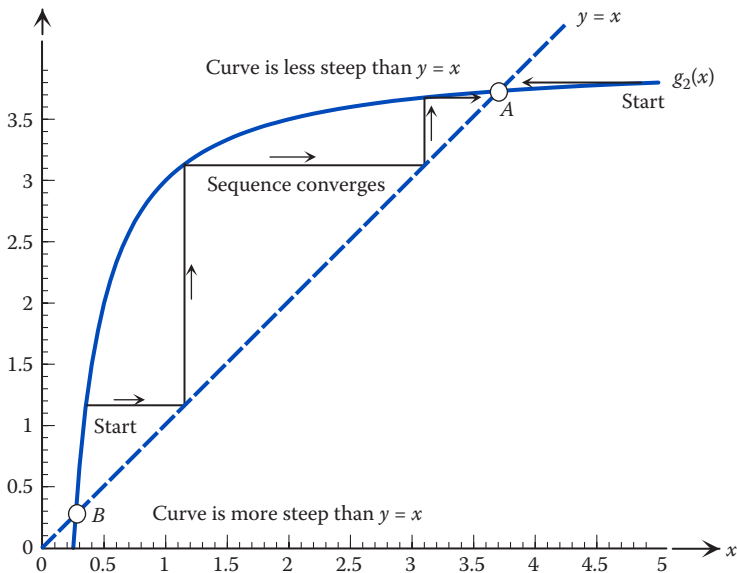


FIGURE 3.13
Fixed points of $g_2(x) = 4 - \frac{1}{x}$.

Figure 3.13 shows that the fixed point A certainly falls in the interval $x > 1$. However, we see that our starting choice of around $x_1 = 0.3$ led to convergence, even though it was not inside the required interval. This of course is due to the fact that the condition of convergence is only sufficient and not necessary.

```

>> g1 = @(x)((x^2+1)/4); g2 = @(x)(4-1/x);
>> [r1, n] = FixedPoint(g1, 2)

r1 =

    0.2680    % First root found by using iteration function g1

n =

     8

>> [r2, n] = FixedPoint(g2, 0.3)

r2 =

    3.7320    % Second root found by using iteration function g2

n =

     7

```

The original equation is $x^2 - 4x + 1 = 0$ so that we are looking for the roots of a polynomial. MATLAB has a built-in function `roots`, which performs this task:

```

>> roots([1 -4 1])

ans =

    3.7321
    0.2679

```

Of course, the approximate values returned by the `FixedPoint` function may be improved, and closer to those returned by `roots`, if a smaller tolerance than the default $1e-4$ is employed.

3.4.3 Rate of Convergence of the Fixed-Point Iteration

Suppose r is a fixed point of $g(x)$, and that $g(x)$ satisfies the hypotheses of Theorem 3.1 in some interval I . Also assume the $(k+1)$ th derivative of $g(x)$ is continuous in I . Expanding $g(x)$ in a Taylor's series about $x = r$, and noting that $r = g(r)$, $x_{n+1} = g(x_n)$, and $e_n = x_n - r$, the error at the $(n+1)$ th iteration is obtained as

$$\begin{aligned}
 e_{n+1} &= x_{n+1} - r = g(x_n) - g(r) \\
 &= g'(r)e_n + \frac{g''(r)}{2!}e_n^2 + \dots + \frac{g^{(k)}(r)}{k!}e_n^k + E_{k,n}
 \end{aligned} \tag{3.6}$$

where $E_{k,n}$, the error due to truncation, is described by

$$E_{k,n} = \frac{g^{(k+1)}(\xi_n)}{(k+1)!}e_n^{k+1} \quad \text{for some } \xi_n \in (r, x_n)$$

Assume $g'(x) \neq 0 \forall x \in I$. Then, for $k=0$, Equation 3.6 yields $e_{n+1} = g'(\xi_n)e_n$. But since $x_n \rightarrow r$ as $n \rightarrow \infty$ (by Theorem 3.1), we have $\xi_n \rightarrow r$ as well. Consequently,

$$\lim_{n \rightarrow \infty} \frac{e_{n+1}}{e_n} = \lim_{n \rightarrow \infty} g'(\xi_n) = g'(r) \neq 0$$

Therefore, convergence is linear. The rate of convergence will be improved if $g'(r) = 0$ and $g''(x) \neq 0 \forall x \in I$. In that case, it can be shown that

$$\lim_{n \rightarrow \infty} \frac{e_{n+1}}{e_n^2} = \frac{g''(r)}{2!} \neq 0$$

so that convergence is quadratic. We will see shortly that Newton's method falls in this category. From the foregoing analysis it is evident that the more derivatives of $g(x)$ vanish at the root, the faster the rate of the fixed-point iteration.

3.5 Newton's Method (Newton–Raphson Method)

Newton's method is the most commonly used open method to solve $f(x) = 0$, where f' is continuous. Consider the graph of $f(x)$ in [Figure 3.14](#). Start with an initial point x_1 and locate the point $(x_1, f(x_1))$ on the curve. Draw the tangent line to the curve at that point, and let its x -intercept be x_2 . The equation of this tangent line is

$$y - f(x_1) = f'(x_1)(x - x_1)$$

Therefore, its x -intercept is found by setting $y = 0$ and solving for x :

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

Once x_2 is available, locate the point $(x_2, f(x_2))$, draw the tangent line to the curve at that point, and let x_3 be its x -intercept, which is found as

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$$

Continue this process until the sequence $\{x_n\}$ converges to the intended root r . In general, two consecutive elements x_n and x_{n+1} are related via

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 1, 2, 3, \dots, \quad x_1 = \text{initial point} \quad (3.7)$$

The user-defined function `Newton` uses an initial x_1 and generates a sequence of elements $\{x_n\}$ via Equation 3.7 that eventually converges to the root of $f(x) = 0$. The function accepts $f(x)$ symbolically as an input so that it can calculate $f'(x)$ using the `diff` command; see

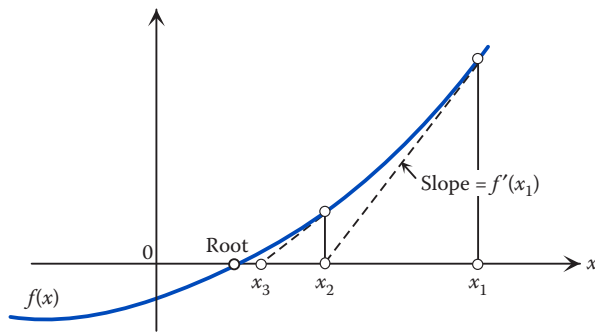


FIGURE 3.14

Geometry of Newton's method.

Chapter 2. Both f and f' are subsequently converted to MATLAB functions for evaluation purposes. The iterations stop when two consecutive generated elements are sufficiently close to one another, that is, $|x_{n+1} - x_n| < \epsilon$, where ϵ is a prescribed tolerance. The outputs are the approximate value of the root and the number of iterations needed to meet the tolerance.

```
function [r, n] = Newton(f, x1, tol, N)
%
% Newton uses Newton's method to approximate a root of f(x) = 0.
%
% [r, n] = Newton(f, x1, tol, N), where
%
% f is a symbolic function representing f(x),
% x1 is the initial point,
% tol is the scalar tolerance for convergence (default 1e-4),
% N is the maximum number of iterations (default 20),
%
% r is the approximate root of f(x) = 0,
% n is the number of iterations required for convergence.
%
if nargin < 4 || isempty(N), N = 20; end
if nargin < 3 || isempty(tol), tol = 1e-4; end
% Find f' and convert to MATLAB function for evaluation
fp = matlabFunction(diff(f));
% Convert f to MATLAB function for evaluation
f = matlabFunction(f);
x = zeros(1, N+1); % Pre-allocate
x(1) = x1;
for n = 1:N,
    if fp(x(n)) == 0,
        r = 'failure';
        return
    end
    x(n+1) = x(n) - f(x(n))/fp(x(n));
    if abs(x(n+1) - x(n)) < tol,
        r = x(n+1);
        return
    end
end
end
```

EXAMPLE 3.5: NEWTON'S METHOD

Consider $x \cos x + 1 = 0$ of Examples 3.1 and 3.2.

1. Using Newton's method with $x_1 = 1$, calculate x_2 and x_3 of the sequence that eventually converges to the root.
2. Find the root by executing the user-defined function `Newton` with $\epsilon = 10^{-4}$ and maximum 20 iterations.

Solution

1. Since $f(x) = x \cos x + 1$, we find $f'(x) = \cos x - x \sin x$. To calculate x_2 we apply Equation 3.7 with $n = 1$, that is,

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 1 - \frac{f(1)}{f'(1)} = 6.1144$$

Applying Equation 3.7 with $n = 2$,

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} = 2.6230$$

2. We will execute `Newton` with initial point `x1=1` while omitting the next two variables since they assume their default values in this example.

```
>> f = sym('x*cos(x)+1');  
>> [r, n] = Newton(f, 1)           % Default values for tol and N  
  
r =  
  
    2.0739  
  
n =  
  
    6
```

The result agrees to at least four decimal places with the highly accurate estimate returned by `fzero` earlier. Recall that for a bracketing method such as bisection, a similar accuracy was achieved by using a tolerance of 10^{-8} and performing 30 iterations.

EXAMPLE 3.6: NEWTON'S METHOD

Find the roots of $8x^3 - 18x^2 + x + 6 = 0$ using Newton's method with $\epsilon = 10^{-4}$ and maximum 20 iterations.

Solution

We first plot $f(x) = 8x^3 - 18x^2 + x + 6$ using the `ezplot` function to find approximate locations of its roots. The default range for `ezplot` is $[-2\pi, 2\pi]$, but by inspection the range is narrowed down to $[-1, 2.5]$.

```
>> f = sym('8*x^3-18*x^2+x+6');  
>> ezplot(f, [-1, 2.5])           % Figure 3.15
```

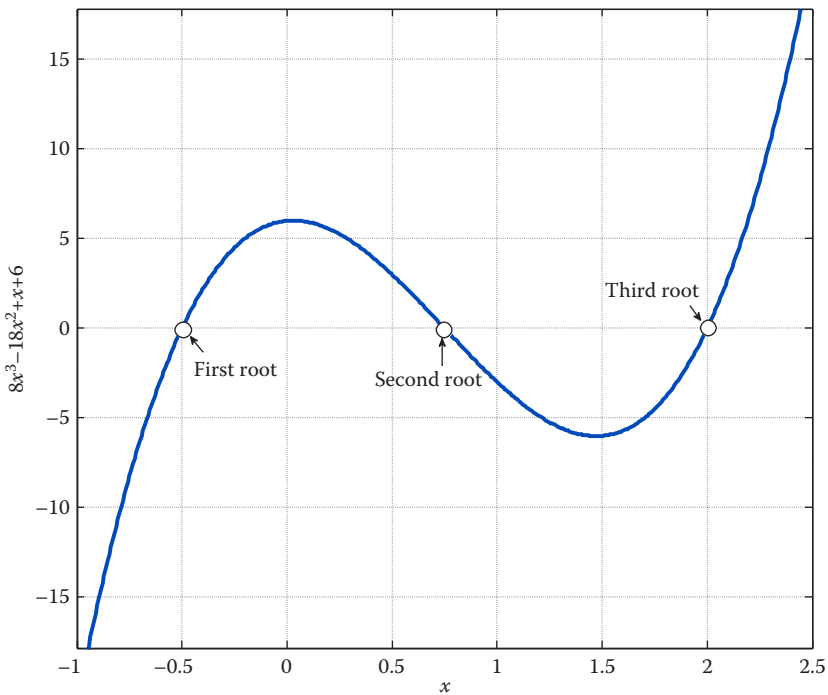


FIGURE 3.15

Location of the three roots of $8x^3 - 18x^2 + x + 6 = 0$.

Inspired by [Figure 3.15](#), we will execute the user-defined function `Newton` on three separate times, once with initial point $x_1 = -1$, a second time with $x_1 = 0.5$, and a third time with $x_1 = 1.5$.

```
>> [r1, n1] = Newton(f, -1)           % f was defined (symbolically) earlier
r1 =
    -0.5000           % First root
n1 =
     5
>> [r2, n2] = Newton(f, 0.5)
r2 =
     0.7500           % Second root
n2 =
     3
>> [r3, n3] = Newton(f, 1.5)
```

```

r3 =
    2.0000    % Third root
n3 =
    10

```

Since $f(x)$ is a polynomial here, the built-in MATLAB function `roots` can be used to find its roots.

```

>> roots([8 -18 1 6])
ans =
    2.0000
    0.7500
   -0.5000

```

3.5.1 Rate of Convergence of Newton's Method

It turns out that the speed of convergence of Newton's method depends on the multiplicity of an intended root of $f(x) = 0$. We say that a root r of $f(x) = 0$ is of multiplicity (or order) m if and only if

$$f(r) = 0, f'(r) = 0, f''(r) = 0, \dots, f^{(m-1)}(r) = 0, f^{(m)}(r) \neq 0$$

A root of order 1 is commonly known as a simple root.

Theorem 3.2: Rate of Convergence of Newton's Method

Let r be a root of $f(x) = 0$, and in Newton's iteration, Equation 3.7, let x_1 be sufficiently close to r .

a. If $f''(x)$ is continuous and r is a simple root, then

$$|e_{n+1}| \cong \frac{1}{2} \frac{|f''(r)|}{|f'(r)|} |e_n|^2 \Rightarrow \lim_{n \rightarrow \infty} \frac{|e_{n+1}|}{|e_n|^2} = \frac{1}{2} \frac{|f''(r)|}{|f'(r)|} \neq 0 \quad (3.8)$$

and convergence $\{x_n\} \rightarrow r$ is quadratic.

b. If $\{x_n\} \rightarrow r$, where r is root of order $m > 1$, then

$$|e_{n+1}| \cong \frac{m-1}{m} |e_n| \Rightarrow \lim_{n \rightarrow \infty} \frac{|e_{n+1}|}{|e_n|} = \frac{m-1}{m} \neq 0 \quad (3.9)$$

and convergence is linear. ■

EXAMPLE 3.7: QUADRATIC CONVERGENCE; NEWTON'S METHOD

Suppose Newton's method is employed to find the two roots of $x^2 - 3x - 4 = 0$, which are $r = -1, 4$. Let us focus on the task of finding the larger root, $r = 4$. Since the root is simple, by Equation 3.8 we have

$$\lim_{n \rightarrow \infty} \frac{|e_{n+1}|}{|e_n|^2} \cong \frac{1}{2} \frac{|f''(4)|}{|f'(4)|} = \frac{1}{2} \left(\frac{2}{5} \right) = 0.2$$

This indicates that convergence is quadratic, as stated in Theorem 3.2. While finding the smaller root $r = -1$, this limit is once again 0.2, thus confirming quadratic convergence again. The reader can readily verify this by tracking the ratio $|e_{n+1}|/|e_n|^2$ while running Newton's method.

3.5.2 A Few Notes on Newton's Method

- When Newton's method works, it generates a sequence that converges rapidly to the intended root.
- Several factors may cause Newton's method to fail. A usual factor is that the initial point x_1 is not sufficiently close to the intended root. Another one is that at some point in the iterations, $f'(x_n)$ may be close to or equal to zero. Other scenarios, where the iteration simply halts or the sequence diverges, are shown in Figure 3.16 and explained in Example 3.8.
- If $f(x)$, $f'(x)$, and $f''(x)$ are continuous, $f'(\text{root}) \neq 0$, and the initial point x_1 is close to the root, then the sequence generated by Newton's method converges to the root.
- A downside of Newton's method is that it requires the calculation of $f'(x)$, which at times may be difficult. In these cases, the secant method (described below in Section 3.6) can be used instead.

EXAMPLE 3.8: NEWTON'S METHOD

Apply Newton's method to find the root of $2/(x+1) = 1$. For the initial point use (1) $x_1 = 3$, and (2) $x_1 = 4$.

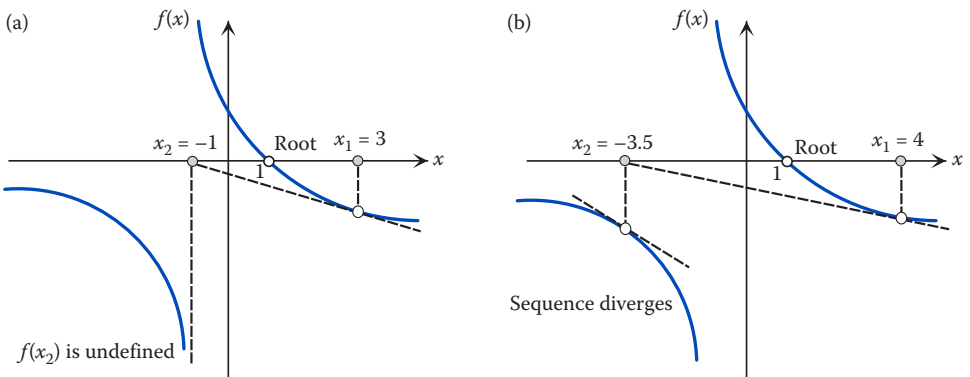


FIGURE 3.16

Two cases where Newton's method fails: (a) sequence halts, and (b) sequence diverges.

Solution

$$f(x) = \frac{2}{x+1} - 1 \text{ so that } f'(x) = -\frac{2}{(x+1)^2}.$$

1. Starting with $x_1 = 3$, we find

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 3 - \frac{-\frac{1}{2}}{-\frac{1}{8}} = -1$$

The iterations halt at this point because $f(-1)$ is undefined. This is illustrated in [Figure 3.16a](#).

2. Starting with $x_1 = 4$, we find

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 4 - \frac{-\frac{3}{5}}{-\frac{2}{25}} = -3.5, \quad x_3 = -9.1250, \quad x_4 = -50.2578, \quad \dots$$

The sequence clearly diverges. This is illustrated in [Figure 3.16b](#).

3.5.3 Modified Newton's Method for Roots with Multiplicity 2 or Higher

If r is a root of $f(x)$ and r has a multiplicity 2 or higher, then convergence of the sequence generated by Newton's method is linear; see Theorem 3.2. In these situations, Newton's method may be modified to improve the speed of convergence. The modified Newton's method designed for roots of multiplicity 2 or higher is described as

$$x_{n+1} = x_n - \frac{f(x_n)f'(x_n)}{[f'(x_n)]^2 - f(x_n)f''(x_n)}, \quad n = 1, 2, 3, \dots, \quad x_1 = \text{initial point} \quad (3.10)$$

The user-defined function `NewtonMod` uses an initial x_1 and generates a sequence of elements $\{x_n\}$ via Equation 3.10 that eventually converges to the root of $f(x) = 0$, where the root has multiplicity 2 or higher. The iterations stop when two consecutive elements are sufficiently close to one another, that is, $|x_{n+1} - x_n| < \epsilon$, where ϵ is the prescribed tolerance. The outputs are the approximate value of the root and the number of iterations needed to meet the tolerance.

```
function [r, n] = NewtonMod(f, x1, tol, N)
%
% NewtonMod uses modified Newton's method to approximate a root (with
% multiplicity 2 or higher) of f(x) = 0.
%
% [r, n] = NewtonMod(f, x1, tol, N), where
%
% f is a symbolic function representing f(x),
% x1 is the initial point,
% tol is the scalar tolerance for convergence (default 1e-4),
% N is the maximum number of iterations (default 20),
%
% r is the approximate root of f(x) = 0,
% n is the number of iterations required for convergence.
%
```

```

if nargin < 4 || isempty(N), N = 20; end
if nargin < 3 || isempty(tol), tol = 1e-4; end
% Find f', f'' and convert to MATLAB functions for evaluation
fp = matlabFunction(diff(f));
f2p = matlabFunction(diff(f,2));
% Convert f to MATLAB function for evaluation
f = matlabFunction(f);
x = zeros(1,N+1);      % Pre-allocate
x(1) = x1;
for n = 1:N,
    x(n+1) = x(n) - (f(x(n))*fp(x(n)))/(fp(x(n))^2-f(x(n))*f2p(x(n)));
    if abs(x(n+1)-x(n)) < tol,
        r = x(n+1);
        return
    end
end
end

```

EXAMPLE 3.9: MODIFIED NEWTON'S METHOD

Find the roots of $4x^3 + 4x^2 - 7x + 2 = 0$ using Newton's method and modified Newton's method. Discuss the results.

Solution

Figure 3.17 reveals that $f(x) = 4x^3 + 4x^2 - 7x + 2$ has a simple root at -2 and a double root (multiplicity 2) at 0.5 since it is tangent to the x -axis at that point. We will execute `NewtonMod` with default parameter values and $x_1 = 0$.

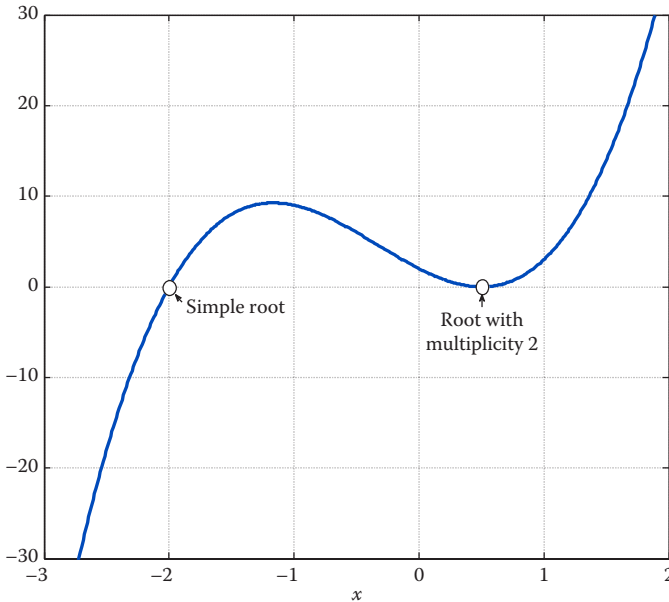


FIGURE 3.17

A simple and a double root of $4x^3 + 4x^2 - 7x + 2 = 0$.

```
>> f = sym('4*x^3+4*x^2-7*x+2');
>> ezplot(f,[-3,2]) % Figure 3.17
>> [r, n] = NewtonMod(f, 0)
```

```
r =
    0.5000
```

```
n =
    4
```

Executing Newton with default parameters and $x_1 = 0$ yields

```
>> [r, n] = Newton(f, 0)
```

```
r =
    0.4999
```

```
n =
    12
```

The modified Newton's method is clearly superior to the standard Newton's method when approximating a multiple root. The same, however, is not true for simple roots. Applying both methods with $x_1 = -3$ and default parameters yields

```
>> [r, n] = NewtonMod(f, -3)
```

```
r =
   -2.0000
```

```
n =
    6
```

```
>> [r, n] = Newton(f, -3)
```

```
r =
   -2.0000
```

```
n =
    5
```

The standard Newton's method generally exhibits a faster convergence (quadratic, by Theorem 3.2) to the simple root. The modified Newton's method outperforms the standard one when finding multiple roots.

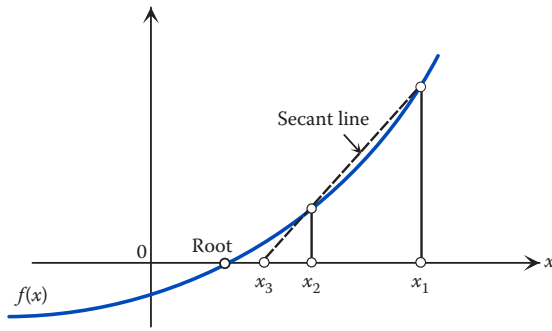


FIGURE 3.18
Geometry of secant method.

3.6 Secant Method

The secant method is another open method to solve $f(x) = 0$. Consider the graph of $f(x)$ in [Figure 3.18](#). Start with two initial points x_1 and x_2 , locate the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ on the curve, and draw the secant line connecting them. The x -intercept of this secant line is x_3 . Next, use x_2 and x_3 to define a secant line and let the x -intercept of this line be x_4 . Continue the process until the sequence $\{x_n\}$ converges to the root. In general, two consecutive elements x_n and x_{n+1} generated by secant method are related via

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n), \quad n = 2, 3, 4, \dots, \quad x_1, x_2 = \text{initial points} \quad (3.11)$$

Comparing with Newton's method, we see that $f'(x_n)$ in Equation 3.7 is essentially approximated by, and replaced with, the difference quotient

$$\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

The user-defined function `Secant` uses initial points x_1 and x_2 and generates a sequence of elements $\{x_n\}$ that eventually converges to the root of $f(x) = 0$. The function accepts $f(x)$ symbolically as an input and converts it to a MATLAB function for evaluation purposes. The iterations stop when two consecutive elements are sufficiently close to one another, that is, $|x_{n+1} - x_n| < \epsilon$, where ϵ is the prescribed tolerance. The outputs are the approximate value of the root and the number of iterations needed to meet the tolerance.

```
function [r, n] = Secant(f, x1, x2, tol, N)
%
% Secant uses secant method to approximate roots of f(x) = 0.
%
% [r, n] = Secant(f, x1, x2, tol, N), where
%
% f is a symbolic function representing f(x),
% x1 and x2 are the initial values of x,
```

```

% tol is the scalar tolerance of convergence (default 1e-4),
% N is the maximum number of iterations (default 20),
%
% r is the approximate root of f(x) = 0,
% n is the number of iterations required for convergence.
%
if nargin < 5 || isempty(N), N = 20; end
if nargin < 4 || isempty(tol), tol = 1e-4; end
f = matlabFunction(f);
x = zeros(1, N+1); % Pre-allocate
for n = 2:N,
    if x1 == x2,
        r='failure';
        return
    end
    x(1) = x1; x(2) = x2;
    x(n+1) = x(n) - ((x(n)-x(n-1))/(f(x(n))-f(x(n-1))))*f(x(n));
    if abs(x(n+1)-x(n)) < tol,
        r = x(n+1);
        return
    end
end
end

```

EXAMPLE 3.10: SECANT METHOD

Consider $x \cos x + 1 = 0$.

1. Using secant method with $x_1 = 1$ and $x_2 = 1.5$, calculate x_3 and x_4 of the sequence that eventually converges to the root
2. Find the root by executing the user-defined function Secant with $\varepsilon = 10^{-4}$ and maximum 20 iterations

Solution

1. Let $f(x) = x \cos x + 1$. To calculate x_3 we apply Equation 3.11 with $n = 2$, that is,

$$x_3 = x_2 - \frac{x_2 - x_1}{f(x_2) - f(x_1)} f(x_2) = 1.5 - \frac{1.5 - 1}{f(1.5) - f(1)} f(1.5) = 2.7737$$

Applying Equation 3.11 with $n = 3$,

$$x_4 = x_3 - \frac{x_3 - x_2}{f(x_3) - f(x_2)} f(x_3) = 2.0229$$

- 2.

```

>> f = sym('x*cos(x)+1');
>> [r, n] = Secant(f, 1, 1.5)

```

```

r =
    2.0739

```

```

n =
    6

```

Recall that Newton's method starting with $x_1 = 1$ also required six (6) iterations. Therefore, for this particular problem at least, the secant and Newton's methods have similar rates of convergence.

3.6.1 Rate of Convergence of Secant Method

Assuming a simple root r , the rate of convergence of secant method is $\frac{1}{2}(1 + \sqrt{5}) \cong 1.618$. More exactly,

$$\lim_{n \rightarrow \infty} \frac{|e_{n+1}|}{|e_n|^{1.618}} = \left| \frac{f''(r)}{2f'(r)} \right|^{0.618} \neq 0 \quad (3.12)$$

3.6.2 A Few Notes on Secant Method

- The sequence generated by the secant method is not guaranteed to converge to the intended root because the root is not bracketed in each step.
- For the case of a simple root, the rate of convergence for the secant method is 1.618, thus the generated sequence converges faster than the linear but slower than the quadratic. Therefore, it is slower than Newton's method—which has quadratic convergence for simple root—but $f'(x)$ does not need to be calculated.
- If $f(x)$, $f'(x)$, and $f''(x)$ are continuous on an interval I , which contains the root, $f'(\text{root}) \neq 0$, and the initial points x_1 and x_2 are close to the root, then the secant method converges to the root.

3.7 Equations with Several Roots

The bracketing and open methods presented in this chapter are capable of finding as many roots of $f(x) = 0$ as desired, but they can only achieve this by finding one root at a time. And because in many applications several roots of an equation are sought, this approach proves to be quite ineffective. In what follows we will present two practical options to find several roots of $f(x) = 0$.

3.7.1 Finding Roots to the Right of a Specified Point

The user-defined function `Nzeros` finds n roots of function f to the right of the specified initial point x_0 by starting at x_0 and incrementing x by Δx and inspecting the sign of the corresponding f . A root is identified when $|\Delta x/x| < \epsilon$, where ϵ is a prescribed tolerance. The output is the list of the desired number of approximate roots.

```
function Nroots = Nzeros(f, n, x0, tol, delx)
%
% Nzeros approximates a desired number of roots of f(x) on the right of
% a specified point.
%
% Nroots = Nzeros(f, n, x0, tol, delx), where
```

```

%
%     f is an anonymous function representing f(x),
%     n is the number of desired roots,
%     x0 is the starting value,
%     tol is the scalar tolerance (default is 1e-6),
%     delx is the increment in x (default is 0.1),
%
%     Nroots is the list of n roots of f(x) to the right of x0.
%
if nargin < 5 || isempty(delx), delx = 0.1; end
if nargin < 4 || isempty(tol), tol = 1e-6; end
x = x0; dx = delx;
Nroots = zeros(n,1);      % Pre-allocate
for i = 1:n,
%     sgn1 = sign((f(x)));
    while abs(dx/x) > tol,
        if sign((f(x))) ~= sign((f(x+dx))),
            dx = dx/2;
        else
            x = x + dx;
        end
    end
    Nroots(i) = x; dx = delx;
    x = x + abs(0.05*x);
end

```

EXAMPLE 3.11: ROOTS TO THE RIGHT OF A POINT

Find the first three positive roots of $x \cos x + 1 = 0$.

Solution

We will execute the user-defined function `Nzeros` with $x_0 = 0$ (to find positive roots) and default values for `tol` and `delx`.

```

>> f = @(x)(x*cos(x)+1);
>> Nroots = Nzeros(f, 3, 0)

Nroots =

    2.0739
    4.4877
    7.9796

```

Figure 3.19 clearly confirms the numerical values returned by `Nzeros`.

3.7.2 Finding Several Roots in an Interval Using `fzero`

Discretize f over the given interval, identify several subintervals where the function f experiences sign changes, and subsequently apply the built-in MATLAB function `fzero` to find a root in each identified interval. The following example will demonstrate the details of this approach.

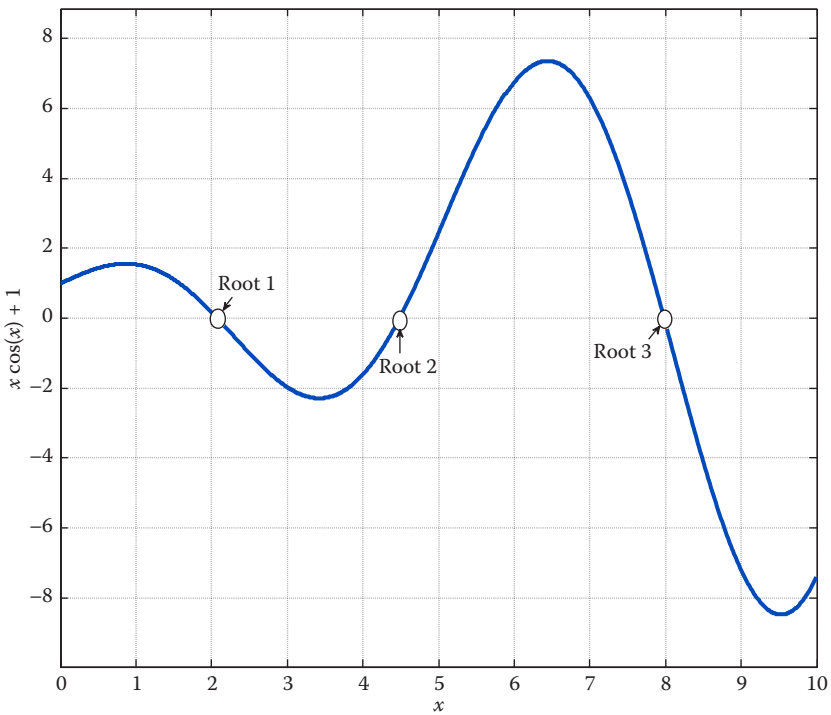


FIGURE 3.19

The first three positive roots of $x \cos x + 1 = 0$.

EXAMPLE 3.12: SEVERAL ROOTS

Find all roots of $x \sin x = 0$ in $[-10, 10]$.

Solution

There are several roots in the given range, and each root must be found individually using an appropriate initial guess. These initial guesses can be generated by evaluating the function at a few points in the given range, and identifying any sign changes.

```
>> fun = @(x)(x.*sin(x));
>> x = linspace(-10,10,20); % Generate 20 points in the given range
>> f = fun(x); % Evaluate function at the selected points
>> plot(x,f) % Figure 3.20
```

The vector f has the following 20 components:

$f =$

-5.4402	4.1111	7.8882	3.6282	-2.7436	-4.7354	-1.9025	1.2847
1.5789	0.2644	0.2644	1.5789	1.2847	-1.9025	-4.7354	-2.7436
3.6282	7.8882	4.1111	-5.4402				

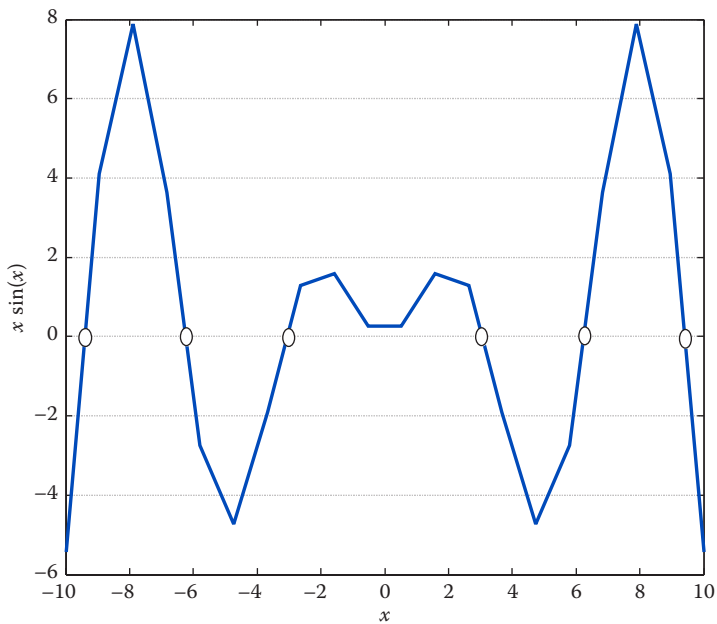


FIGURE 3.20

Several roots of $x \sin x = 0$ in $[-10, 10]$.

Any sign changes in f can be identified by

```
>> I = find(sign(f(2:end)) ~= sign(f(1:end-1)))
```

```
I =
```

```
     1     4     7    13    16    19
```

These values refer to the locations of the elements of f representing sign changes, that is, the boxed entries shown above. To find the first root, we use `fzero` with the two-element initial guess $x([I(1) \ I(1)+1])$, which in this case translates to

```
>> x([1 2])
```

```
ans =
```

```
-10.0000    -8.9474
```

This means the first root will be located in the interval $[-10.0000, -8.9474]$, and is found as

```
>> r(1) = fzero(fun, x([1 2]))
```

```
r =
```

```
-9.4248    % First root
```

The next root will be found using `fzero` with the two-element initial guess `x([I(2) I(2)+1])`, which is

```
>> x([4 5])
ans =
    -6.8421    -5.7895
```

Therefore, the second root is in the interval `[-6.8421, -5.7895]`, found via

```
>> r(2) = fzero(fun, x([4 5]))
r =
    -6.2832    % Second root
```

This process continues until all roots have been identified.

```
>> for n = 1:length(I),
    r(n) = fzero(fun, x([I(n) I(n)+1]));    % Approximate roots
>> end

>> disp(r)    % Display all roots
r =
    -9.4248    -6.2832    -3.1416     3.1416     6.2832     9.4248
```

These six roots agree with those in [Figure 3.20](#). However, the equation $x \sin x = 0$ has an obvious root at $x = 0$ which has not been identified here. For a better understanding of the situation, we plot our function $x \sin x$ using 100 points:

```
>> x = linspace(-10,10); f = fun(x);
>> plot(x, f)    % Figure 3.21
```

It is then clear that $x = 0$ is a point of tangency, hence no sign changes experienced by f on its two sides. This explains why this root was missed, as `fzero` only finds roots where the function changes sign.

EXAMPLE 3.13: POINTS OF DISCONTINUITY

Find all roots of $\tan x = \tanh x$ in `[-2, 2]`.

Solution

Following the strategy employed in [Example 3.12](#), we execute the script below to identify the roots:

```
>> fun = @(x)(tan(x)-tanh(x));
>> ezplot(fun, [-2,2])    % Figure 3.22
>> x = linspace(-2,2);
>> f = fun(x);
>> I = find(sign(f(2:end)) ~= sign(f(1:end-1)));
```

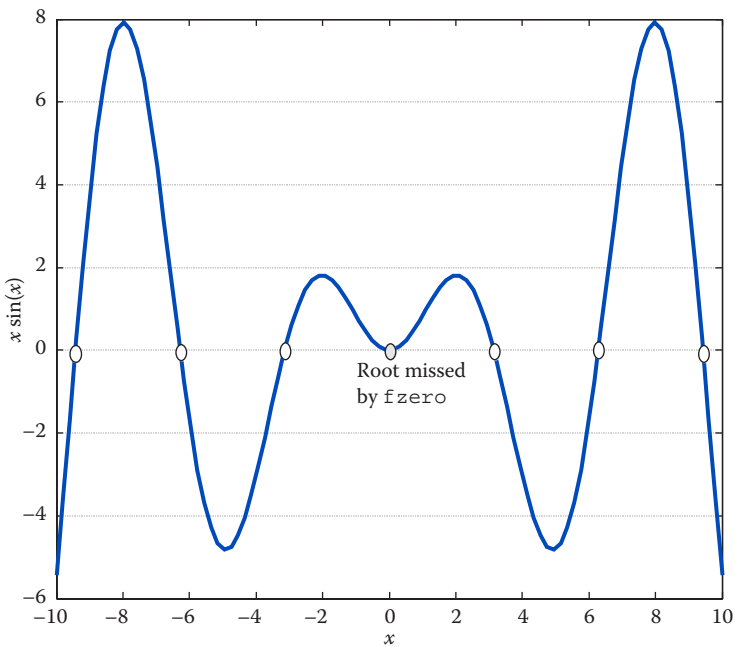


FIGURE 3.21

All roots of $x \sin x = 0$ in $[-10, 10]$.

```
>> for n = 1:length(I)
    r(n) = fzero(fun, x([I(n) I(n)+1]));
end

>> r

r =

    -1.5708    -0.0000    1.5708
```

Figure 3.22 shows the only legitimate root to be at 0, while the other two are merely points of discontinuity. Obviously, the two erroneous values are returned by the script because the function experience sign changes at the points of discontinuity.

PROBLEM SET (CHAPTER 3)

Bisection Method (Section 3.2)

In Problems 1 through 6, the given equation has a root in the indicated interval.

- ✎ Using the bisection method, generate the first three midpoints and intervals (in addition to the one given).
- 🖱 Find the root estimate by executing the user-defined function `Bisection` with default values for `kmax` and `tol`.

1. $x^2 - 4x + 2 = 0$, $[3, 4]$

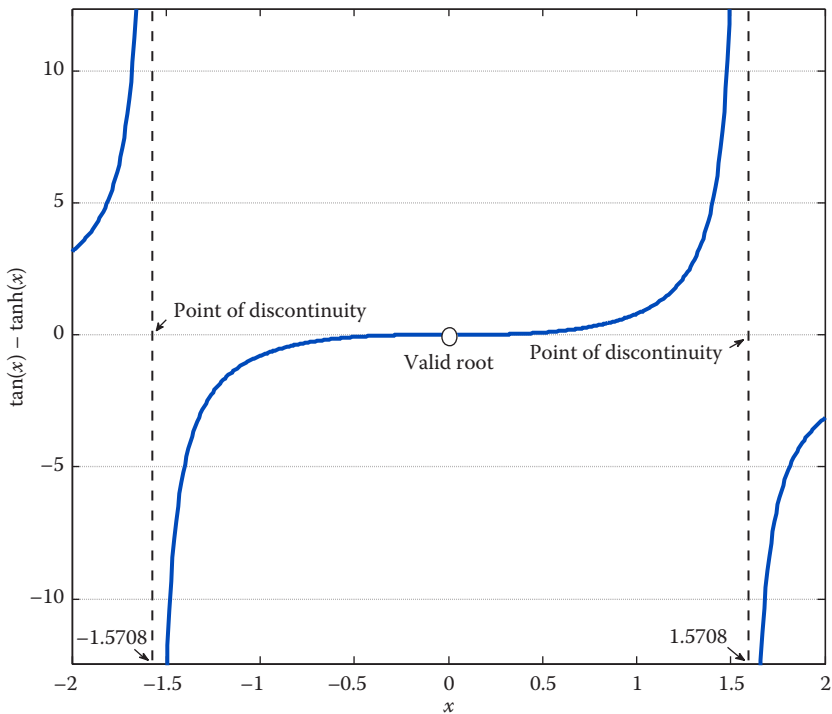









FIGURE 3.22

Discontinuity points mistaken for roots.


2. $\cos 2x + \frac{2}{3} \sin x = 0$, $[0, 2]$
3. $e^{x/3} - 3x = 2$, $[-2, 0]$
4. $1 + \cos x \cosh x = 0$, $[-5, -4]$
5. $\frac{1}{3}x + \ln x = 1$, $[1, 2]$
6. $\tan(0.4x) + x = -1$, $[-2, 0]$
7.  Modify the user-defined function `Bisection` so that the table is not generated and the outputs are the approximate root and the number of iterations needed for the tolerance to be met. All other parameters, including default values, are to remain as in `Bisection`. Save this function as `Bisection_New`. Apply `Bisection_New` to the following problem: $e^x + \cos x = 3$, $[0, 1]$.
8.  Apply the user-defined function `Bisection_New` (Problem 7) to find the root in the indicated interval: $3^{1-x} = 2x^3$, $[0, 2]$.
9.  The goal is to find all roots of $\sin x \sinh x = \frac{3}{2}$ in $[-2, 2]$ using the bisection method, as follows: First, locate the roots graphically and identify the intervals containing the roots. The endpoints of each interval must be chosen as the integers closest to the root on each side of that root. Then apply the user-defined function `Bisection_New` (Problem 7) with default values for `tol` and `kmax` to find one root at a time.
10.  Repeat Problem 9 for the roots of $e^{-x/3} \sin x = 0$ in $[1, 10]$.

Regula Falsi Method (Section 3.3)



In Problems 11 through 16, the given equation has a root in the indicated interval.


- a.  Using the regula falsi method, find the first three elements in the sequence that eventually converges to the root.
 - b.  Find the root by executing the user-defined function `RegulaFalsi` with `kmax=20` and `tol=1e-3`.
11. $e^{-2x/3} + \ln\left(\frac{1}{2}x\right) = 0$, $[1, 2]$
 12. $\cos x \cosh x = 1$, $[4, 5]$
 13. $\cos x + \cos 2x = 1$, $[0, 2]$
 14. $x^3 - 5x + 3 = 0$, $[1, 2]$
 15. $e^{-x} = x^2$, $[0, 2]$
 16. $x^2 + e^{x/2} = 5$, $[1, 2]$
 17.  The goal is to find all roots of $\sin\left(\frac{1}{2}x\right) + \frac{1}{2}\ln x = 1$ in $[5, 20]$ using the regula falsi method, as follows: First, locate the roots graphically and identify the intervals containing the roots. The endpoints of each interval must be chosen as the integers closest to the root on each side of that root. Then apply the user-defined function `RegulaFalsi` (but suppress the table) with default values for `tol` and `kmax` to find one root at a time.

Modified Regula Falsi

18.  Modify the user-defined function `RegulaFalsi` so that if an endpoint remains stationary for three consecutive iterations, $\frac{1}{2}f(\text{endpoint})$ is used in the calculation of the next x -intercept, and if the endpoint still remains stationary for three consecutive iterations, $\frac{1}{4}f(\text{endpoint})$ is used, and so on. All other parameters, including the default values, and the terminating condition are to remain the same as in `RegulaFalsi`. Save this function as `RegulaFalsi_Mod`.
Apply `RegulaFalsi` to find a root of $\frac{1}{3}(x-2)^2 - 3 = 0$ in $[-6, 2]$. Next apply `RegulaFalsi_Mod` and compare the results.

Fixed-Point Method (Section 3.4)




19. The two roots of $x + 3^{-x} = 4$ are to be found by the fixed-point method as follows: Define two iteration functions $g_1(x) = 4 - 3^{-x}$ and $g_2(x) = -\log_3(4 - x)$.
 - a.  Locate the fixed points of $g_1(x)$ and $g_2(x)$ graphically.
 - b.  Referring to the figure showing the fixed points of g_1 , set x_1 to be the nearest integer to the left of the smaller fixed point and perform four iterations using the fixed-point method. Next, set x_1 to be the nearest integer to the right of the same fixed point and perform four iterations. If both fixed points were not found this way, repeat the process applied to g_2 . Discuss any convergence issues as related to Theorem 3.1.


20.  The two roots of $3x^2 + 2.72x - 1.24 = 0$ are to be found using the fixed-point method as follows: Define iteration functions

$$g_1(x) = \frac{-3x^2 + 1.24}{2.72}, \quad g_2(x) = \frac{-2.72x + 1.24}{3x}$$

- Locate the fixed points of $g_1(x)$ and $g_2(x)$ graphically.
 - Focus on g_1 first. Execute the user-defined function `FixedPoint` with initial point x_1 chosen as the nearest integer to the left of the smaller fixed point. Execute a second time with x_1 an integer between the two fixed points. Finally, with x_1 to the right of the larger fixed point. In all cases, use the default tolerance, but increase `kmax` if necessary. Discuss all convergence issues as related to Theorem 3.1.
 - Repeat Part (b), this time focusing on g_2 .
21. Consider the fixed-point iteration described by

$$x_{n+1} = g(x_n) = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right), \quad n = 1, 2, 3, \dots, \quad a > 0$$

-  Show that the iteration converges to \sqrt{a} for any initial point $x_1 > 0$, and that the convergence is quadratic.
 -  Apply this iteration function $g(x)$ to approximate $\sqrt{5}$. Execute the user-defined function `FixedPoint` using default values for `kmax` and `tol`, and x_1 chosen as the nearest integer on the left of the fixed point.
22.  The goal is to find the root of $0.3x^2 - x^{1/3} = 1.4$ using the fixed-point method.

- As a potential iteration function, select $g_1(x) = \frac{x^{1/3} + 1.4}{0.3x}$. Graphically locate the fixed point of $g_1(x)$. Execute the user-defined function `FixedPoint` twice, once with initial point chosen as the integer nearest the fixed point on its left, and a second time with the nearest integer on its right. Use default values for `kmax` and `tol`, but increase `kmax` if necessary. Fully discuss convergence issues as related to Theorem 3.1.
 - Next, as the iteration function select $g_2(x) = (0.3x^2 - 1.4)^3$ and repeat all steps in Part (a).
23.  The two roots of $x^2 - 3.13x + 2.0332 = 0$ are to be found using the fixed-point method as follows: Define iteration functions

$$g_1(x) = \frac{3.13x - 2.0332}{x}, \quad g_2(x) = \frac{x^2 + 2.0332}{3.13}$$

- Locate the fixed points of $g_1(x)$ and $g_2(x)$ graphically.
- Focus on g_1 first. Execute the user-defined function `FixedPoint` with initial point x_1 chosen as the nearest integer to the left of the smaller fixed point. Execute a second time with x_1 an integer between the two fixed points. Finally,



with x_1 to the right of the larger fixed point. In all cases, use the default tolerance, but increase k_{\max} if necessary. Discuss all convergence issues as related to Theorem 3.1.

- c. Repeat Part (b), this time focusing on g_2 .
24. The two roots of $2^{-x/3} + e^x = 2.2$ are to be found by the fixed-point method as follows: Define two iteration functions $g_1(x) = -3 \log_2(2.2 - e^x)$ and $g_2(x) = \ln(2.2 - 2^{-x/3})$.
- Locate the fixed points of $g_1(x)$ and $g_2(x)$ graphically.
 - Referring to the figure showing the fixed points of g_1 , choose x_1 to be the nearest integer to the left of the smaller fixed point and perform four iterations using the fixed-point method. Next, let x_1 be the nearest integer to the right of the same fixed point and perform four iterations. If both fixed points were not found, repeat the process with g_2 . Discuss any convergence issues as related to Theorem 3.1.

Newton's Method (Section 3.5)





In Problems 25 through 30, the given equation has a root in the indicated interval.

- Using Newton's method, with the initial point set to be the left end of the interval, generate the next four elements in the sequence that eventually converges to the root.
 - Find the root by executing the user-defined function `Newton` with $k_{\max}=20$ and $\text{tol}=1e-6$.
25. $x^3 + 2x^2 + x + 2 = 0$, $[-3, -1]$
26. $3x^2 - x - 4 = 0$, $[-2, 0]$
27. $\cos x = 2x - 1$, $[0, 2]$
28. $\ln(\frac{1}{3}x + 1) = 2x + 1$, $[-1, 0]$
29. $e^{-(x-1)} = 2.6 + \cos(x + 1)$, $[-1, 1]$
30. $\sin x \sinh x + 1 = 0$, $[3, 4]$
31. Determine graphically how many roots the equation $0.4x^3 - x^{3/2} = 1.3$ has. Then find each root by executing the user-defined function `Newton` with default parameter values and x_1 chosen as the closest integer on the left of the root.
32. The goal is to find two roots of $\cos x \cosh x = -1.3$ in $[-4, 4]$.
- Graphically locate the roots.
 - To approximate each root, execute the user-defined function `Newton` with default parameter values and x_1 chosen as the closest integer on the left of the root. If the intended root is not found this way, set x_1 to be the integer closest to the root on its right and re-execute `Newton`. Discuss the results.
33. All three roots of the equation $x^3 - 0.8x^2 - 1.12x - 0.2560 = 0$ lie inside the interval $[-2, 2]$.
- Graphically locate the roots, and decide whether each root is simple or of higher multiplicity.


- b. Approximate the root with higher multiplicity by executing the user-defined function `NewtonMod` and the simple root by executing `Newton`. In both cases use default parameter values, and x_1 chosen as the closest integer on the left of the root.
34.  Roots of $x^3 - 0.9x^2 + 0.27x - 0.027 = 0$ lie inside $[-1, 1]$.
- Graphically locate the roots, and determine if a root is simple or of higher multiplicity.
 - Estimate the root with higher multiplicity by executing the user-defined function `NewtonMod`. Use default parameter values, and let x_1 be the closest integer on the left of the root.
35.  Locate the root(s) of $0.2[x - 3 \sin(x + 1)] = x^3$ graphically, and depending on multiplicity, use Newton's method or the modified Newton's method to find the root(s). Use default parameter values, and let x_1 be the closest integer on the left of the root. Verify the result by using the built-in `fzero` function.






Secant Method (Section 3.6)


In Problems 36 through 42,

-  Apply the secant method with the given initial points x_1 and x_2 to generate the next four elements in the sequence that eventually converges to the root.
 -  Estimate the root by executing the user-defined function `Secant` with `kmax=20` and `tol=1e-6`.
36. $x^3 - x^{1/4} = 3.45$, $x_1 = 4$, $x_2 = 3.5$
37. $x^3 + 2.7x + 2.6 = 0$, $x_1 = -2$, $x_2 = -1.8$
38. $e^{-x/2} + \ln(x + 2) = 2$, $x_1 = 0$, $x_2 = 1$
39. $\sin(x - 1) = \frac{3}{2}x$, $x_1 = 5$, $x_2 = 4$
40. $\sinh(0.6x - 1) - 1.3x + 3.2 = 0$, $x_1 = -5$, $x_2 = -4$
41. $\cosh\left(\frac{2}{5}x\right) = x$, $x_1 = -4$, $x_2 = -2$
42. $x\sqrt{x^2 + 250} = 450$, $x_1 = 10$, $x_2 = 12$
43.  Graphically locate the root of $10x^3 + 15x^2 + 6x + 9 = 0$. Find the root numerically by applying the user-defined function `Secant` [with $x_1 = 1.5$, $x_2 = 1$]. Next, apply the function `Newton` with $x_1 = 1.5$. In both cases, use default parameter values. Compare the results.
44.  Graphically locate the roots of $x^2 + x + 0.4x^{-1/3} = 1$. Find the roots numerically by applying the user-defined function `Secant` and properly selected initial points. Use default parameter values for tolerance and maximum number of iterations.

Equations with Several Roots (Section 3.7)

45.  Find the first five positive roots of $\sin x + (1/x^2)\cos 2x = 0$ and confirm the numerical results graphically.

46.  Using the user-defined function `Nzeros` find all roots of $\sin(\frac{1}{2}\pi x) = \frac{1}{3}$ in $[-4, 4]$.
47.  A very important function in engineering applications is the Bessel function of the first kind. The Bessel function of the first kind of order 0, denoted by $J_0(x)$, is represented in MATLAB by `besselj(0,x)`. The zeros of Bessel functions arise in applications such as vibration analysis of circular membranes. Find the first four positive zeros of $J_0(x)$, and verify them graphically.
48.  Find the first four positive zeros of the Bessel function of the first kind of order 1, denoted by $J_1(x)$, represented in MATLAB by `besselj(1,x)`, and verify them graphically.
49.  The natural frequencies of a beam are directly related to the roots of the frequency equation. For a beam fixed at both of its ends, the frequency equation is derived as $\cos x \cosh x = 1$. Find the first five positive roots of this frequency equation
- Using the user-defined function `Nzeros`.
 - By identifying the intervals where sign changes occur, followed by the application of `fzero` with two-element initial guesses.
50.  The natural frequencies of a beam are directly related to the roots of the frequency equation. For a beam fixed at its left end and pinned (hinged) at its right end, the frequency equation is derived as $\tan x = \tanh x$. Find the first three positive roots of this frequency equation
- Using the user-defined function `Nzeros`.
 - By identifying the intervals where sign changes occur, followed by the application of `fzero` with two-element initial guesses.

 In Problems 51 through 54 find all the roots of the polynomial equation by identifying the intervals where sign changes occur, followed by the application of `fzero` with two-element initial guesses. Verify the findings by using the MATLAB built-in function `roots`.

51. $0.2x^4 + 0.58x^3 - 12.1040x^2 + 20.3360x - 6.24 = 0$

52. $x^4 - 3.9x^3 + 1.2625x^2 + 1.4250x + 0.2125 = 0$

53. $8x^5 - 44x^4 + 86x^3 - 73x^2 + 28x - 4 = 0$

54. $4x^4 + 15x^3 + 13.5x^2 - 6.75x - 10.125 = 0$

4

Numerical Solution of Systems of Equations

This chapter covers the numerical solution of linear and nonlinear systems of equations. Linear systems are discussed first, followed by more specialized methods to efficiently handle large linear systems. Ill-conditioning symptoms, as well as pertinent remedies are also introduced. The chapter ends with iterative solution of nonlinear systems of equations.

4.1 Linear Systems of Equations

A linear system of n algebraic equations in n unknowns x_1, x_2, \dots, x_n is in the form

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases} \quad (4.1)$$

where a_{ij} ($i, j = 1, 2, \dots, n$) and b_k ($k = 1, 2, \dots, n$) are known constants, and a_{ij} 's are the coefficients. If every b_k is zero, the system is homogeneous, otherwise it is nonhomogeneous. Equation 4.1 can be conveniently expressed in matrix form, as

$$\mathbf{Ax} = \mathbf{b} \quad (4.2)$$

with

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}_{n \times n}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}_{n \times 1}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix}_{n \times 1}$$

where \mathbf{A} is the coefficient matrix. A set of values for x_1, x_2, \dots, x_n satisfying Equation 4.1 forms a solution of the system. The vector \mathbf{x} with components x_1, x_2, \dots, x_n is the solution vector for Equation 4.2. If $x_1 = 0 = x_2 = \dots = x_n$, the solution $\mathbf{x} = \mathbf{0}_{n \times 1}$ is called the trivial solution. The augmented matrix for Equation 4.2 is defined as

$$[\mathbf{A} \mid \mathbf{b}] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array} \right]_{n \times (n+1)} \quad (4.3)$$

4.2 Numerical Solution of Linear Systems

As described in Figure 4.1, numerical methods for solving linear systems of equations are divided into two categories: direct methods and indirect methods.

A direct method computes the solution of Equation 4.2 by performing a pre-determined number of operations. These methods transform the original system into an equivalent system in which the coefficient matrix is upper-triangular, lower-triangular, or diagonal, making the system much easier to solve. Indirect methods use iterations to find the approximate solution. The iteration process begins with an initial vector and generates successive approximations that eventually converge to the actual solution. Unlike direct methods, the number of operations required by iterative methods is not known in advance.

4.3 Gauss Elimination Method

Gauss elimination is a procedure that transforms a linear system of equations into upper-triangular form, the solution of which is found by back substitution. It is important to note that the augmented matrix $[A|b]$ completely represents the linear system $Ax = b$, therefore all modifications must be applied to the augmented matrix and not matrix A alone. The transformation into upper-triangular form is achieved by using elementary row operations (EROs) listed below.

- ERO₁ Multiply a row of the augmented matrix by a nonzero constant,
- ERO₂ Interchange any two rows of the augmented matrix,
- ERO₃ Multiply the i th row of the augmented matrix by a constant $\alpha \neq 0$ and add the result to the k th row, then replace the k th row with the outcome. The i th row is called the pivot row.

The nature of a linear system is preserved under EROs. If a linear system undergoes a finite number of EROs, then the new system and the original one are called row-equivalent.

Consider the system in Equation 4.1. The first objective is to eliminate x_1 in all equations below the first, thus the first row is the pivot row. The entry that plays the most important

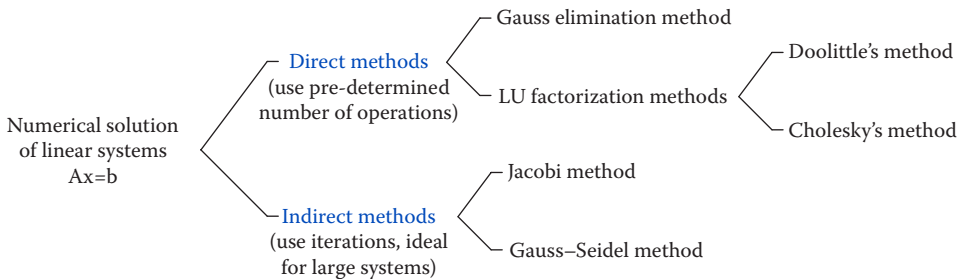


FIGURE 4.1

Classification of methods to solve a linear system of equations.

role here is a_{11} , known as the pivot, the coefficient of x_1 in the first row. If $a_{11} = 0$, the first row must be interchanged with another row (ERO₂) to ensure that x_1 has a nonzero coefficient. This is called partial pivoting. Another situation that may lead to partial pivoting is when a pivot is very small in magnitude, with a potential to cause round-off errors. Suppose x_1 has been eliminated via EROs, so that we now have a new system in which the first equation is as in the original, while the rest are generally changed, and are free of x_1 . The next step is to focus on the coefficient of x_2 in the second row of this new system. If it is nonzero, and not very small, we use it as the pivot and eliminate x_2 in all the lower-level equations. Here, the second row is the pivot row and remains unchanged. This continues until an upper-triangular system is formed. Finally, back substitution is used to find the solution.

EXAMPLE 4.1: GAUSS ELIMINATION WITH PARTIAL PIVOTING

Using Gauss elimination, find the solution x_1, x_2, x_3, x_4 of the system whose augmented matrix is

$$\left[\begin{array}{cccc|c} -1 & 2 & 3 & 1 & 3 \\ 2 & -4 & 1 & 2 & -1 \\ -3 & 8 & 4 & -1 & 6 \\ 1 & 4 & 7 & -2 & -4 \end{array} \right]$$

Solution

Because the (1, 1) entry is nonzero, we use it as the pivot to eliminate the entries directly below it. For instance, multiply the first row (pivot row) by 2 and add the result to the second row, then replace the second row by the outcome; ERO₂. All details are shown in Figure 4.2. Next, we focus on the (2, 2) entry in the second row of the new system. Because it is zero, the second row must be switched with any other row below it, say, the third row. As a result, the (2, 2) element is now 2, and is used as the pivot to zero out the entries below it. Since the one directly beneath it is already zero, by design, only one ERO₃ is needed. Finally, the (3, 3) entry in the latest system is 7, and applying one last ERO₃ yields an upper-triangular system as shown in Figure 4.3.

$$\begin{array}{l} \textcircled{1} \quad \textcircled{-3} \quad \textcircled{2} \\ + \quad + \quad + \end{array} \left[\begin{array}{cccc|c} -1 & 2 & 3 & 1 & 3 \\ 2 & -4 & 1 & 2 & -1 \\ -3 & 8 & 4 & -1 & 6 \\ 1 & 4 & 7 & -2 & -4 \end{array} \right] \text{Pivot row} \longrightarrow \left[\begin{array}{cccc|c} -1 & 2 & 3 & 1 & 3 \\ 0 & 0 & 7 & 4 & 5 \\ 0 & 2 & -5 & -4 & -3 \\ 0 & 6 & 10 & -1 & -1 \end{array} \right]$$

FIGURE 4.2

First three operations in Example 4.1.

$$\begin{array}{l} \textcircled{-3} \\ + \end{array} \left[\begin{array}{cccc|c} -1 & 2 & 3 & 1 & 3 \\ 0 & 2 & -5 & -4 & -3 \\ 0 & 0 & 7 & 4 & 5 \\ 0 & 6 & 10 & -1 & -1 \end{array} \right] \text{Pivot row} \xrightarrow{\textcircled{-25/7} \quad +} \left[\begin{array}{cccc|c} -1 & 2 & 3 & 1 & 3 \\ 0 & 2 & -5 & -4 & -3 \\ 0 & 0 & 7 & 4 & 5 \\ 0 & 0 & 25 & 11 & 8 \end{array} \right] \text{Pivot row} \longrightarrow \left[\begin{array}{cccc|c} -1 & 2 & 3 & 1 & 3 \\ 0 & 2 & -5 & -4 & -3 \\ 0 & 0 & 7 & 4 & 5 \\ 0 & 0 & 0 & -23/7 & -69/7 \end{array} \right]$$

FIGURE 4.3

Transformation into upper-triangular form in Example 4.1.

The solution is then found by back substitution as follows. The last row gives

$$-\frac{23}{7}x_4 = -\frac{69}{7} \Rightarrow x_4 = 3$$

Moving up one row at a time, each time using the latest information on the unknowns, we find

$$\begin{aligned}x_3 &= \frac{1}{7}(5 - 4x_4) = -1 & x_1 &= 1 \\x_2 &= \frac{1}{2}(5x_3 + 4x_4 - 3) = 2 & \Rightarrow & x_2 = 2 \\x_1 &= 2x_2 + 3x_3 + x_4 - 3 = 1 & x_3 &= -1\end{aligned}$$

Therefore, the solution is $x_1 = 1, x_2 = 2, x_3 = -1, x_4 = 3$.

4.3.1 Choosing the Pivot Row: Partial Pivoting with Row Scaling

When using partial pivoting, in the first step of the elimination process, it is common to choose as the pivot row the row in which x_1 has the largest (in absolute value) coefficient. The subsequent steps are treated in a similar manner. This is mainly to handle round-off error while dealing with large matrices. There is also total pivoting where the idea is to locate the entry of the coefficient matrix \mathbf{A} that is the largest in absolute value. This entry corresponds to one of the unknowns, say, x_m . Then, the first variable to be eliminated is x_m . A similar logic applies to the new system to decide which variable has to be eliminated next. However, total pivoting is not very practical because it requires much more computational effort than partial pivoting. Instead, partial pivoting with scaling is used where we choose the pivot row to be the row in which x_1 has the largest (absolute value) coefficient relative to the other entries in that row. More specifically, consider the first step, where x_1 is to be eliminated. We will choose the pivot row as follows. Assume \mathbf{A} is $n \times n$.

1. In each row i of \mathbf{A} , find the entry with the largest absolute value, and call it M_i .
2. In each row i , find the ratio of the absolute value of the coefficient of x_1 to the absolute value of M_i , that is,

$$r_i = \frac{|a_{i1}|}{|M_i|}$$

3. Among r_i ($i = 1, 2, \dots, n$) pick the largest. Whichever row is responsible for this maximum value is picked as the pivot row. Eliminate x_1 to obtain a new system.
4. In the new system, consider the $(n-1) \times (n-1)$ submatrix of the coefficient matrix occupying the lower right corner. In this matrix use the same logic as above to choose the pivot row to eliminate x_2 , and so on.

EXAMPLE 4.2: PARTIAL PIVOTING WITH SCALING

Use partial pivoting with scaling to solve the 3×3 system with the augmented matrix

$$[\mathbf{A} \mid \mathbf{b}] = \left[\begin{array}{ccc|c} -4 & -3 & 5 & 0 \\ 6 & 7 & -3 & 2 \\ 2 & -1 & 1 & 6 \end{array} \right]$$

Solution

The three values of r_i are found as

$$r_1 = \frac{|-4|}{|5|} = \frac{4}{5}, \quad r_2 = \frac{|6|}{|7|} = \frac{6}{7}, \quad r_3 = \frac{|2|}{|2|} = 1$$

Since r_3 is the largest, it is the third row that produces the maximum value hence it is chosen as the pivot row. Switch the first and the third row in the original system and eliminate x_1 using EROs to obtain [Figure 4.4](#).

To eliminate x_2 , consider the 2×2 submatrix \mathbf{B} and compute the corresponding ratios,

$$\frac{|10|}{|10|} = 1, \quad \frac{|-5|}{|7|} = \frac{5}{7}$$

so that the first row (in matrix \mathbf{B}) is picked as the pivot row. Row operations yield

$$\left[\begin{array}{ccc|c} 2 & -1 & 1 & 6 \\ 0 & 10 & -6 & -16 \\ 0 & 0 & 4 & 4 \end{array} \right]$$

and back substitution gives the solution; $x_3 = 1$, $x_2 = -1$, $x_1 = 2$.

4.3.2 Permutation Matrices

In the foregoing analysis, a linear $n \times n$ system was solved by Gauss elimination via EROs. In the process, the original system $\mathbf{Ax} = \mathbf{b}$ was transformed into $\mathbf{Ux} = \tilde{\mathbf{b}}$ where \mathbf{U} is an upper-triangular matrix with nonzero diagonal entries. So, there must exist an $n \times n$ matrix \mathbf{P} so that pre-multiplication of the original system by \mathbf{P} yields

$$\mathbf{P}[\mathbf{Ax}] = \mathbf{Pb} \Rightarrow [\mathbf{PA}]\mathbf{x} = \mathbf{Pb} \Rightarrow \mathbf{Ux} = \tilde{\mathbf{b}} \quad (4.4)$$

$$\left[\begin{array}{ccc|c} 2 & -1 & 1 & 6 \\ 6 & 7 & -3 & 2 \\ -4 & -3 & 5 & 0 \end{array} \right] \xrightarrow{\text{Eliminate } x_1} \left[\begin{array}{ccc|c} 2 & -1 & 1 & 6 \\ 0 & 10 & -6 & -16 \\ 0 & -5 & 7 & 12 \end{array} \right]$$

\mathbf{B}

FIGURE 4.4

Partial pivoting with scaling.

where $\mathbf{U} = \mathbf{P}\mathbf{A}$ and $\tilde{\mathbf{b}} = \mathbf{P}\mathbf{b}$. In order to identify this matrix \mathbf{P} , we need permutation matrices. The simplest way to describe these matrices is to go through an example. Let us refer to the 4×4 system in Example 4.1. Because the size is 4, we start with the 4×4 identity matrix,

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Consider the three EROs in the first step of Example 4.1; see Figure 4.2. Apply them to \mathbf{I} to get the matrix \mathbf{P}_1 (shown below). Next, focus on the second step, where there was only one ERO; the second and third rows were switched. Apply that to \mathbf{I} to obtain \mathbf{P}_2 . The third step also involved one ERO only. Apply to \mathbf{I} to get \mathbf{P}_3 . Finally, application of the operation in the last step to \mathbf{I} gives \mathbf{P}_4 .

$$\mathbf{P}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -3 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{P}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{P}_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -3 & 0 & 1 \end{bmatrix}, \quad \mathbf{P}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{25}{7} & 1 \end{bmatrix}$$

Each \mathbf{P}_i is called a permutation matrix, reflecting the operations in each step of Gauss elimination. Then,

$\mathbf{P}_1\mathbf{A}$ yields the coefficient matrix at the conclusion of the first step in Example 4.1

$\mathbf{P}_2(\mathbf{P}_1\mathbf{A})$ gives the coefficient matrix at the end of the second step

$\mathbf{P}_3(\mathbf{P}_2\mathbf{P}_1\mathbf{A})$ produces the coefficient matrix at the end of the third step

$\mathbf{P}_4(\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1\mathbf{A})$ gives the upper-triangular coefficient matrix \mathbf{U} at the end of the fourth step

Letting $\mathbf{P} = \mathbf{P}_4\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1$, then

$$\mathbf{P}\mathbf{A} = \begin{bmatrix} -1 & 2 & 3 & 1 \\ 0 & 2 & -5 & -4 \\ 0 & 0 & 7 & 4 \\ 0 & 0 & 0 & -\frac{23}{7} \end{bmatrix} \quad \text{and} \quad \mathbf{P}\mathbf{b} = \tilde{\mathbf{b}} = \begin{bmatrix} 3 \\ -3 \\ 5 \\ -\frac{69}{7} \end{bmatrix}$$

Subsequently, the final triangular system has the augmented matrix $[\mathbf{U}|\tilde{\mathbf{b}}]$, as suggested by Equation 4.4.

The user-defined function `GaussPivotScale` uses Gauss elimination with partial pivoting and row scaling to transform a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ into an upper-triangular system, and subsequently finds the solution vector by back substitution. The user-defined function `BackSub` performs the back substitution portion and is given below.


```

function x = GaussPivotScale(A,b)
%
% GaussPivotScale uses Gauss elimination with partial pivoting and
% row scaling to solve the linear system Ax = b.
%
% x = GaussPivotScale(A,b), where
%
% A is the n-by-n coefficient matrix,
% b is the n-by-1 result vector,
%
% x is the n-by-1 solution vector.
%
n = length(b);
A = [A b]; % Define augmented matrix
for k = 1:n-1,
% Find maximum magnitude in each row
M = max(abs(A(k:end, k:end-1)), [], 2);
a = abs(A(k:end, k)); % Find maximum in kth column
I = max(a./M); % Find row with maximum ratio
I = I + k - 1; % Adjust relative row to actual row
if I > k
A([k I], :) = A([I k], :); % Pivot rows
end
m = A(k+1:n, k)/A(k, k); % Construct multipliers
[Ak, M] = meshgrid(A(k, :), m); % Create mesh
A(k+1:n, :) = A(k+1:n, :) - Ak.*M;
end
Ab = A;
% Find the solution vector using back substitution
x = BackSub(Ab);

```

```

function x = BackSub(Ab)
%
% BackSub returns the solution vector of the upper triangular augmented
% matrix Ab using back substitution.
%
% x = BackSub(Ab), where
%
% Ab is the n-by-(n+1) augmented matrix,
%
% x is the n-by-1 solution vector.
%
n = size(Ab, 1);
for k = n:-1:1,
Ab(k, :) = Ab(k, :)./Ab(k, k); % Construct multipliers
Ab(1:k-1, n+1) = Ab(1:k-1, n+1)-Ab(1:k-1, k)*Ab(k, n+1); % Adjust rows
end
x = Ab(:, end);

```

EXAMPLE 4.3: PARTIAL PIVOTING WITH SCALING

The linear system in Example 4.2 can be solved by executing the under-defined function GaussPivotScale:

```
>> A = [-4 -3 5;6 7 -3;2 -1 1]; b = [0;2;6];
>> x = GaussPivotScale(A,b)

x =
     2
    -1
     1
```

4.3.3 Counting the Number of Operations

The objective is to determine approximately the total number of operations required by Gauss elimination for solving an $n \times n$ system. We note that the entire process consists of two parts: (1) elimination, and (2) back substitution.

4.3.3.1 Elimination

Suppose the first $k-1$ steps of elimination have been performed, and we are in the k th step. This means that the coefficients of x_k must be made into zeros in the remaining $n-k$ rows of the augmented matrix. There,

$n-k$ divisions are needed to figure out the multipliers

$(n-k)(n-k+1)$ multiplications

$(n-k)(n-k+1)$ additions

Noting that the elimination process consists of $n-1$ steps, the total number of operations N_e is

$$N_e = \underbrace{\sum_{k=1}^{n-1} (n-k)}_{\text{Divisions}} + \underbrace{\sum_{k=1}^{n-1} (n-k)(n-k+1)}_{\text{Multiplications}} + \underbrace{\sum_{k=1}^{n-1} (n-k)(n-k+1)}_{\text{Additions}} \quad (4.5)$$

Equation 4.5 may be rewritten as (verify)

$$N_e = \sum_{p=1}^{n-1} p + 2 \sum_{p=1}^{n-1} p(p+1) = 3 \sum_{p=1}^{n-1} p + 2 \sum_{p=1}^{n-1} p^2 \quad (4.6)$$

Using the well-known identities

$$\sum_{p=1}^M p = \frac{M(M+1)}{2} \quad \text{and} \quad \sum_{p=1}^M p^2 = \frac{M(M+1)(2M+1)}{6}$$

in Equation 4.6, the total number of operations in the elimination process is given by

$$N_e = 3 \frac{(n-1)n}{2} + 2 \frac{(n-1)n(2n-1)}{6} \stackrel{\text{For large } n}{\cong} \frac{2}{3} n^3 \quad (4.7)$$

where we have neglected lower powers of n . The approximation is particularly useful for a large system. With the above information, we can show, for example, that the total number of multiplications is roughly $\frac{1}{3}n^3$.

4.3.3.2 Back Substitution

When back substitution is used to determine x_k , one performs

$n-k$ multiplications

$n-k$ subtractions

1 division

In Example 4.1, for instance, $n = 4$, and solving for x_2 (so that $k = 2$) requires two multiplications ($n-k = 2$), two subtractions, and 1 division. So, the total number of operations N_s for the back substitution process is

$$N_s = \sum_{k=1}^n 1 + \sum_{k=1}^n (n-k) + \sum_{k=1}^n (n-k) = n + \frac{n(n-1)}{2} + \frac{n(n-1)}{2} \stackrel{\text{For large } n}{\cong} n^2 \quad (4.8)$$

Divisions
Multiplications
Subtractions

If n is large, N_e dominates N_s , and the total number of operations in Gauss elimination (for a large system) is

$$N_o = N_e + N_s \cong \frac{2}{3}n^3$$

4.3.4 Tridiagonal Systems

Tridiagonal systems arise often in engineering applications and appear in the special form

$$\begin{bmatrix} d_1 & u_1 & 0 & 0 & \dots & 0 \\ l_2 & d_2 & u_2 & 0 & \dots & 0 \\ 0 & l_3 & d_3 & u_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & 0 & l_{n-1} & d_{n-1} & u_{n-1} \\ 0 & 0 & \dots & 0 & l_n & d_n \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_{n-1} \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_{n-1} \\ b_n \end{Bmatrix} \quad (4.9)$$

where d_i ($i = 1, 2, \dots, n$) are the diagonal entries, l_i ($i = 2, \dots, n$) the lower diagonal entries, and u_i ($i = 1, 2, \dots, n-1$) the upper diagonal entries of the coefficient matrix. Gauss elimination can be used for solving such systems, but is not recommended. This is because Gauss elimination does not take into account the very special structure of a tridiagonal coefficient matrix, and as a result will perform unnecessary operations to find the solution. Instead, we use an efficient technique known as the Thomas method, which takes advantage of the fact that the coefficient matrix has several zero entries. The Thomas method uses Gauss elimination with the diagonal entry scaled to 1 in each step.

4.3.4.1 Thomas Method

Writing out the equations in Equation 4.9, we have

$$\begin{aligned}d_1x_1 + u_1x_2 &= b_1 \\l_2x_1 + d_2x_2 + u_2x_3 &= b_2 \\&\dots \\l_{n-1}x_{n-2} + d_{n-1}x_{n-1} + u_{n-1}x_n &= b_{n-1} \\l_nx_{n-1} + d_nx_n &= b_n\end{aligned}$$

In the first equation the diagonal entry is scaled to 1, that is, multiply the equation by $1/a_{11}$. Therefore, in the first equation the modified elements are

$$u_1 = \frac{u_1}{d_1}, \quad b_1 = \frac{b_1}{d_1}$$

All remaining equations, except the very last one, involve three terms. In these equations the modified elements are

$$u_i = \frac{u_i}{d_i - u_{i-1}l_i}, \quad b_i = \frac{b_i - b_{i-1}l_i}{d_i - u_{i-1}l_i}, \quad i = 2, 3, \dots, n-1$$

Note that in every stage, the (latest) modified values for all elements must be used. In the last equation,

$$b_n = \frac{b_n - b_{n-1}l_n}{d_n - u_{n-1}l_n}$$

Finally, use back substitution to solve the system:

$$\begin{aligned}x_n &= b_n \\x_i &= b_i - u_ix_{i+1}, \quad i = n-1, n-2, \dots, 2, 1\end{aligned}$$

EXAMPLE 4.4: THOMAS METHOD

Solve the following tridiagonal system using the Thomas method:

$$\begin{bmatrix} 3 & -1 & 0 \\ 1 & 2 & 1 \\ 0 & -1 & -3 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ -4 \\ 0 \end{Bmatrix}$$

Solution

We first identify all elements:

$$d_1 = 3, d_2 = 2, d_3 = -3, l_2 = 1, l_3 = -1, u_1 = -1, u_2 = 1, b_1 = 6, b_2 = -4, b_3 = 0.$$

In the first equation, the modified elements are

$$u_1 = \frac{u_1}{d_1} = \frac{-1}{3}, \quad b_1 = \frac{b_1}{d_1} = \frac{6}{3} = 2$$

In the second equation,

$$u_2 = \frac{u_2}{d_2 - u_1 l_2} = \frac{1}{2 - (-\frac{1}{3})(1)} = \frac{3}{7}, \quad b_2 = \frac{b_2 - b_1 l_2}{d_2 - u_1 l_2} = \frac{-4 - (2)(1)}{2 - (-\frac{1}{3})(1)} = -\frac{18}{7}$$

In the last equation,

$$b_3 = \frac{b_3 - b_2 l_3}{d_3 - u_2 l_3} = \frac{0 - (-\frac{18}{7})(-1)}{-3 - (\frac{3}{7})(-1)} = 1$$

Back substitution yields

$$\begin{aligned} x_3 &= b_3 = 1 \\ x_2 &= b_2 - u_2 x_3 = -\frac{18}{7} - \left(\frac{3}{7}\right)(1) = -3 \\ x_1 &= b_1 - u_1 x_2 = 2 - \left(-\frac{1}{3}\right)(-3) = 1 \end{aligned} \quad \Rightarrow \quad \text{Solution vector} \quad \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 1 \\ -3 \\ 1 \end{Bmatrix}$$

The user-defined function `ThomasMethod` uses the Thomas method to solve an $n \times n$ tridiagonal system $\mathbf{Ax} = \mathbf{b}$. The inputs are matrix \mathbf{A} and vector \mathbf{b} . From \mathbf{A} , three $n \times 1$ vectors will be constructed:

$$\mathbf{d} = [a_{11} \quad a_{22} \quad a_{33} \quad \dots \quad a_{nn}]^T$$

$$\mathbf{l} = [0 \quad a_{21} \quad a_{32} \quad \dots \quad a_{n,n-1}]^T$$

$$\mathbf{u} = [a_{12} \quad a_{23} \quad \dots \quad a_{n-1,n} \quad 0]^T$$

These are subsequently used in the procedure outlined above to determine the solution vector \mathbf{x} .

```
function x = ThomasMethod(A,b)
%
% ThomasMethod uses Thomas method to find the solution vector x of a
% tridiagonal system Ax = b.
%
% x = ThomasMethod(A,b), where
%
% A is a tridiagonal n-by-n coefficient matrix,
% b is the n-by-1 vector of the right-hand sides,
%
```

```

%      x is the n-by-1 solution vector.
%
n = size(A,1);
d = diag(A);           % Vector of diagonal entries of A
l = [0;diag(A,-1)];   % Vector of lower diagonal elements
u = [diag(A,1);0];    % Vector of upper diagonal elements

u(1) = u(1)/d(1); b(1) = b(1)/d(1);      % First equation

for i=2:n-1,          % The next n-2 equations
    den = d(i) - u(i-1)*l(i);
    if den == 0,
        x = 'failure, division by zero';
        return
    end
    u(i) = u(i)/den; b(i) = (b(i)-b(i-1)*l(i))/den;
end

b(n)=(b(n)-b(n-1)*l(n))/(d(n)-u(n-1)*l(n)); % Last equation
x(n) = b(n);
for i=n-1:-1:1,
    x(i) = b(i) - u(i)*x(i+1);
end
x = x';

```

The result obtained in Example 4.4 can be verified by executing this function, as

```

>> A = [3 -1 0;1 2 1;0 -1 -3]; b = [6;-4;0];
>> x = ThomasMethod(A,b)

```

```

x =
    1.0000
   -3.0000
    1.0000

```

4.3.4.2 MATLAB Built-In Function "\ "

The built-in function in MATLAB for solving a linear system $\mathbf{Ax} = \mathbf{b}$ is the backslash (\backslash), and the solution vector is obtained via $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$. It is important to note that $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$ computes the solution vector by Gauss elimination and not by $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

For the linear system in Example 4.4, this yields

```

>> x = A\b

ans =

    1.0000
   -3.0000
    1.0000

```

4.4 LU Factorization Methods

In the last section, we learned that solving a large $n \times n$ system $\mathbf{Ax} = \mathbf{b}$ using Gauss elimination requires approximately $\frac{2}{3}n^3$ operations. There are other direct methods that require fewer operations than Gauss elimination. These methods make use of the LU factorization of the coefficient matrix \mathbf{A} .

LU factorization (or decomposition) of a matrix $\mathbf{A}_{n \times n}$ means expressing the matrix as $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}_{n \times n}$ is a lower triangular matrix and $\mathbf{U}_{n \times n}$ is upper triangular. Subsequently, the original system $\mathbf{Ax} = \mathbf{b}$ is rewritten as

$$[\mathbf{LU}]\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad \mathbf{L}[\mathbf{Ux}] = \mathbf{b} \quad \Rightarrow \quad \mathbf{L} \underbrace{[\mathbf{Ux}]_y} = \mathbf{b}$$

Letting $\mathbf{Ux} = \mathbf{y}_{n \times 1}$ the above can be solved in two steps:

$$\begin{cases} \mathbf{Ly} = \mathbf{b} & \xrightarrow{\text{Forward substitution}} & \mathbf{y} \\ \mathbf{Ux} = \mathbf{y} & \xrightarrow{\text{Back substitution}} & \mathbf{x} \end{cases}$$

Note that each of the two systems is triangular, hence easy to solve. Because $\mathbf{Ly} = \mathbf{b}$ is a lower triangular system, it can be solved using forward substitution. The system $\mathbf{Ux} = \mathbf{y}$ is upper triangular and is solved via back substitution.

There are different ways to accomplish the factorization of matrix \mathbf{A} , depending on the specific restrictions imposed on \mathbf{L} or \mathbf{U} . For example, Crout factorization (see Problem Set) requires the diagonal entries of \mathbf{U} be 1's, while \mathbf{L} is a general lower triangular matrix. Another technique, known as Doolittle factorization, uses the results from different steps of Gauss elimination. These two approaches have similar performances, but we will present Doolittle factorization here.

4.4.1 Doolittle Factorization

Doolittle factorization of \mathbf{A} is $\mathbf{A} = \mathbf{LU}$, where \mathbf{L} is lower triangular consisting of 1's along the diagonal, and \mathbf{U} is upper triangular. That the diagonal elements of \mathbf{L} are chosen as 1's can be explained using a generic 3×3 matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

If \mathbf{L} and \mathbf{U} are selected in their most general forms, then

$$\mathbf{A} = \mathbf{LU} \quad \Rightarrow \quad \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

This implies that there are nine known quantities (entries of \mathbf{A}), but 12 unknown quantities: six in matrix \mathbf{L} and six in matrix \mathbf{U} . By selecting the diagonal entries l_{11} , l_{22} , and l_{33} to be 1's, the number of unknown quantities is reduced to the number of known quantities. The same strategy remains valid for any $n \times n$ matrix \mathbf{A} .

4.4.2 Finding \mathbf{L} and \mathbf{U} Using Steps of Gauss Elimination

The lower triangular matrix \mathbf{L} comprises 1's along the main diagonal and negatives of the multipliers (from Gauss elimination) below the main diagonal. The upper triangular matrix \mathbf{U} is the upper triangular form of \mathbf{A} in the final step of Gauss elimination.

EXAMPLE 4.5: DOOLITTLE FACTORIZATION USING STEPS OF GAUSS ELIMINATION

Find the Doolittle factorization of

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 6 \\ 2 & -1 & 1 \\ 4 & -2 & 3 \end{bmatrix}$$

Solution

Imagine \mathbf{A} as the coefficient matrix in a linear system, which is being solved by Gauss elimination. Figure 4.5 shows a sequence of EROs that transform \mathbf{A} into an upper triangular matrix.

The final upper triangular form is \mathbf{U} . Three multipliers, -2 , -4 , and -2 , have been used to create zeros in the (2,1), (3,1), and (3,2) positions, respectively. Therefore, 2, 4, and 2 will occupy the respective slots in matrix \mathbf{L} . As a result,

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ \boxed{2} & 1 & 0 \\ \boxed{4} & \boxed{2} & 1 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 1 & 3 & 6 \\ 0 & -7 & -11 \\ 0 & 0 & 1 \end{bmatrix}$$

4.4.3 Finding \mathbf{L} and \mathbf{U} Directly

A more efficient way to find \mathbf{L} and \mathbf{U} is a direct approach, as demonstrated in the following example.

EXAMPLE 4.6: DIRECT CALCULATION OF \mathbf{L} AND \mathbf{U} IN DOOLITTLE FACTORIZATION

Consider the matrix in Example 4.5,

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 6 \\ 2 & -1 & 1 \\ 4 & -2 & 3 \end{bmatrix}$$

Based on the structures of \mathbf{L} and \mathbf{U} in Doolittle factorization, we write

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

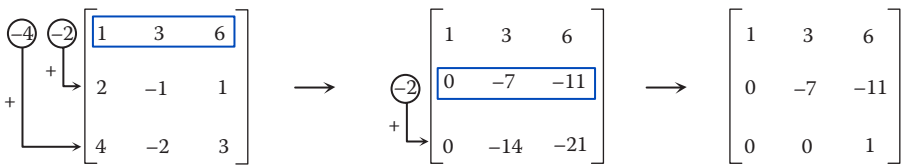


FIGURE 4.5
Reduction to an upper triangular matrix.

Setting $\mathbf{A} = \mathbf{LU}$, we find

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{bmatrix}$$

Each entry on the left must be equal to the corresponding entry on the right. This generates nine equations in nine unknowns. The entries in the first row of \mathbf{U} are found immediately, as

$$u_{11} = a_{11}, \quad u_{12} = a_{12}, \quad u_{13} = a_{13}$$

The elements in the first column of \mathbf{L} are found as

$$l_{21} = \frac{a_{21}}{u_{11}}, \quad l_{31} = \frac{a_{31}}{u_{11}}$$

The entries in the second row of \mathbf{U} are calculated via

$$u_{22} = a_{22} - l_{21}u_{12}, \quad u_{23} = a_{23} - l_{21}u_{13}$$

The element in the second column of \mathbf{L} is found as

$$l_{32} = \frac{a_{32} - l_{31}u_{12}}{u_{22}}$$

Finally, the entry in the third row of \mathbf{U} is given by

$$u_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23}$$

Using the entries of matrix \mathbf{A} and solving the nine equations just listed, we find

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 2 & 1 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 1 & 3 & 6 \\ 0 & -7 & -11 \\ 0 & 0 & 1 \end{bmatrix}$$

This clearly agrees with the outcome of Example 4.5.

The direct calculation of the entries of \mathbf{L} and \mathbf{U} in Doolittle factorization can be performed systematically for an $n \times n$ matrix \mathbf{A} using the steps outlined in Example 4.6. The user-defined function `DoolittleFactor` performs all the

operations in the order suggested in Example 4.6 and returns the appropriate L and U matrices.

```
function [L, U] = DoolittleFactor(A)
%
% DoolittleFactor returns the Doolittle factorization of matrix A.
%
% [L, U] = DoolittleFactor(A), where
%
% A is an n-by-n matrix,
%
% L is the lower triangular matrix with 1's along the diagonal,
% U is an upper triangular matrix.
%
n = size(A,1);
L = eye(n); U = zeros(n,n); % Initialize
for i = 1:n,
    U(i,i) = A(i,i)-L(i,1:i-1)*U(1:i-1,i);
    for j = i+1:n,
        U(i,j) = A(i,j)-L(i,1:i-1)*U(1:i-1,j);
        L(j,i) = (A(j,i)-L(j,1:i-1)*U(1:i-1,i))/U(i,i);
    end
end
end
```

The findings of the last example can readily be confirmed by executing this function.

```
>> A = [1 3 6;2 -1 1;4 -2 3];
>> [L, U] = DoolittleFactor(A)
```

```
L =
    1     0     0
    2     1     0
    4     2     1
```

```
U =
    1     3     6
    0    -7   -11
    0     0     1
```

4.4.3.1 Doolittle's Method to Solve a Linear System

Doolittle's method uses Doolittle factorization of A to solve $Ax = b$:

$$[LU]x = b \Rightarrow L[Ux] = b$$

This will be solved in two steps: Solving a lower-triangular system by forward substitution, followed by solving an upper-triangular system by back substitution.

$$\begin{cases} Ly = b & \Rightarrow y \\ Ux = y & \Rightarrow x \end{cases} \quad (4.10)$$

The user-defined function `DoolittleMethod` uses Doolittle factorization of the coefficient matrix, and subsequently solves the two triangular systems in Equation 4.10 using forward and back substitution, respectively, to find the solution vector x .

```

function x = DoolittleMethod(A,b)
%
% DoolittleMethod uses the Doolittle factorization of matrix A and
% solves the ensuing triangular systems to find the solution vector x.
%
% x = DoolittleMethod(A,b), where
%
% A is the n-by-n coefficient matrix,
% b is the n-by-1 vector of the right-hand sides,
%
% x is the n-by-1 solution vector.
%
[L, U] = DoolittleFactor(A);      % Find Doolittle factorization of A
n = size(A,1);

% Solve the lower triangular system Ly = b (forward substitution)
y = zeros(n,1);
y(1) = b(1);
for i = 2:n,
    y(i) = b(i)-L(i,1:i-1)*y(1:i-1);
end

% Solve the upper triangular system Ux = y (back substitution)
x = zeros(n,1);
x(n) = y(n)/U(n,n);
for i = n-1:-1:1,
    x(i) = (y(i)-U(i,i+1:n)*x(i+1:n))/U(i,i);
end
end

```

EXAMPLE 4.7: DOOLITTLE'S METHOD TO SOLVE A LINEAR SYSTEM

Using Doolittle's method, solve $Ax = b$ where

$$A = \begin{bmatrix} 1 & 3 & 6 \\ 2 & -1 & 1 \\ 4 & -2 & 3 \end{bmatrix}, \quad b = \begin{Bmatrix} 3 \\ 9 \\ 19 \end{Bmatrix}, \quad x = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}$$

Solution

Doolittle factorization of A was done in Examples 4.5 and 4.6. Using L and U in Equation 4.10,

$$Ly = b: \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 2 & 1 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} 3 \\ 9 \\ 19 \end{Bmatrix} \quad \begin{array}{l} \text{Forward substitution} \\ \Rightarrow \end{array} \quad \begin{array}{l} y_1 = 3 \\ 2y_1 + y_2 = 9 \\ 4y_1 + 2y_2 + y_3 = 19 \end{array} \quad \Rightarrow \quad y = \begin{Bmatrix} 3 \\ 3 \\ 1 \end{Bmatrix}$$

$$Ux = y: \begin{bmatrix} 1 & 3 & 6 \\ 0 & -7 & -11 \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 3 \\ 3 \\ 1 \end{Bmatrix} \quad \begin{array}{l} \text{Back substitution} \\ \Rightarrow \end{array} \quad \begin{array}{l} x_1 + 3x_2 + 6x_3 = 3 \\ -7x_2 - 11x_3 = 3 \\ x_3 = 1 \end{array} \quad \Rightarrow \quad x = \begin{Bmatrix} 3 \\ -2 \\ 1 \end{Bmatrix}$$

The result can be verified by executing the user-defined function `DoolittleMethod`.

```
>> A = [1 3 6; 2 -1 1; 4 -2 3]; b = [3; 9; 19];
>> x = DoolittleMethod(A,b)

x =
     3
    -2
     1
```

4.4.3.2 Operations Count

Doolittle's method comprises two phases: LU factorization of the coefficient matrix and forward/back substitution to solve the two subsequent triangular systems. For a large system $\mathbf{Ax} = \mathbf{b}$, the Doolittle factorization of \mathbf{A} requires roughly $\frac{1}{3}n^3$ operations. The ensuing triangular systems are simply solved by forward and back substitutions, each of which requires n^2 operations; Section 4.3. Therefore, the total number of operations is $\frac{1}{3}n^3 + n^2$, which is approximately $\frac{1}{3}n^3$ since n is large. This implies that Doolittle's method requires half as many operations as the Gauss elimination method.

4.4.4 Cholesky Factorization

A very special class of matrices encountered in many engineering applications is symmetric, positive definite matrices. A matrix $\mathbf{A} = [a_{ij}]_{n \times n}$ is positive definite if all of the following determinants are positive:

$$D_1 = a_{11} > 0, \quad D_2 = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} > 0, \quad D_3 = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} > 0, \dots, \quad D_n = |\mathbf{A}| > 0$$

Of course, \mathbf{A} is symmetric if $\mathbf{A} = \mathbf{A}^T$. For a symmetric, positive definite matrix there is a very special form of LU factorization, where the lower triangular matrix \mathbf{L} is in the general form (with no restrictions on the diagonal entries) and the upper triangular matrix \mathbf{U} is the transpose of \mathbf{L} . This is known as Cholesky factorization,

$$\mathbf{A} = \mathbf{LL}^T$$

For instance, in the case of a 3×3 matrix,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{bmatrix} = \begin{bmatrix} l_{11}^2 & l_{11}l_{21} & l_{11}l_{31} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & l_{21}l_{31} + l_{22}l_{32} \\ l_{31}l_{11} & l_{21}l_{31} + l_{22}l_{32} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{bmatrix} \quad (4.11)$$

Owing to symmetry, only six equations—as opposed to nine for Doolittle—need to be solved. The user-defined function `CholeskyFactor` performs all the operations and returns the appropriate \mathbf{L} and $\mathbf{U} = \mathbf{L}^T$ matrices.

```

function [L, U] = CholeskyFactor(A)
%
% CholeskyFactor returns the Cholesky factorization of matrix A.
%
% [L, U] = CholeskyFactor(A), where
%
%     A is a symmetric, positive definite n-by-n matrix,
%
%     L is a lower triangular matrix,
%     U = L' is an upper triangular matrix.
%
n = size(A,1);
L = zeros(n,n);    % Initialize
for i = 1:n,
    L(i,i) = sqrt(A(i,i)-L(i,1:i-1)*L(i,1:i-1)');
    for j = i+1:n,
        L(j,i) = (A(j,i)-L(j,1:i-1)*L(i,1:i-1)')/L(i,i);
    end
end
U = L';

```

4.4.4.1 Cholesky's Method to Solve a Linear System

Cholesky's method uses Cholesky factorization of \mathbf{A} to solve $\mathbf{Ax} = \mathbf{b}$. Substitution of $\mathbf{A} = \mathbf{LL}^T$ into the system yields

$$[\mathbf{LL}^T]\mathbf{x} = \mathbf{b} \Rightarrow \mathbf{L}[\mathbf{L}^T\mathbf{x}] = \mathbf{b}$$

which will be solved in two steps:

$$\begin{cases} \mathbf{Ly} = \mathbf{b} & \Rightarrow \mathbf{y} \\ \mathbf{L}^T\mathbf{x} = \mathbf{y} & \Rightarrow \mathbf{x} \end{cases} \quad (4.12)$$

Both systems are triangular, for which the solutions are found by forward and back substitutions. The user-defined function `CholeskyMethod` uses Cholesky factorization of the coefficient matrix, and subsequently solves the two triangular systems in Equation 4.12 to find the solution vector \mathbf{x} .

```

function x = CholeskyMethod(A,b)
%
% CholeskyMethod uses the Cholesky factorization of matrix A and
% solves the ensuing triangular systems to find the solution vector x.
%
% x = CholeskyMethod(A,b), where
%
%     A is a symmetric, positive definite n-by-n coefficient matrix,
%     b is the n-by-1 vector of the right-hand sides,
%
%     x is the n-by-1 solution vector.
%

```

```

[L, U] = CholeskyFactor(A); % Find Cholesky factorization of A
n = size(A,1);

% Solve the lower triangular system Ly = b (forward substitution)
y = zeros(n,1);
y(1) = b(1)/L(1,1);
for i = 2:n,
    y(i) = (b(i)-L(i,1:i-1)*y(1:i-1))/L(i,i);
end

% Solve the upper triangular system L'x = y (back substitution)
x = zeros(n,1);
x(n) = y(n)/U(n,n);
for i = n-1:-1:1,
    x(i) = (y(i)-U(i,i+1:n)*x(i+1:n))/U(i,i);
end
end

```

EXAMPLE 4.8: CHOLESKY'S METHOD TO SOLVE A LINEAR SYSTEM

Using Cholesky's method solve $\mathbf{Ax} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} 9 & 6 & -3 \\ 6 & 13 & -5 \\ -3 & -5 & 18 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -18 \\ -45 \\ 97 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Solution

The matrix \mathbf{A} is clearly symmetric since $\mathbf{A} = \mathbf{A}^T$, and it is positive definite because

$$D_1 = 9 > 0, \quad D_2 = \begin{vmatrix} 9 & 6 \\ 6 & 13 \end{vmatrix} = 81 > 0, \quad D_3 = |\mathbf{A}| = 1296 > 0$$

The elements listed in Equation 4.11 can be directly used to determine the six entries of \mathbf{L} . For instance,

$$l_{11}^2 = a_{11} \Rightarrow l_{11}^2 = 9 \Rightarrow l_{11} = 3$$

$$l_{11}l_{21} = a_{12} \Rightarrow l_{21} = \frac{a_{12}}{l_{11}} = \frac{6}{3} \Rightarrow l_{21} = 2$$

and so on. Continuing this process, we find

$$\mathbf{L} = \begin{bmatrix} 3 & 0 & 0 \\ 2 & 3 & 0 \\ -1 & -1 & 4 \end{bmatrix}$$

Using \mathbf{L} and \mathbf{L}^T in Equation 4.12, yields

$$\mathbf{L}\mathbf{y} = \mathbf{b} : \begin{bmatrix} 3 & 0 & 0 \\ 2 & 3 & 0 \\ -1 & -1 & 4 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} -18 \\ -45 \\ 97 \end{Bmatrix} \quad \begin{array}{l} \text{Forward substitution} \\ \Rightarrow \end{array} \quad \begin{array}{l} 3y_1 = -18 \\ 2y_1 + 3y_2 = -45 \\ -y_1 - y_2 + 4y_3 = 97 \end{array} \Rightarrow \mathbf{y} = \begin{Bmatrix} -6 \\ -11 \\ 20 \end{Bmatrix}$$

$$\mathbf{L}^T \mathbf{x} = \mathbf{y} : \begin{bmatrix} 3 & 2 & -1 \\ 0 & 3 & -1 \\ 0 & 0 & 4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} -6 \\ -11 \\ 20 \end{Bmatrix} \quad \begin{array}{l} \text{Back substitution} \\ \Rightarrow \end{array} \quad \begin{array}{l} 3x_1 + 2x_2 - x_3 = -6 \\ 3x_2 - x_3 = -11 \\ 4x_3 = 20 \end{array} \Rightarrow \mathbf{x} = \begin{Bmatrix} 1 \\ -2 \\ 5 \end{Bmatrix}$$

The result can be verified by executing the user-defined function `CholeskyMethod`.

```
>> A = [9 6 -3; 6 13 -5; -3 -5 18]; b = [-18; -45; 97];
>> x = CholeskyMethod(A,b)

x =
     1
    -2
     5
```

4.4.4.2 Operations Count

Cholesky's method comprises two parts: LU factorization of the coefficient matrix and forward/back substitutions to solve the two triangular systems. For a large system $\mathbf{Ax} = \mathbf{b}$, the Cholesky factorization of \mathbf{A} requires roughly $\frac{1}{3}n^3$ operations. The ensuing triangular systems are solved by forward and back substitutions, each requiring n^2 operations. Therefore, the total number of operations is roughly $\frac{1}{3}n^3 + n^2$, which is approximately $\frac{1}{3}n^3$ since n is large. This implies that Cholesky's method requires half as many operations as Gauss elimination method.

4.4.4.3 MATLAB Built-In Functions `lu` and `chol`

MATLAB has built-in functions to perform LU factorization of a square matrix: `lu` for general square matrices, and `chol` for symmetric, positive definite matrices. There are different ways of calling the function `lu`. For example, the outputs in `[L,U]=lu(A)` are \mathbf{U} , which is upper triangular and \mathbf{L} , which is the product of a lower triangular matrix and permutation matrices such that $\mathbf{LU} = \mathbf{A}$. On the other hand, `[L,U,P]=lu(A)` returns a lower triangular \mathbf{L} , an upper triangular \mathbf{U} , and permutation matrices \mathbf{P} such that $\mathbf{LU} = \mathbf{PA}$. Other options in `lu` allow for the control of pivoting when working with sparse matrices (large matrices with a large number of zero entries).

For a symmetric, positive definite matrix \mathbf{A} , the function call `U=chol(A)` returns an upper triangular matrix \mathbf{U} such that $\mathbf{U}^T\mathbf{U} = \mathbf{A}$. If the matrix is not positive definite, `chol` returns an error message.

EXAMPLE 4.9: BUILT-IN FUNCTION "LU"

Consider

$$\mathbf{A} = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 3 & -5 \\ 2 & 5 & 6 \end{bmatrix}$$

```

>> A = [4 -1 0;-1 3 -5;2 5 6];
>> [L,U,P] = lu(A)

L =
    1.0000         0         0
    0.5000    1.0000         0
   -0.2500    0.5000    1.0000

U =
    4.0000   -1.0000         0
         0    5.5000    6.0000
         0         0   -8.0000

P =
     1     0     0
     0     0     1
     0     1     0

```

It is readily seen that $\mathbf{LU} = \mathbf{PA}$,

```

>> L*U, P*A

ans =
     4     -1     0
     2     5     6
    -1     3    -5

ans =
     4     -1     0
     2     5     6
    -1     3    -5

```

Note that the permutation matrix \mathbf{P} is the 3×3 identity matrix with its second and third rows interchanged. This indicates that the second and third rows of matrix \mathbf{A} were first interchanged to obtain $\mathbf{PA} = \tilde{\mathbf{A}}$, followed by the Doolittle factorization of $\tilde{\mathbf{A}}$, that is, $\tilde{\mathbf{A}} = \mathbf{LU}$.

4.5 Iterative Solution of Linear Systems

In Sections 4.3 and 4.4, we introduced direct methods for solving $\mathbf{Ax} = \mathbf{b}$, which included the Gauss elimination and methods based on LU factorization of the coefficient matrix \mathbf{A} . We now turn our attention to indirect, or iterative, methods. In principle, a successful iteration process starts with an initial vector and generates a sequence of successive vectors that eventually converges to the solution vector \mathbf{x} .

Unlike direct methods, where the total number of operations is known in advance, the number of operations required by an iterative method depends on how many iteration steps must be performed for satisfactory convergence, as well as the nature of the system at hand. What is meant by convergence is that the iteration must be terminated as soon as two successive vectors are close to one another. A measure of the proximity of two vectors is provided by a vector norm.

4.5.1 Vector Norms

Norm of a vector $\mathbf{v}_{n \times 1}$, denoted by $\|\mathbf{v}\|$, provides a measure of how small or large \mathbf{v} is, and has the following properties:

- $\|\mathbf{v}\| \geq 0$ for all \mathbf{v} , and $\|\mathbf{v}\| = 0$ if and only if $\mathbf{v} = \mathbf{0}_{n \times 1}$

- $\|\alpha\mathbf{v}\| = |\alpha|\|\mathbf{v}\|$, $\alpha = \text{scalar}$
- $\|\mathbf{v} + \mathbf{w}\| \leq \|\mathbf{v}\| + \|\mathbf{w}\|$ for all vectors \mathbf{v} and \mathbf{w}

There are three commonly used vector norms, listed below. In all cases, vector \mathbf{v} is assumed in the form

$$\mathbf{v} = \begin{Bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{Bmatrix}$$

l_1 -norm, denoted by $\|\mathbf{v}\|_1$, is the sum of the absolute values of all components of \mathbf{v} :

$$\|\mathbf{v}\|_1 = |v_1| + |v_2| + \dots + |v_n| \quad (4.13)$$

l_∞ -norm, denoted by $\|\mathbf{v}\|_\infty$, is the largest (in absolute value) of all components of \mathbf{v} :

$$\|\mathbf{v}\|_\infty = \max\{|v_1|, |v_2|, \dots, |v_n|\} \quad (4.14)$$

l_2 -norm, denoted by $\|\mathbf{v}\|_2$, is the square root of the sum of the squares of all components of \mathbf{v} :

$$\|\mathbf{v}\|_2 = \left[v_1^2 + v_2^2 + \dots + v_n^2 \right]^{1/2} \quad (4.15)$$

EXAMPLE 4.10: VECTOR NORMS

Find the three norms of

$$\mathbf{v} = \begin{Bmatrix} 8.3 \\ -2.9 \\ -12 \\ 6.7 \end{Bmatrix}$$

1. Using Equations 4.13 through 4.15.
2. Using the MATLAB built-in function `norm`.

Solution

1. By Equations 4.13 through 4.15,

$$\|\mathbf{v}\|_1 = |8.3| + |-2.9| + |-12| + |6.7| = 29.9$$

$$\|\mathbf{v}\|_\infty = \max\{|8.3|, |-2.9|, |-12|, |6.7|\} = 12$$

$$\|\mathbf{v}\|_2 = \sqrt{(8.3)^2 + (-2.9)^2 + (-12)^2 + 6.7^2} = 16.3153$$

Note that all three norms return values that are of the *same order of magnitude*, as is always the case. If a certain norm of a vector happens to be small, the other norms will also be somewhat small, and so on.

2. MATLAB built-in function norm calculates vector and matrix norms.

```
>> v = [8.3;-2.9;-12;6.7];  
>> [norm(v,1), norm(v,inf), norm(v,2)]  
  
ans =  
  
29.9000 12.0000 16.3153
```

4.5.2 Matrix Norms

Norm of a matrix $\mathbf{A}_{n \times n}$, denoted by $\|\mathbf{A}\|$, is a nonnegative real number that provides a measure of how small or large \mathbf{A} is, and has the following properties:

- $\|\mathbf{A}\| \geq 0$ for all \mathbf{A} , and $\|\mathbf{A}\| = 0$ if and only if $\mathbf{A} = \mathbf{0}_{n \times n}$
- $\|\alpha \mathbf{A}\| = |\alpha| \|\mathbf{A}\|$, $\alpha = \text{scalar}$
- $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$ for all $n \times n$ matrices \mathbf{A} and \mathbf{B}
- $\|\mathbf{AB}\| \leq \|\mathbf{A}\| \|\mathbf{B}\|$ for all $n \times n$ matrices \mathbf{A} and \mathbf{B}

There are three commonly used matrix norms, listed below. In all cases, matrix \mathbf{A} is in the form $\mathbf{A} = [a_{ij}]_{n \times n}$.

1-norm (column-sum norm), denoted by $\|\mathbf{A}\|_1$, is defined as

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq n} \left\{ \sum_{i=1}^n |a_{ij}| \right\} \quad (4.16)$$

The sum of the absolute values of entries in each column of \mathbf{A} is calculated, and the largest is selected.

Infinite-norm (row-sum norm), denoted by $\|\mathbf{A}\|_\infty$, is defined as

$$\|\mathbf{A}\|_\infty = \max_{1 \leq i \leq n} \left\{ \sum_{j=1}^n |a_{ij}| \right\} \quad (4.17)$$

The sum of the absolute values of entries in each row of \mathbf{A} is calculated, and the largest is selected.

Euclidean norm (2-norm, Frobenius norm), denoted by $\|\mathbf{A}\|_E$, is defined as

$$\|\mathbf{A}\|_E = \left[\sum_{i=1}^n \sum_{j=1}^n a_{ij}^2 \right]^{1/2} \quad (4.18)$$

EXAMPLE 4.11: MATRIX NORMS

Find the three norms of

$$\mathbf{A} = \begin{bmatrix} 3 & 1.26 & -2 & 5 \\ -1 & 0 & 5.4 & 4.8 \\ 0.93 & -4 & 1 & 3.6 \\ -2 & -4.5 & 6 & 10 \end{bmatrix}$$

- Using Equations 4.16 through 4.18.
- Using the MATLAB built-in function `norm`.

Solution

- By Equations 4.16 through 4.18,

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq 4} \left\{ \sum_{i=1}^4 |a_{ij}| \right\} = \max\{6.93, 9.76, 14.4, 23.4\} = 23.4$$

$$\|\mathbf{A}\|_\infty = \max_{1 \leq i \leq 4} \left\{ \sum_{j=1}^4 |a_{ij}| \right\} = \max\{11.26, 11.2, 9.53, 22.5\} = 22.5$$

$$\|\mathbf{A}\|_E = \left\{ \sum_{i=1}^4 \sum_{j=1}^4 a_{ij}^2 \right\}^{1/2} = 16.8482$$

As it was the case with vector norms, the values returned by all three matrix norms are of the *same order of magnitude*.

2.

```
>> A = [3 1.26 -2 5;-1 0 5.4 4.8;0.93 -4 1 3.6;-2 -4.5 6 10];
>> [norm(A,1), norm(A,inf), norm(A,'fro')]
ans =
    23.4000    22.5000    16.8482
```

4.5.2.1 Compatibility of Vector and Matrix Norms

The three matrix norms above are compatible with the three vector norms introduced earlier, in the exact order they were presented. More specifically, the compatibility relations are

$$\begin{aligned} \|\mathbf{A}\mathbf{v}\|_1 &\leq \|\mathbf{A}\|_1 \|\mathbf{v}\|_1 \\ \|\mathbf{A}\mathbf{v}\|_\infty &\leq \|\mathbf{A}\|_\infty \|\mathbf{v}\|_\infty \\ \|\mathbf{A}\mathbf{v}\|_2 &\leq \|\mathbf{A}\|_E \|\mathbf{v}\|_2 \end{aligned} \quad (4.19)$$

EXAMPLE 4.12: COMPATIBILITY RELATIONS

The relations in Equation 4.19 can be verified for the vector and the matrix used in Examples 4.10 and 4.11 as follows:

$$\mathbf{A}\mathbf{v} = \begin{Bmatrix} 78.7460 \\ -40.9400 \\ 31.4390 \\ -8.5500 \end{Bmatrix} \quad \begin{array}{l} \text{Calculate vector norms} \\ \Rightarrow \end{array} \quad \|\mathbf{A}\mathbf{v}\|_1 = 159.6750, \quad \|\mathbf{A}\mathbf{v}\|_\infty = 78.7460, \quad \|\mathbf{A}\mathbf{v}\|_2 = 94.5438$$

Then, the compatibility relations in Equation 4.19 are readily verified as follows.

$$159.6750 \leq (23.4)(29.9), \quad 78.7460 \leq (22.5)(12), \quad 94.5438 \leq (16.8482)(16.3153)$$

4.5.3 General Iterative Method

The general idea behind iterative methods to solve $\mathbf{Ax} = \mathbf{b}$ is outlined as follows: Split the coefficient matrix as $\mathbf{A} = \mathbf{Q} - \mathbf{P}$, with the provision that \mathbf{Q} is non-singular so that \mathbf{Q}^{-1} exists, and substitute into $\mathbf{Ax} = \mathbf{b}$ to obtain

$$[\mathbf{Q} - \mathbf{P}]\mathbf{x} = \mathbf{b} \Rightarrow \mathbf{Qx} = \mathbf{Px} + \mathbf{b}$$

Of course, this system cannot be solved in its present form, as the solution vector \mathbf{x} appears on both sides. Instead, it will be solved by iterations. Choose an initial vector $\mathbf{x}^{(0)}$ and solve the following system for the new vector $\mathbf{x}^{(1)}$:

$$\mathbf{Qx}^{(1)} = \mathbf{Px}^{(0)} + \mathbf{b}$$

Next, use $\mathbf{x}^{(1)}$ to find the new vector $\mathbf{x}^{(2)}$:

$$\mathbf{Qx}^{(2)} = \mathbf{Px}^{(1)} + \mathbf{b}$$

and so on. In general, a sequence of vectors is generated via

$$\mathbf{Qx}^{(k+1)} = \mathbf{Px}^{(k)} + \mathbf{b}, \quad k = 0, 1, 2, \dots \quad (4.20)$$

Since \mathbf{Q} is assumed non-singular, Equation 4.20 is easily solved at each step for the updated vector $\mathbf{x}^{(k+1)}$, as

$$\mathbf{x}^{(k+1)} = \mathbf{Q}^{-1}\mathbf{Px}^{(k)} + \mathbf{Q}^{-1}\mathbf{b}, \quad k = 0, 1, 2, \dots \quad (4.21)$$

In the general procedure, splitting of \mathbf{A} is arbitrary, except that \mathbf{Q} must be non-singular. This arbitrary nature of the split causes the procedure to be generally ineffective. In specific iterative methods presented shortly, matrices \mathbf{P} and \mathbf{Q} obey very specific formats for successful implementation.

4.5.3.1 Convergence of the General Iterative Method

The sequence of vectors obtained through Equation 4.21 converges if the sequence of error vectors associated the iteration steps approaches the zero vector. The error vector at iteration k is defined as

$$\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}_a, \quad \mathbf{x}_a = \text{Actual solution vector}$$

Note that the actual solution \mathbf{x}_a is unknown, and the notation is being used in the analysis merely for the development of the important theoretical results. That said, since \mathbf{x}_a is the actual solution of $\mathbf{Ax} = \mathbf{b}$, then

$$\mathbf{A}\mathbf{x}_a = \mathbf{b} \quad \Rightarrow \quad [\mathbf{Q} - \mathbf{P}]\mathbf{x}_a = \mathbf{b}$$

Inserting this into Equation 4.20, yields

$$\mathbf{Q}\mathbf{x}^{(k+1)} = \mathbf{P}\mathbf{x}^{(k)} + [\mathbf{Q} - \mathbf{P}]\mathbf{x}_a \quad \Rightarrow \quad \mathbf{Q}[\mathbf{x}^{(k+1)} - \mathbf{x}_a] = \mathbf{P}[\mathbf{x}^{(k)} - \mathbf{x}_a]$$

Noting that the bracketed quantities are simply the error vectors at iteration k and $k + 1$, the above is written as

$$\mathbf{Q}\mathbf{e}^{(k+1)} = \mathbf{P}\mathbf{e}^{(k)}$$

Pre-multiplication of this equation by \mathbf{Q}^{-1} , and letting $\mathbf{M} = \mathbf{Q}^{-1}\mathbf{P}$, results in

$$\mathbf{e}^{(k+1)} = \mathbf{Q}^{-1}\mathbf{P}\mathbf{e}^{(k)} = \mathbf{M}\mathbf{e}^{(k)}, \quad k = 0, 1, 2, \dots$$

so that

$$\mathbf{e}^{(1)} = \mathbf{M}\mathbf{e}^{(0)}, \quad \mathbf{e}^{(2)} = \mathbf{M}\mathbf{e}^{(1)} = \mathbf{M}^2\mathbf{e}^{(0)}, \dots, \quad \mathbf{e}^{(k)} = \mathbf{M}^k\mathbf{e}^{(0)}$$

Taking the infinite-norm of both sides of the last equation and k applications of the second compatibility relation in Equation 4.19, we find

$$\|\mathbf{e}^{(k)}\|_{\infty} \leq \|\mathbf{M}\|_{\infty}^k \|\mathbf{e}^{(0)}\|_{\infty}$$

Thus, a sufficient condition for $\|\mathbf{e}^{(k)}\|_{\infty} \rightarrow 0$ as $k \rightarrow \infty$ is that $\|\mathbf{M}\|_{\infty}^k \rightarrow 0$ as $k \rightarrow \infty$, which is satisfied if $\|\mathbf{M}\|_{\infty} < 1$. The matrix $\mathbf{M} = \mathbf{Q}^{-1}\mathbf{P}$ plays a key role in the convergence of iterative schemes. The above analysis suggests that in splitting matrix \mathbf{A} , matrices \mathbf{Q} and \mathbf{P} must be selected so that the infinite norm of $\mathbf{M} = \mathbf{Q}^{-1}\mathbf{P}$ is less than one. We note that $\|\mathbf{M}\|_{\infty} < 1$ is only a sufficient condition and not necessary. This means that if it holds, the iteration converges, but if it does not hold, convergence is not automatically ruled out.

4.5.4 Jacobi Iteration Method

Let \mathbf{D} , \mathbf{L} , and \mathbf{U} be the diagonal, lower, and upper triangular portions of matrix $\mathbf{A} = [a_{ij}]_{n \times n}$, respectively, that is,

$$\mathbf{D} = \begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \dots & \\ & & & a_{nn} \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 0 & 0 & \dots & 0 \\ a_{21} & 0 & & 0 \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & 0 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & & a_{2n} \\ \dots & & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix}$$

In the Jacobi method, \mathbf{A} is split as

$$\begin{aligned} \mathbf{A} &= \mathbf{Q} - \mathbf{P} \\ &= \mathbf{D} + [\mathbf{L} + \mathbf{U}] \end{aligned} \quad \text{so that} \quad \begin{aligned} \mathbf{Q} &= \mathbf{D} \\ \mathbf{P} &= -[\mathbf{L} + \mathbf{U}] \end{aligned}$$

Subsequently, Equation 4.20 takes the specific form

$$\mathbf{D}\mathbf{x}^{(k+1)} = -[\mathbf{L} + \mathbf{U}]\mathbf{x}^{(k)} + \mathbf{b}, \quad k = 0, 1, 2, \dots \quad (4.22)$$

For \mathbf{D}^{-1} to exist, the diagonal entries of \mathbf{D} , and hence of \mathbf{A} , must all be nonzero. If a zero entry appears in a diagonal slot, the responsible equation in the original system must be switched with another equation so that no zero entry appears along the diagonal in the resulting coefficient matrix. Then, pre-multiplication of Equation 4.22 by \mathbf{D}^{-1} , yields

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1} \left\{ -[\mathbf{L} + \mathbf{U}]\mathbf{x}^{(k)} + \mathbf{b} \right\}, \quad k = 0, 1, 2, \dots \quad (4.23)$$

known as the Jacobi method. Note that $\mathbf{L} + \mathbf{U}$ is exactly matrix \mathbf{A} but with zero diagonal entries, and that the diagonal entries of \mathbf{D}^{-1} are $1/a_{ij}$ for $i = 1, 2, \dots, n$. Denoting the vector generated at the k th iteration by $\mathbf{x}^{(k)} = [x_1^{(k)} \dots x_n^{(k)}]^T$, Equation 4.23 can be expressed component-wise, as

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left\{ - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} x_j^{(k)} + b_i \right\}, \quad i = 1, 2, \dots, n \quad (4.24)$$

The important matrix $\mathbf{M} = \mathbf{Q}^{-1}\mathbf{P}$ takes the special form

$$\mathbf{M}_J = -\mathbf{D}^{-1}[\mathbf{L} + \mathbf{U}]$$

and is called the Jacobi iteration matrix. A sufficient condition for Jacobi iteration to converge is that $\|\mathbf{M}_J\|_\infty < 1$.

4.5.4.1 Convergence of the Jacobi Iteration Method

Convergence of the Jacobi method relies on a special class of matrices known as diagonally dominant. An $n \times n$ matrix \mathbf{A} is diagonally dominant if in each row, the absolute value of the diagonal entry is greater than the sum of the absolute values of all the off-diagonal entries, that is,

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|, \quad i = 1, 2, \dots, n \quad (4.25)$$

or equivalently,

$$\sum_{\substack{j=1 \\ j \neq i}}^n \frac{|a_{ij}|}{|a_{ii}|} < 1, \quad i = 1, 2, \dots, n \quad (4.26)$$

Theorem 4.1: Convergence of Jacobi Iteration

Let \mathbf{A} be diagonally dominant. Then, the linear system $\mathbf{Ax} = \mathbf{b}$ has a unique solution \mathbf{x}_a , and the sequence of vectors generated by Jacobi iteration, Equation 4.23, converges to \mathbf{x}_a regardless of the choice of the initial vector $\mathbf{x}^{(0)}$.

Proof

The Jacobi iteration matrix is formed as

$$\mathbf{M}_J = -\mathbf{D}^{-1}[\mathbf{L} + \mathbf{U}] = \begin{bmatrix} 0 & a_{12}/a_{11} & \dots & \dots & a_{1n}/a_{11} \\ a_{21}/a_{22} & 0 & \dots & \dots & a_{2n}/a_{22} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & a_{n-1,n}/a_{n-1,n-1} \\ a_{n1}/a_{nn} & a_{n2}/a_{nn} & \dots & \dots & 0 \end{bmatrix}$$

Since \mathbf{A} is diagonally dominant, Equation 4.26 holds. In each row of \mathbf{M}_J , the sum of the magnitudes of all entries is less than 1 by Equation 4.26. This means the row-sum norm of \mathbf{M}_J is less than 1, that is, $\|\mathbf{M}_J\|_\infty < 1$. Since this is a sufficient condition for convergence of Jacobi method, the proof is complete. ■

The user-defined function `Jacobi` uses the Jacobi iteration method to solve the linear system $\mathbf{Ax} = \mathbf{b}$, and returns the approximate solution vector, the number of iterations needed for convergence, and $\|\mathbf{M}_J\|_\infty$. The terminating condition is $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| < \epsilon$ for a prescribed tolerance ϵ .

```
function [x, k, MJnorm] = Jacobi(A, b, x0, tol, kmax)
%
% Jacobi uses the Jacobi iteration method to approximate the solution
% of Ax = b.
%
% [x, k, MJnorm] = Jacobi(A, b, x0, tol, kmax), where
%
% A is the n-by-n coefficient matrix,
% b is the n-by-1 right-hand side vector,
% x0 is the n-by-1 initial vector (default zeros),
% tol is the scalar tolerance for convergence (default 1e-4),
% kmax is the maximum number of iterations (default 100),
%
% x is the n-by-1 solution vector,
% k is the number of iterations required for convergence,
```

```

% MJnorm is the infinite norm of the Jacobi iteration matrix.
%
if nargin < 3 || isempty(x0), x0 = zeros(size(b)); end
if nargin < 4 || isempty(tol), tol = 1e-4; end
if nargin < 5 || isempty(kmax), kmax = 100; end
x(:, 1) = x0;
D = diag(diag(A)); At = A - D;
L = tril(At); U = triu(At);
% Norm of Jacobi iteration matrix
M = -D\(L + U); MJnorm = norm(M, inf); B = D\b;
% Perform iterations up to kmax
for k = 1:kmax,
    x(:, k+1) = M*x(:, k) + B; % Compute next approximation
    if norm(x(:, k+1) - x(:, k)) < tol, break; end % Check convergence
end
x = x(:, end);

```

EXAMPLE 4.13: JACOBI ITERATION

Consider the linear system

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} 4 & 1 & -1 \\ -2 & 5 & 0 \\ 2 & 1 & 6 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} 1 \\ -7 \\ 13 \end{Bmatrix}, \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 1 \\ 1 \end{Bmatrix}$$

Initial vector

1. Find $\mathbf{x}^{(1)}$ using both forms of the Jacobi method in Equations 4.23 and 4.24, and confirm the results by executing the user-defined function `Jacobi`.
2. Solve the system by executing the user-defined function `Jacobi` with default values for `tol` and `kmax`.

Solution

1. It is readily verified that the coefficient matrix \mathbf{A} is diagonally dominant since

$$4 > 1 + |-1|, \quad 5 > |-2|, \quad 6 > 2 + 1$$

This implies Theorem 4.1 guarantees convergence of the sequence of vectors generated by Jacobi iteration method to the actual solution. We will find the components of the next vector $\mathbf{x}^{(1)}$ generated by Jacobi iteration using Equations 4.23 and 4.24. Before using Equation 4.23, we first form

$$\mathbf{D} = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 6 \end{bmatrix}, \quad \mathbf{L} + \mathbf{U} = \begin{bmatrix} 0 & 1 & -1 \\ -2 & 0 & 0 \\ 2 & 1 & 0 \end{bmatrix}$$

Then, Equation 4.23 with $k = 0$ yields

$$\mathbf{x}^{(1)} = \mathbf{D}^{-1} \left\{ -[\mathbf{L} + \mathbf{U}]\mathbf{x}^{(0)} + \mathbf{b} \right\} = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ 0 & \frac{1}{5} & 0 \\ 0 & 0 & \frac{1}{6} \end{bmatrix} \left\{ - \begin{bmatrix} 0 & 1 & -1 \\ -2 & 0 & 0 \\ 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ -7 \\ 13 \end{bmatrix} \right\} = \begin{bmatrix} 0.25 \\ -1.4 \\ 2 \end{bmatrix}$$

The vector $\mathbf{x}^{(1)}$ can also be found by using Equation 4.24 with $k = 0$,

$$x_i^{(1)} = \frac{1}{a_{ii}} \left\{ - \sum_{\substack{j=1 \\ j \neq i}}^3 a_{ij} x_j^{(0)} + b_i \right\}, \quad i = 1, 2, 3$$

Specifically,

$$x_1^{(1)} = \frac{1}{a_{11}} \left\{ - [a_{12}x_2^{(0)} + a_{13}x_3^{(0)}] + b_1 \right\} = \frac{1}{4} \left\{ - [(1)(1) + (-1)(1)] + 1 \right\} = 0.25$$

$$x_2^{(1)} = \frac{1}{a_{22}} \left\{ - [a_{21}x_1^{(0)} + a_{23}x_3^{(0)}] + b_2 \right\} = \frac{1}{5} \left\{ - [(-2)(0) + (0)(1)] + (-7) \right\} = -1.4$$

$$x_3^{(1)} = \frac{1}{a_{33}} \left\{ - [a_{31}x_1^{(0)} + a_{32}x_2^{(0)}] + b_3 \right\} = \frac{1}{6} \left\{ - [(2)(0) + (1)(1)] + 13 \right\} = 2$$

Therefore,

$$\mathbf{x}^{(1)} = \begin{Bmatrix} 0.25 \\ -1.4 \\ 2 \end{Bmatrix}$$

The vector $\mathbf{x}^{(1)}$ can be verified by executing the user-defined function `Jacobi` with `kmax = 1` to allow one iteration only.

```
>> A = [4 1 -1; -2 5 0; 2 1 6]; b = [1; -7; 13]; x0 = [0; 1; 1];
>> x = Jacobi(A, b, x0, [], 1)
```

```
x =
    0.2500
   -1.4000
    2.0000           % Agrees with hand calculations
```

2.

```
>> [x, k, MJnorm] = Jacobi(A, b, x0) % Default values for tol and kmax
```

```
x =
    1.0000
   -1.0000
    2.0000
```

```
k =
    13
```

```
MJnorm =
    0.5000
```

Note that the condition $\|\mathbf{M}\|_{\infty} < 1$ is satisfied because the coefficient matrix \mathbf{A} is diagonally dominant.

4.5.5 Gauss–Seidel Iteration Method

Based on Equations 4.23 and 4.24, every component of $\mathbf{x}^{(k+1)}$ is calculated entirely from $\mathbf{x}^{(k)}$ of the previous iteration. In other words, to access $\mathbf{x}^{(k+1)}$, the k th iteration has to be completed

so that $\mathbf{x}^{(k)}$ is entirely available. Performance of Jacobi iteration can be improved if the most updated components of a vector are utilized, as soon as they are available, to compute the subsequent components of the same vector. Consider two successive vectors, as well as the actual solution,

$$\mathbf{x}^{(k)} = \begin{Bmatrix} x_1^{(k)} \\ \dots \\ x_p^{(k)} \\ x_{p+1}^{(k)} \\ \dots \\ x_n^{(k)} \end{Bmatrix}, \quad \mathbf{x}^{(k+1)} = \begin{Bmatrix} x_1^{(k+1)} \\ \dots \\ x_p^{(k+1)} \\ x_{p+1}^{(k+1)} \\ \dots \\ x_n^{(k+1)} \end{Bmatrix}, \quad \mathbf{x}_a = \begin{Bmatrix} x_1 \\ \dots \\ x_p \\ x_{p+1} \\ \dots \\ x_n \end{Bmatrix}$$

Generally, $x_p^{(k+1)}$ is expected to be a better estimate of x_p than $x_p^{(k)}$ is. And as such, using $x_p^{(k+1)}$ instead of $x_p^{(k)}$ should lead to a better approximation of the next component, $x_{p+1}^{(k+1)}$, in the current vector. This is the reasoning behind Gauss–Seidel iteration method, which is considered a *refinement* of Jacobi method. To fulfill this logic, the coefficient matrix \mathbf{A} is split as

$$\begin{aligned} \mathbf{A} &= \mathbf{Q} - \mathbf{P} \\ &= [\mathbf{D} + \mathbf{L}] + \mathbf{U} \end{aligned} \quad \text{so that} \quad \begin{aligned} \mathbf{Q} &= \mathbf{D} + \mathbf{L} \\ \mathbf{P} &= -\mathbf{U} \end{aligned}$$

As a result, Equation 4.20 takes the specific form

$$[\mathbf{D} + \mathbf{L}]\mathbf{x}^{(k+1)} = -\mathbf{U}\mathbf{x}^{(k)} + \mathbf{b}, \quad k = 0, 1, 2, \dots \quad (4.27)$$

But, $\mathbf{D} + \mathbf{L}$ is a lower-triangular matrix whose diagonal entries are those of \mathbf{A} . Thus, $[\mathbf{D} + \mathbf{L}]^{-1}$ exists if \mathbf{A} has nonzero diagonal entries. If a diagonal entry is zero, the responsible equation in the original system must be switched with another equation so that no zero entry appears along the diagonal in the resulting coefficient matrix. Subsequently, pre-multiplication of Equation 4.27 by $[\mathbf{D} + \mathbf{L}]^{-1}$ yields

$$\mathbf{x}^{(k+1)} = [\mathbf{D} + \mathbf{L}]^{-1} \{-\mathbf{U}\mathbf{x}^{(k)} + \mathbf{b}\}, \quad k = 0, 1, 2, \dots \quad (4.28)$$

known as the Gauss–Seidel iteration method. Denoting the vector at the k th iteration by $\mathbf{x}^{(k)} = [x_1^{(k)} \dots x_n^{(k)}]^T$, Equation 4.28 can be expressed component wise, as

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left\{ -\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} + b_i \right\}, \quad i = 1, 2, \dots, n \quad (4.29)$$

where the first sum on the right side is considered zero when $i = 1$. The important matrix $\mathbf{M} = \mathbf{Q}^{-1}\mathbf{P}$ now takes the special form

$$\mathbf{M}_{GS} = -[\mathbf{D} + \mathbf{L}]^{-1} \mathbf{U}$$

known as the Gauss–Seidel iteration matrix. A sufficient condition for the Gauss–Seidel iteration to converge is $\|\mathbf{M}_{\text{GS}}\|_{\infty} < 1$.

4.5.5.1 Convergence of the Gauss–Seidel Iteration Method

Since the Gauss–Seidel method is a refinement of the Jacobi method, it converges whenever the Jacobi method does, and usually faster. Recall that if \mathbf{A} is diagonally dominant, the Jacobi iteration is guaranteed to converge to the solution vector. This implies that if \mathbf{A} is diagonally dominant, the Gauss–Seidel iteration is also guaranteed to converge, and faster than the Jacobi.

If \mathbf{A} is not diagonally dominant, the convergence of the Gauss–Seidel method relies on another special class of matrices known as symmetric, positive definite (Section 4.4).

Theorem 4.2: Convergence of Gauss–Seidel Iteration

Let \mathbf{A} be symmetric, positive definite. Then, the linear system $\mathbf{Ax} = \mathbf{b}$ has a unique solution \mathbf{x}_a , and the sequence of vectors generated by the Gauss–Seidel iteration, Equation 4.28, converges to \mathbf{x}_a regardless of the choice of the initial vector $\mathbf{x}^{(0)}$. ■

The user-defined function `GaussSeidel` uses the Gauss–Seidel iteration method to solve the linear system $\mathbf{Ax} = \mathbf{b}$, and returns the approximate solution vector, the number of iterations needed for convergence, and $\|\mathbf{M}_{\text{GS}}\|_{\infty}$. The terminating condition is $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| < \epsilon$ for a prescribed tolerance ϵ .

```
function [x, k, MGSnorm] = GaussSeidel(A, b, x0, tol, kmax)
%
% GaussSeidel uses the Gauss-Seidel iteration method to approximate the
% solution of Ax = b.
%
% [x, k, MGSnorm] = GaussSeidel(A, b, x0, tol, kmax), where
%
% A is the n-by-n coefficient matrix,
% b is the n-by-1 right-hand side vector,
% x0 is the n-by-1 initial vector (default zeros),
% tol is the scalar tolerance for convergence (default 1e-4),
% kmax is the maximum number of iterations (default 100),
%
% x is the n-by-1 solution vector,
% k is the number of iterations required for convergence,
% MGSnorm is the infinite norm of the Gauss-Seidel iteration matrix.
%
if nargin < 3 || isempty(x0), x0 = zeros(size(b)); end
if nargin < 4 || isempty(tol), tol = 1e-4; end
if nargin < 5 || isempty(kmax), kmax = 100; end
x(:, 1) = x0;
D = diag(diag(A)); At = A - D;
L = tril(At); U = At - L;
% Norm of Gauss-Seidel iteration matrix
M = -(D + L)\U; MGSnorm = norm(M, inf); B = (D + L)\b;
% Perform iterations up to kmax
```

```

for k = 1:kmax,
    x(:, k+1) = M*x(:, k) + B;
    if norm(x(:,k+1)-x(:, k)) < tol,
        break
    end
end
x = x(:, end);

```

EXAMPLE 4.14: GAUSS–SEIDEL ITERATION: DIAGONALLY DOMINANT COEFFICIENT MATRIX

Consider the linear system of Example 4.13,

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} 4 & 1 & -1 \\ -2 & 5 & 0 \\ 2 & 1 & 6 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} 1 \\ -7 \\ 13 \end{Bmatrix}, \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 1 \\ 1 \end{Bmatrix}$$

Initial vector

1. Find $\mathbf{x}^{(1)}$ using both forms of the Gauss–Seidel method in Equations 4.28 and 4.29, and confirm the results by executing the user-defined function GaussSeidel.
2. Solve the system by executing the user-defined function GaussSeidel with default values for `tol` and `kmax`.

Solution

1. We will find the components of the next vector $\mathbf{x}^{(1)}$ generated by the Gauss–Seidel iteration using Equations 4.28 and 4.29. Before using Equation 4.28, we first form

$$\mathbf{D} + \mathbf{L} = \begin{bmatrix} 4 & 0 & 0 \\ -2 & 5 & 0 \\ 2 & 1 & 6 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 0 & 1 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Equation 4.28 with $k = 0$ yields

$$\mathbf{x}^{(1)} = [\mathbf{D} + \mathbf{L}]^{-1} \{-\mathbf{U}\mathbf{x}^{(0)} + \mathbf{b}\} = \frac{1}{120} \begin{bmatrix} 30 & 0 & 0 \\ 12 & 24 & 0 \\ -12 & -4 & 20 \end{bmatrix} \left\{ -\begin{bmatrix} 0 & 1 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ -7 \\ 13 \end{bmatrix} \right\} = \begin{Bmatrix} 0.25 \\ -1.3 \\ 2.3 \end{Bmatrix}$$

The vector $\mathbf{x}^{(1)}$ may also be found by using Equation 4.29 with $k = 0$,

$$x_i^{(1)} = \frac{1}{a_{ii}} \left\{ -\sum_{j=1}^{i-1} a_{ij}x_j^{(1)} - \sum_{j=i+1}^3 a_{ij}x_j^{(0)} + b_i \right\}, \quad i = 1, 2, 3$$

As previously mentioned, the first sum on the right side is considered zero when $i = 1$. For the problem at hand,

$$x_1^{(1)} = \frac{1}{a_{11}}[-a_{12}x_2^{(0)} - a_{13}x_3^{(0)} + b_1] = \frac{1}{4}[(-1)(1) - (-1)(1) + 1] = 0.25$$

$$x_2^{(1)} = \frac{1}{a_{22}}[-a_{21}x_1^{(1)} - a_{23}x_3^{(0)} + b_2] = \frac{1}{5}[(-2)(0.25) - (0)(1) - 7] = -1.3$$

$$x_3^{(1)} = \frac{1}{a_{33}}[-a_{31}x_1^{(1)} - a_{32}x_2^{(1)} + b_3] = \frac{1}{6}[(-2)(0.25) - (1)(-1.3) + 13] = 2.3$$

Therefore,

$$\mathbf{x}^{(1)} = \begin{Bmatrix} 0.25 \\ -1.3 \\ 2.3 \end{Bmatrix}$$

This vector can be verified by executing GaussSeidel with kmax=1 so that one iteration only is performed.

```
>> A = [4 1 -1; -2 5 0; 2 1 6]; b = [1; -7; 13]; x0 = [0; 1; 1];
>> x = GaussSeidel(A, b, x0, [], 1)

x =
    0.2500
   -1.3000
    2.3000      % Agrees with hand calculations
```

2. Since \mathbf{A} is diagonally dominant, the Gauss–Seidel iteration is guaranteed to converge because the Jacobi method is guaranteed to converge.

```
>> [x, k, MGSnorm] = GaussSeidel(A, b, x0)

x =
    1.0000
   -1.0000
    2.0000

k =
     8

MGSnorm =
    0.5000
```

As expected, Gauss–Seidel exhibits a faster convergence (eight iterations) than Jacobi (13 iterations).

EXAMPLE 4.15: GAUSS–SEIDEL ITERATION: SYMMETRIC, POSITIVE-DEFINITE COEFFICIENT MATRIX

Consider

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} 1 & 1 & -2 \\ 1 & 10 & 4 \\ -2 & 4 & 24 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} 5.5 \\ 17.5 \\ -19 \end{Bmatrix}, \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix}$$

Initial vector

The coefficient matrix is symmetric since $\mathbf{A} = \mathbf{A}^T$. It is also positive definite because

$$1 > 0, \quad \begin{vmatrix} 1 & 1 \\ 1 & 10 \end{vmatrix} = 9 > 0, \quad \begin{vmatrix} 1 & 1 & -2 \\ 1 & 10 & 4 \\ -2 & 4 & 24 \end{vmatrix} = 144 > 0$$

Thus, the Gauss–Seidel iteration will converge to the solution vector for any initial vector. Executing the user-defined function `GaussSeidel` with default values for `tol` and `kmax`, we find

```
>> A = [1 1 -2;1 10 4;-2 4 24]; b = [5.5;17.5;-19];
>> [x, k, MGSnorm] = GaussSeidel(A, b)

x =
    1.5000
    2.0000
   -1.0000

k =
    15

MGSnorm =
     3
```

The input argument `x0` was left out because the initial vector happens to be the zero vector, which agrees with the default. Also note that $\|\mathbf{M}_{\text{GS}}\| = 3 > 1$ even though iterations did converge. This is because the condition $\|\mathbf{M}_{\text{GS}}\|_{\infty} < 1$ is only sufficient and not necessary for convergence of the Gauss–Seidel iteration. Also note that unlike the fact that a diagonally dominant coefficient matrix guarantees $\|\mathbf{M}_J\|_{\infty} < 1$, a symmetric, positive definite coefficient matrix does not guarantee $\|\mathbf{M}_{\text{GS}}\|_{\infty} < 1$, but does guarantee convergence for the Gauss–Seidel method.

4.5.6 Indirect Methods versus Direct Methods for Large Systems

Indirect methods such as Gauss–Seidel are commonly used for large linear systems $\mathbf{Ax} = \mathbf{b}$. Suppose a large system is being solved by the general iterative method, Equation 4.21,

$$\mathbf{x}^{(k+1)} = \mathbf{Q}^{-1}\mathbf{P}\mathbf{x}^{(k)} + \mathbf{Q}^{-1}\mathbf{b}, \quad k = 0, 1, 2, \dots$$

and that convergence is observed after m iterations. Because each iteration requires roughly n^2 multiplications, a total of n^2m multiplications are performed by the time convergence is achieved. On the other hand, a direct method such as Gauss elimination requires $\frac{1}{3}n^3$ multiplications to find the solution. Therefore, an indirect method is superior to a direct method as long as

$$n^2m < \frac{1}{3}n^3 \quad \Rightarrow \quad m < \frac{1}{3}n$$

For example, for a 100×100 system, this yields $m < \frac{1}{3}(100)$ so that an iterative method is preferred as long as it converges within 33 iterations. In many physical applications, not only the coefficient matrix \mathbf{A} is large, it is also sparse, that is, it contains a large number of zero entries. As one example, consider the numerical solution of partial differential equations using the finite-differences method (Chapter 10). In these cases, we encounter a large, sparse system where the coefficient matrix has at most five nonzero entries in each row. Therefore, based on Equations 4.24 and/or 4.29, six multiplications must be performed to find each component $x_i^{(k+1)}$ of the generated vector. But each vector has n components, thus a total of $6n$ multiplications per iteration are performed. If it takes m iterations for convergence, then a total of $6nm$ multiplications are required for the indirect method. Therefore, the indirect method is computationally more efficient than a direct method as long as

$$6nm < \frac{1}{3}n^3 \quad \Rightarrow \quad m < \frac{1}{18}n^2$$

For a 1000×1000 system with a sparse coefficient matrix, this translates to $m < \frac{1}{18}(1000)^2$ so that an iterative method such as Gauss–Seidel is superior to Gauss elimination if it converges within 55,556 iterations, which is quite likely.

4.6 Ill-Conditioning and Error Analysis

So far this chapter has focused on methods to find the solution vector for linear systems in the form $\mathbf{Ax} = \mathbf{b}$. In this section we study the conditioning of a linear system and how it may impact the error associated with a computed solution.

4.6.1 Condition Number

The condition number of a non-singular matrix $\mathbf{A}_{n \times n}$ is defined as

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\| \quad (4.30)$$

where the same matrix norm is used for both \mathbf{A} and \mathbf{A}^{-1} . It can be shown that for any $\mathbf{A}_{n \times n}$

$$\kappa(\mathbf{A}) \geq 1$$

It turns out that the smaller the condition number of a matrix, the better the condition of the matrix. A useful measure of the condition of a matrix is provided by the ratio of the largest (magnitude) to the smallest (magnitude) eigenvalue of the matrix.

EXAMPLE 4.16: CONDITION NUMBER

Calculate the condition number of the following matrix using all three norms, and verify the results using the MATLAB built-in command `cond`.

$$\mathbf{A} = \begin{bmatrix} 6 & 4 & 3 \\ 4 & 3 & 2 \\ 3 & 4 & 2 \end{bmatrix}$$

Solution

The inverse is found as

$$\mathbf{A}^{-1} = \begin{bmatrix} -2 & 4 & -1 \\ -2 & 3 & 0 \\ 7 & -12 & 2 \end{bmatrix}$$

Then,

$$\|\mathbf{A}\|_1 = 13, \quad \|\mathbf{A}^{-1}\|_1 = 19 \Rightarrow \kappa(\mathbf{A}) = 247$$

$$\|\mathbf{A}\|_\infty = 13, \quad \|\mathbf{A}^{-1}\|_\infty = 21 \Rightarrow \kappa(\mathbf{A}) = 273$$

$$\|\mathbf{A}\|_E = 10.9087, \quad \|\mathbf{A}^{-1}\|_E = 15.1987 \Rightarrow \kappa(\mathbf{A}) = 165.7981$$

In MATLAB, `cond(A, P)` returns the condition number of matrix A in P -norm.

```
>> A = [6 4 3; 4 3 2; 3 4 2];
>> [cond(A,1), cond(A,inf), cond(A,'fro')] % Using three different matrix norms

ans =
    247.0000    273.0000    165.7981
```

Note that all three returned values are of the same order of magnitude, regardless of the choice of norm used.

4.6.2 Ill-Conditioning

The system $Ax = b$ is said to be well-conditioned if small errors generated during the solution process, or small changes in the coefficients, have small effects on the solution. For instance, if the diagonal entries of A are much larger in magnitude than the off-diagonal ones, the system is well-conditioned. If small errors and changes during the solution process have large impacts on the solution, the system is ill-conditioned. Ill-conditioned systems often arise in areas such as statistical analysis and least-squares fits (Chapter 5).

EXAMPLE 4.17: ILL-CONDITIONING

Investigate the ill-conditioning of

$$Ax = b, \quad A = \begin{bmatrix} 1 & -2 \\ 1.0001 & -1.9998 \end{bmatrix}, \quad b = \begin{Bmatrix} 2 \\ 2 \end{Bmatrix}$$

Solution

The actual solution of this system can be easily verified to be

$$x_a = \begin{Bmatrix} 1 \\ -0.5 \end{Bmatrix}$$

Suppose the first component of vector b is slightly perturbed by a very small $\epsilon > 0$ so that the new vector is

$$\tilde{b} = \begin{Bmatrix} 2 + \epsilon \\ 2 \end{Bmatrix}$$

The ensuing system $A\tilde{x} = \tilde{b}$ is then solved via Gauss elimination, as

$$\left[\begin{array}{cc|c} 1 & -2 & 2 + \epsilon \\ 1.0001 & -1.9998 & 2 \end{array} \right] \xrightarrow{-1.0001(\text{row}_1) + \text{row}_2} \left[\begin{array}{cc|c} 1 & -2 & 2 + \epsilon \\ 0 & 0.0004 & -0.0002 - 1.0001\epsilon \end{array} \right] \rightarrow \left[\begin{array}{cc|c} 1 & -2 & 2 + \epsilon \\ 0 & 1 & -0.5 - 2500.25\epsilon \end{array} \right]$$

so that

$$\tilde{x} = \begin{Bmatrix} 1 - 4999.50\epsilon \\ -0.5 - 2500.25\epsilon \end{Bmatrix} = \begin{Bmatrix} 1 \\ -0.5 \end{Bmatrix} - \begin{Bmatrix} 4999.5 \\ 2500.25 \end{Bmatrix} \epsilon = x_a - \begin{Bmatrix} 4999.5 \\ 2500.25 \end{Bmatrix} \epsilon$$

Therefore, even though one of the components of \mathbf{b} was subjected to a very small change of ϵ , the resulting solution vector shows very large relative changes in its components. This indicates that the system is ill-conditioned.

4.6.2.1 Indicators of Ill-Conditioning

There are essentially three indicators of ill-conditioning for a linear system $\mathbf{Ax} = \mathbf{b}$:

1. $\det(\mathbf{A})$ is very small in absolute value relative to the largest entries of \mathbf{A} and \mathbf{b} in absolute value.
2. The entries of \mathbf{A}^{-1} are large in absolute value relative to the components of the solution vector.
3. $\kappa(\mathbf{A})$ is very large.

EXAMPLE 4.18: INDICATORS OF ILL-CONDITIONING

Consider the system in Example 4.17:

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} 1 & -2 \\ 1.0001 & -1.9998 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} 2 \\ 2 \end{Bmatrix}$$

We will see that this system is ill-conditioned by verifying all three indicators listed above.

1. $\det(\mathbf{A}) = 0.0004$, which is considerably smaller than the absolute values of entries of \mathbf{A} and \mathbf{b} .
2. The inverse of \mathbf{A} is found as

$$\mathbf{A}^{-1} = \begin{bmatrix} -4999.50 & 5000 \\ -2500.25 & 2500 \end{bmatrix}$$

The entries are very large in magnitude relative to the components of the solution vector

$$\mathbf{x}_a = \begin{Bmatrix} 1 \\ -0.5 \end{Bmatrix}.$$

3. Using the 1-norm, we find the condition number of \mathbf{A} as

$$\|\mathbf{A}\|_1 = 3.9998, \quad \|\mathbf{A}^{-1}\|_1 = 7500 \Rightarrow \kappa(\mathbf{A}) = 29,998.5$$

which is quite large.

4.6.3 Computational Error

Suppose \mathbf{x}_c is the computed solution of a linear system $\mathbf{Ax} = \mathbf{b}$, while \mathbf{x}_a represents the actual solution. Note that in practice the actual solution \mathbf{x}_a is not available and that the notation

is merely used to establish an important result concerning possible error bounds on the computed solution. The residual vector is defined as

$$\mathbf{r} = \mathbf{A}\mathbf{x}_c - \mathbf{b}$$

The norm of the residual vector $\|\mathbf{r}\|$ gives a measure of the accuracy of the computed solution, so does the absolute error defined by $\|\mathbf{x}_c - \mathbf{x}_a\|$. The most commonly used measure is the relative error

$$\frac{\|\mathbf{x}_c - \mathbf{x}_a\|}{\|\mathbf{x}_a\|}$$

Theorem 4.3: Relative Error Bounds

Let \mathbf{x}_a and \mathbf{x}_c be the actual and computed solutions of $\mathbf{A}\mathbf{x} = \mathbf{b}$, respectively. If $\mathbf{r} = \mathbf{A}\mathbf{x}_c - \mathbf{b}$ is the residual vector and $\kappa(\mathbf{A})$ is the condition number of \mathbf{A} , then

$$\frac{1}{\kappa(\mathbf{A})} \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} \leq \frac{\|\mathbf{x}_c - \mathbf{x}_a\|}{\|\mathbf{x}_a\|} \leq \kappa(\mathbf{A}) \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} \quad (4.31)$$

A selected matrix norm and its compatible vector norm must be used throughout.

Proof

We first write

$$\mathbf{r} = \mathbf{A}\mathbf{x}_c - \mathbf{b} = \mathbf{A}\mathbf{x}_c - \mathbf{A}\mathbf{x}_a = \mathbf{A}(\mathbf{x}_c - \mathbf{x}_a) \Rightarrow \mathbf{x}_c - \mathbf{x}_a = \mathbf{A}^{-1}\mathbf{r}$$

so that

$$\|\mathbf{x}_c - \mathbf{x}_a\| = \|\mathbf{A}^{-1}\mathbf{r}\| \leq \|\mathbf{A}^{-1}\| \|\mathbf{r}\| \xrightarrow[\text{by } \|\mathbf{x}_a\|]{\text{Divide both sides}} \frac{\|\mathbf{x}_c - \mathbf{x}_a\|}{\|\mathbf{x}_a\|} \leq \|\mathbf{A}^{-1}\| \frac{\|\mathbf{r}\|}{\|\mathbf{x}_a\|} \quad (4.32)$$

But by Equation 4.30,

$$\|\mathbf{A}^{-1}\| = \frac{\kappa(\mathbf{A})}{\|\mathbf{A}\|}$$

and

$$\mathbf{b} = \mathbf{A}\mathbf{x}_a \Rightarrow \|\mathbf{b}\| \leq \|\mathbf{A}\| \|\mathbf{x}_a\| \Rightarrow \frac{1}{\|\mathbf{x}_a\|} \leq \frac{\|\mathbf{A}\|}{\|\mathbf{b}\|}$$

Inserting these into Equation 4.32, yields

$$\frac{\|\mathbf{x}_c - \mathbf{x}_a\|}{\|\mathbf{x}_a\|} \leq \kappa(\mathbf{A}) \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$$

which establishes the upper bound for relative error. To derive the lower bound, we first note that

$$\mathbf{r} = \mathbf{A}\mathbf{x}_c - \mathbf{A}\mathbf{x}_a \Rightarrow \|\mathbf{r}\| \leq \|\mathbf{A}\| \|\mathbf{x}_c - \mathbf{x}_a\| \xrightarrow[\|\mathbf{A}\| > 0 \text{ for any nonzero } \mathbf{A}]{\text{Divide by } \|\mathbf{A}\|} \|\mathbf{x}_c - \mathbf{x}_a\| \geq \frac{\|\mathbf{r}\|}{\|\mathbf{A}\|}$$

Also,

$$\mathbf{x}_a = \mathbf{A}^{-1}\mathbf{b} \Rightarrow \|\mathbf{x}_a\| \leq \|\mathbf{A}^{-1}\| \|\mathbf{b}\| \Rightarrow \frac{1}{\|\mathbf{x}_a\|} \geq \frac{1}{\|\mathbf{A}^{-1}\| \|\mathbf{b}\|}$$

Multiplication of the last two inequalities results in

$$\frac{\|\mathbf{x}_c - \mathbf{x}_a\|}{\|\mathbf{x}_a\|} \geq \frac{1}{\kappa(\mathbf{A})} \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$$

This completes the proof. ■

4.6.3.1 Consequences of Ill-Conditioning

Ill-conditioning has an immediate impact on the accuracy of the computed solution. Consider the relative error bounds given in Equation 4.31. For a computed solution, it is safe to assume that the norm of the residual vector $\|\mathbf{r}\|$ is relatively small compared to $\|\mathbf{b}\|$. A small value for $\kappa(\mathbf{A})$ raises the lower bound while lowering the upper bound, thus narrowing the interval for relative error. A large value for $\kappa(\mathbf{A})$, on the other hand, lowers the lower bound and raises the upper bound, thus widening the interval and allowing for a large relative error associated with the computed solution.

Another consequence of ill-conditioning is less conspicuous in the sense that a poor approximation of the actual solution vector may come with a very small residual vector norm. Once again, refer to the system in Examples 4.17 and 4.18, and consider

$$\hat{\mathbf{x}} = \begin{Bmatrix} 2 \\ 0.0002 \end{Bmatrix}$$

which is clearly a poor approximation of the actual solution

$$\mathbf{x}_a = \begin{Bmatrix} 1 \\ -0.5 \end{Bmatrix}$$

The corresponding residual vector is

$$\mathbf{r} = \mathbf{A}\hat{\mathbf{x}} - \mathbf{b} = \begin{bmatrix} 1 & -2 \\ 1.0001 & -1.9998 \end{bmatrix} \begin{Bmatrix} 2 \\ 0.0002 \end{Bmatrix} - \begin{Bmatrix} 2 \\ 2 \end{Bmatrix} = \begin{Bmatrix} -0.0004 \\ -0.0002 \end{Bmatrix}$$

Any one of the three vector norms returns a very small value for $\|\mathbf{x}\|$, incorrectly suggesting $\hat{\mathbf{x}}$ may be a valid solution.

4.6.4 Effects of Parameter Changes on the Solution

The following theorem illustrates how changes in the entries of \mathbf{A} or components of \mathbf{b} may affect the resulting solution vector \mathbf{x} , as well as the role of condition number of \mathbf{A} .

Theorem 4.4: Percent Change

Consider the linear system $\mathbf{Ax} = \mathbf{b}$. Let $\Delta\mathbf{A}$, $\Delta\mathbf{b}$, and $\Delta\mathbf{x}$ reflect the changes in the entries or components of \mathbf{A} , \mathbf{b} , and \mathbf{x} , respectively. Then

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} \kappa(\mathbf{A}) \quad (4.33)$$

and

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} \kappa(\mathbf{A}) \quad (4.34)$$

A selected matrix norm and its compatible vector norm must be used throughout.

Proof

Suppose entries of \mathbf{A} have been changed and these changes are recorded in matrix $\Delta\mathbf{A}$. As a result, the solution \mathbf{x} will also change, say, by $\Delta\mathbf{x}$. In $\mathbf{Ax} = \mathbf{b}$, insert $\mathbf{A} + \Delta\mathbf{A}$ for \mathbf{A} and $\mathbf{x} + \Delta\mathbf{x}$ for \mathbf{x} to obtain

$$\begin{aligned} (\mathbf{A} + \Delta\mathbf{A})(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} & \xrightarrow{\text{Expand}} \mathbf{Ax} + \mathbf{A}(\Delta\mathbf{x}) + [\Delta\mathbf{A}]\mathbf{x} + [\Delta\mathbf{A}](\Delta\mathbf{x}) = \mathbf{b} \\ & \xrightarrow[\text{from both sides}]{\text{Cancel } \mathbf{Ax}=\mathbf{b}} \mathbf{A}(\Delta\mathbf{x}) + [\Delta\mathbf{A}]\mathbf{x} + [\Delta\mathbf{A}](\Delta\mathbf{x}) = \mathbf{0} \end{aligned}$$

Solving for $\Delta\mathbf{x}$, we have

$$\mathbf{A}(\Delta\mathbf{x}) = -[\Delta\mathbf{A}](\mathbf{x} + \Delta\mathbf{x}) \Rightarrow \Delta\mathbf{x} = -\mathbf{A}^{-1}[\Delta\mathbf{A}](\mathbf{x} + \Delta\mathbf{x})$$

Taking the (vector) norm of both sides, and applying compatibility relations, Equation 4.19, twice, we have

$$\|\Delta\mathbf{x}\| = \|\mathbf{A}^{-1}[\Delta\mathbf{A}](\mathbf{x} + \Delta\mathbf{x})\| \leq \|\mathbf{A}^{-1}\| \|\Delta\mathbf{A}\| \|\mathbf{x} + \Delta\mathbf{x}\|$$

Inserting $\|\mathbf{A}^{-1}\| = \kappa(\mathbf{A})/\|\mathbf{A}\|$ in this last equation, and dividing both sides by $\|\mathbf{x} + \Delta\mathbf{x}\|$, yields

$$\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x} + \Delta \mathbf{x}\|} \leq \frac{\|\Delta \mathbf{A}\|}{\|\mathbf{A}\|} \kappa(\mathbf{A})$$

Since $\Delta \mathbf{x}$ represents small changes in \mathbf{x} , then

$$\|\mathbf{x} + \Delta \mathbf{x}\| \cong \|\mathbf{x}\| \quad \Rightarrow \quad \frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x} + \Delta \mathbf{x}\|} \cong \frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}\|}$$

Using this in the previous equation establishes Equation 4.33. In order to verify Equation 4.34, insert $\mathbf{b} + \Delta \mathbf{b}$ for \mathbf{b} , and $\mathbf{x} + \Delta \mathbf{x}$ for \mathbf{x} in $\mathbf{A}\mathbf{x} = \mathbf{b}$ and proceed as before. This completes the proof. ■

Equations 4.33 and 4.34 assert that if $\kappa(\mathbf{A})$ is small, then small percent changes in \mathbf{A} or \mathbf{b} will result in small percent changes in \mathbf{x} . This, of course, is in line with the previous findings in this section. Furthermore, Equations 4.33 and 4.34 only provide upper bounds and not estimates of the percent change in solution.

EXAMPLE 4.19: PERCENT CHANGE

Consider the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} 1 & 2 \\ 2 & 4.0001 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} -1 \\ -2.0002 \end{Bmatrix} \Rightarrow \underset{\text{solution}}{\mathbf{x}} = \begin{Bmatrix} 3 \\ -2 \end{Bmatrix}$$

Suppose the (1,2) entry of \mathbf{A} is reduced by 0.01 while its (2,1) entry is increased by 0.01 so that

$$\Delta \mathbf{A} = \begin{bmatrix} 0 & -0.01 \\ 0.01 & 0 \end{bmatrix} \xrightarrow{\text{New coefficient matrix}} \bar{\mathbf{A}} = \mathbf{A} + \Delta \mathbf{A} = \begin{bmatrix} 1 & 1.99 \\ 2.01 & 4.0001 \end{bmatrix}$$

Solving the new system yields

$$\bar{\mathbf{A}}\bar{\mathbf{x}} = \mathbf{b} \Rightarrow \bar{\mathbf{x}} = \begin{Bmatrix} -98.51 \\ 49 \end{Bmatrix} \quad \text{so that} \quad \Delta \mathbf{x} = \mathbf{x} - \bar{\mathbf{x}} = \begin{Bmatrix} 101.51 \\ -51 \end{Bmatrix}$$

From this point forward, the 1-norm will be used for all vectors and matrices. The condition number of \mathbf{A} is calculated as $\kappa(\mathbf{A}) = 3.6001 \times 10^5$ indicating ill-conditioning. The upper bound for the percent change in solution can be found as

$$\frac{\|\Delta \mathbf{x}\|_1}{\|\mathbf{x}\|_1} \leq \frac{\|\Delta \mathbf{A}\|_1}{\|\mathbf{A}\|_1} \kappa(\mathbf{A}) = \frac{0.01}{6.0001} (3.6001 \times 10^5) = 600.0100$$

The upper bound is rather large as a consequence of ill-conditioning. The actual percent change is calculated as

$$\frac{\|\Delta \mathbf{x}\|_1}{\|\mathbf{x}\|_1} = \frac{152.51}{5} = 30.5020$$

which is much smaller than the upper bound offered by Equation 4.33, thus asserting that the upper bound is in no way an estimate of the actual percent change.

4.7 Systems of Nonlinear Equations

Systems of nonlinear equations can be solved numerically by either using Newton's method (for small systems) or the fixed-point iteration method* (for large systems).

4.7.1 Newton's Method for a System of Nonlinear Equations

Newton's method for solving a single nonlinear equation was discussed in [Chapter 3](#). An extension of that technique can be used for solving a system of nonlinear equations. We will first present the idea and the details as pertained to a system of two nonlinear equations, followed by a general system of n nonlinear equations.

4.7.1.1 Newton's Method for Solving a System of Two Nonlinear Equations

A system of two (nonlinear) equations in two unknowns can generally be expressed as

$$\begin{aligned} f_1(x, y) &= 0 \\ f_2(x, y) &= 0 \end{aligned} \tag{4.35}$$

We begin by selecting (x_1, y_1) as an initial estimate of the solution. For the current two-dimensional case, for instance, (x_1, y_1) may be selected by first plotting f_1 and f_2 , then picking a point near their point of intersection (the solution of the system). Suppose (x_2, y_2) denotes the actual solution so that $f_1(x_2, y_2) = 0$ and $f_2(x_2, y_2) = 0$. If x_1 is sufficiently close to x_2 , and y_1 to y_2 , then $x_2 - x_1$ and $y_2 - y_1$ are small and by Taylor series expansion we have

$$\begin{aligned} f_1(x_2, y_2) &= f_1(x_1, y_1) + \left. \frac{\partial f_1}{\partial x} \right|_{(x_1, y_1)} (x_2 - x_1) + \left. \frac{\partial f_1}{\partial y} \right|_{(x_1, y_1)} (y_2 - y_1) + \dots \\ f_2(x_2, y_2) &= f_2(x_1, y_1) + \left. \frac{\partial f_2}{\partial x} \right|_{(x_1, y_1)} (x_2 - x_1) + \left. \frac{\partial f_2}{\partial y} \right|_{(x_1, y_1)} (y_2 - y_1) + \dots \end{aligned}$$

* Refer to Newton's method and fixed-point iteration method for a single nonlinear equation, [Chapter 3](#).

where the terms involving higher powers of small quantities $x_2 - x_1$ and $y_2 - y_1$ have been neglected. Let $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$ and recall that $f_1(x_2, y_2) = 0$ and $f_2(x_2, y_2) = 0$ to rewrite the above equations as

$$\begin{aligned} \frac{\partial f_1}{\partial x} \Big|_{(x_1, y_1)} \Delta x + \frac{\partial f_1}{\partial y} \Big|_{(x_1, y_1)} \Delta y + \dots &= -f_1(x_1, y_1) \\ \frac{\partial f_2}{\partial x} \Big|_{(x_1, y_1)} \Delta x + \frac{\partial f_2}{\partial y} \Big|_{(x_1, y_1)} \Delta y + \dots &= -f_2(x_1, y_1) \end{aligned}$$

which can be expressed as

$$\begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix} \Big|_{(x_1, y_1)} \begin{Bmatrix} \Delta x \\ \Delta y \end{Bmatrix} = \begin{Bmatrix} -f_1 \\ -f_2 \end{Bmatrix} \Big|_{(x_1, y_1)} \quad (4.36)$$

This is a linear system and can be solved for Δx and Δy as long as the coefficient matrix is non-singular. The matrix

$$J(f_1, f_2) = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix}$$

is called the Jacobian matrix of f_1 and f_2 . Therefore, for Equation 4.36 to have a non-trivial solution, we must have

$$\det \left\{ \left[J(f_1, f_2) \right] \Big|_{(x_1, y_1)} \right\} \neq 0$$

Once Equation 4.36 is solved, the values of $x_2 = x_1 + \Delta x$ and $y_2 = y_1 + \Delta y$ become available. And they clearly do not describe the actual solution because higher-order terms in Taylor series expansions were neglected earlier. Since (x_2, y_2) is closer to the actual solution than (x_1, y_1) was, we use (x_2, y_2) as the new estimate to the solution and solve Equation 4.36, with (x_2, y_2) replacing (x_1, y_1) , and continue the process until values generated at successive iterations meet a prescribed tolerance condition. A reasonable terminating condition is

$$\left\| \begin{Bmatrix} \Delta x \\ \Delta y \end{Bmatrix} \right\|_2 \leq \epsilon \quad (4.37)$$

where ϵ is a specified tolerance. Keep in mind that in each iteration step, the Jacobian matrix must be non-singular.

EXAMPLE 4.20: NEWTON'S METHOD

Solve the nonlinear system below using Newton's method, terminating condition as in Equation 4.37 with tolerance $\varepsilon = 10^{-4}$ and maximum number of iterations set to 20:

$$\begin{cases} 3.2x^3 + 1.8y^2 + 24.43 = 0 \\ -2x^2 + 3y^3 = 5.92 \end{cases}$$

Solution

The original system is written in the form of Equation 4.35 as

$$\begin{cases} f_1(x, y) = 3.2x^3 + 1.8y^2 + 24.43 = 0 \\ f_2(x, y) = -2x^2 + 3y^3 - 5.92 = 0 \end{cases}$$

First we need to find approximate locations of the roots graphically.

```
>> syms x y
>> f1 = 3.2*x^3+1.8*y^2+24.43; f2 = -2*x^2+3*y^3-5.92;
>> ezplot(f1, [-4, 0, -4, 6])
>> hold on
>> ezplot(f2, [-4, 0, -4, 6]) % Figure 4.6
```

Based on Figure 4.6, there is only one solution, and an appropriate initial estimate is $(-2, 2)$. Performing partial differentiations in Equation 4.36, we find

$$\begin{bmatrix} 9.6x^2 & 3.6y \\ -4x & 9y^2 \end{bmatrix}_{(-2,2)} \begin{Bmatrix} \Delta x \\ \Delta y \end{Bmatrix} = \begin{Bmatrix} -f_1 \\ -f_2 \end{Bmatrix}_{(-2,2)} \Rightarrow \begin{Bmatrix} \Delta x \\ \Delta y \end{Bmatrix}$$

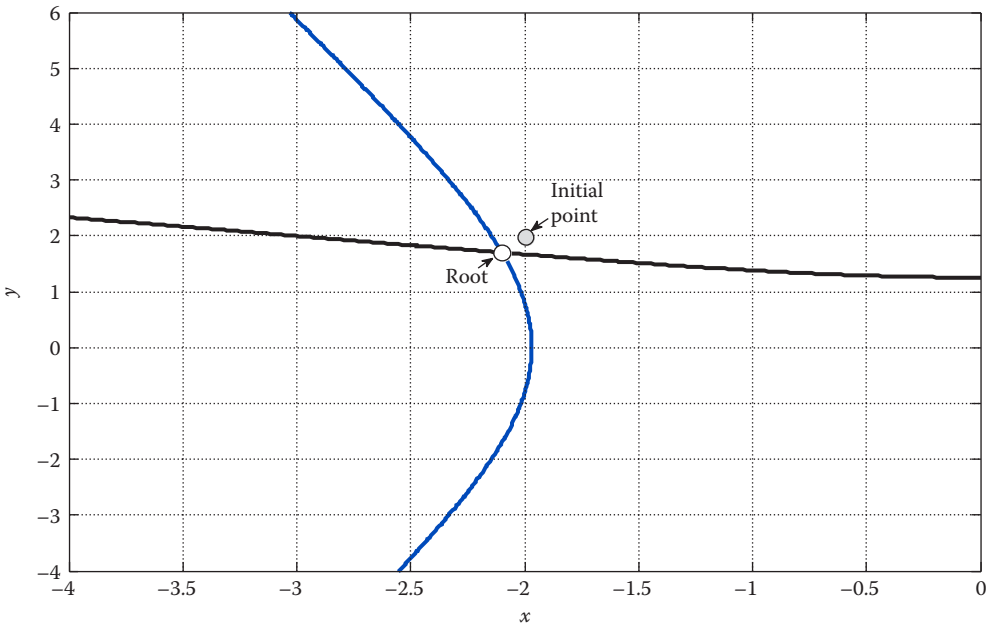


FIGURE 4.6

Graph of the nonlinear system in Example 4.20.

The next solution estimate is then found as

$$\begin{Bmatrix} x_2 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} x_1 \\ y_1 \end{Bmatrix} + \begin{Bmatrix} \Delta x \\ \Delta y \end{Bmatrix}$$

Hand calculation of the first iteration can be performed as follows:

$$\begin{bmatrix} 38.4 & 7.2 \\ 8 & 36 \end{bmatrix} \begin{Bmatrix} \Delta x \\ \Delta y \end{Bmatrix} = \begin{Bmatrix} -6.03 \\ -10.08 \end{Bmatrix} \quad \text{Solve} \quad \Rightarrow \quad \begin{Bmatrix} \Delta x \\ \Delta y \end{Bmatrix} = \begin{Bmatrix} -0.1091 \\ -0.2558 \end{Bmatrix}$$

Then

$$\begin{Bmatrix} x_2 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} -2 \\ 2 \end{Bmatrix} + \begin{Bmatrix} -0.1091 \\ -0.2558 \end{Bmatrix} = \begin{Bmatrix} -2.1091 \\ 1.7442 \end{Bmatrix}$$

To solve the problem entirely, the following MATLAB script will be used. In addition to the terminating condition in Equation 4.37, the script includes a segment that checks to see if $|f_1(x, y)| < \varepsilon$ and $|f_2(x, y)| < \varepsilon$ after each iteration. This is because sometimes an acceptable estimate of a root may have been found, but because of the nature of f_1 and/or f_2 , the current vector and the subsequent vector do not yet meet the terminating condition in Equation 4.37. The MATLAB built-in function `jacobian` is effectively used to generate the coefficient matrix in Equation 4.36.

```
f = [f1;f2]; % Note that f1 and f2 were already defined symbolically above
J = matlabFunction(jacobian(f,[x,y]));
F = matlabFunction(f);

tol = 1e-4; kmax = 20; v(:,1) = [-2;2];

for k = 1:kmax,
    A = J(v(1,k),v(2,k));
    b = -F(v(1,k),v(2,k));

    % The components of vector b are -f1 and -f2.
    if norm(b,inf) < tol,
        root = v(:,k);
        return
    end

    if det(A) == 0,
        break
    end

    delv = A\b;
    v(:,k+1) = v(:,k) + delv;
    if norm(delv) < tol,
        root = v(:,k+1);
        break
    end
end
end
```

Execution of this code results in

```
>> v
v =
-2.0000    -2.1091   -2.1001   -2.0999
 2.0000     1.7442    1.7012    1.7000
```

After three iterations, the solution is computed as $(-2.0999, 1.7000)$. The shaded values agree with the hand calculations presented earlier.

4.7.1.2 Newton's Method for Solving a System of n Nonlinear Equations

A system of n (nonlinear) equations in n unknowns can in general be expressed as

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0 \\ f_2(x_1, x_2, \dots, x_n) &= 0 \\ &\dots \\ f_n(x_1, x_2, \dots, x_n) &= 0 \end{aligned} \tag{4.38}$$

Choose $(x_{1,1}, x_{2,1}, \dots, x_{n,1})$ as the initial estimate and follow the steps that led to Equation 4.36 to arrive at

$$\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}_{(x_{1,1}, x_{2,1}, \dots, x_{n,1})} \begin{Bmatrix} \Delta x_1 \\ \Delta x_2 \\ \dots \\ \Delta x_n \end{Bmatrix} = \begin{Bmatrix} -f_1 \\ -f_2 \\ \dots \\ -f_n \end{Bmatrix}_{(x_{1,1}, x_{2,1}, \dots, x_{n,1})} \tag{4.39}$$

Solve this system to obtain the vector comprised of increments $\Delta x_1, \dots, \Delta x_n$. Then update the solution estimate

$$\begin{Bmatrix} x_{1,2} \\ x_{2,2} \\ \dots \\ x_{n,2} \end{Bmatrix} = \begin{Bmatrix} x_{1,1} \\ x_{2,1} \\ \dots \\ x_{n,1} \end{Bmatrix} + \begin{Bmatrix} \Delta x_1 \\ \Delta x_2 \\ \dots \\ \Delta x_n \end{Bmatrix}$$

If a specified terminating condition is not met, solve Equation 4.39 with $(x_{1,2}, x_{2,2}, \dots, x_{n,2})$ replacing $(x_{1,1}, x_{2,1}, \dots, x_{n,1})$ and continue the process until the terminating condition is satisfied.

4.7.1.3 Convergence of Newton's Method

Convergence of Newton's method is not guaranteed, but it is expected if these conditions hold:

- f_1, f_2, \dots, f_n and their partial derivatives are continuous and bounded near the actual solution.
- The Jacobian matrix $J(f_1, f_2, \dots, f_n)$ is non-singular near the solution.
- The initial solution estimate is sufficiently close to the actual solution.

As it was the case with a single nonlinear equation, if Newton's method does not exhibit convergence, it is usually because the initial solution estimate is not sufficiently close to the actual solution.

4.7.2 Fixed-Point Iteration Method for a System of Nonlinear Equations

The fixed-point iteration to solve a single nonlinear equation (Chapter 3) can be extended to handle systems of nonlinear equations in the form of Equation 4.38. The idea is to find suitable iteration functions $g_i(x_1, x_2, \dots, x_n)$, $i = 1, 2, \dots, n$ and rewrite Equation 4.38 as

$$\begin{aligned} x_1 &= g_1(x_1, x_2, \dots, x_n) \\ x_2 &= g_2(x_1, x_2, \dots, x_n) \\ &\dots \\ x_n &= g_n(x_1, x_2, \dots, x_n) \end{aligned} \tag{4.40}$$

or in vector form,

$$\mathbf{x} = \mathbf{g}(\mathbf{x}), \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{Bmatrix}, \quad \mathbf{g} = \begin{Bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \\ \dots \\ g_n(\mathbf{x}) \end{Bmatrix} \tag{4.41}$$

Choose $(x_{1,1}, x_{2,1}, \dots, x_{n,1})$ as the initial estimate and substitute into the right sides of the equations in Equation 4.40. The updated estimates are calculated as

$$\begin{aligned} x_{1,2} &= g_1(x_{1,1}, x_{2,1}, \dots, x_{n,1}) \\ x_{2,2} &= g_2(x_{1,1}, x_{2,1}, \dots, x_{n,1}) \\ &\dots \\ x_{n,2} &= g_n(x_{1,1}, x_{2,1}, \dots, x_{n,1}) \end{aligned}$$

These new values are then inserted in the right sides of Equation 4.40 to generate the new updates, and so on. The process continues until convergence is observed.

4.7.2.1 Convergence of the Fixed-Point Iteration Method

The conditions for convergence of the fixed-point iteration

$$\mathbf{x}^{(k+1)} = \mathbf{g}(\mathbf{x}^{(k)}), \quad k = 0, 1, 2, \dots \tag{4.42}$$

are similar to those for the case of a function of one variable. Let R be an n -dimensional rectangular region comprised of points x_1, x_2, \dots, x_n such that $a_i \leq x_i \leq b_i$ ($i = 1, 2, \dots, n$) for constants a_1, a_2, \dots, a_n and b_1, b_2, \dots, b_n . Suppose $\mathbf{g}(\mathbf{x})$ is defined on R . Then the sufficient conditions for convergence of the fixed-point iteration method, Equation 4.42, are*:

- Iteration functions g_1, g_2, \dots, g_n and their partial derivatives with respect to x_1, x_2, \dots, x_n are continuous near the actual solution.

* Refer to Atkinson, K.E., *An Introduction to Numerical Analysis*, 2nd ed., John Wiley, NY, 1989

- There exists a constant $K < 1$ such that for each $\mathbf{x} \in R$,

$$\left| \frac{\partial g_j(\mathbf{x})}{\partial x_i} \right| \leq \frac{K}{n}, \quad j = 1, 2, \dots, n, \quad i = 1, 2, \dots, n \quad (4.43)$$

which may also be interpreted as

$$\begin{aligned} \left| \frac{\partial g_1}{\partial x_1} \right| + \left| \frac{\partial g_1}{\partial x_2} \right| + \dots + \left| \frac{\partial g_1}{\partial x_n} \right| &\leq 1 \\ \left| \frac{\partial g_2}{\partial x_1} \right| + \left| \frac{\partial g_2}{\partial x_2} \right| + \dots + \left| \frac{\partial g_2}{\partial x_n} \right| &\leq 1 \\ &\dots \\ \left| \frac{\partial g_n}{\partial x_1} \right| + \left| \frac{\partial g_n}{\partial x_2} \right| + \dots + \left| \frac{\partial g_n}{\partial x_n} \right| &\leq 1 \end{aligned} \quad (4.44)$$

- The initial estimate $(x_{1,1}, x_{2,1}, \dots, x_{n,1})$ is sufficiently close to the actual solution.

EXAMPLE 4.21: FIXED-POINT ITERATION

Using the fixed-point iteration method, solve the nonlinear system in Example 4.20:

$$\begin{cases} 3.2x^3 + 1.8y^2 + 24.43 = 0 \\ -2x^2 + 3y^3 = 5.92 \end{cases}$$

Use the same initial estimate and terminating condition as before.

Solution

We first need to rewire the given equations in the form of Equation 4.40 by selecting suitable iteration functions. Recall that these iteration functions are not unique. One way to rewrite the original system is

$$\begin{aligned} x &= g_1(x, y) = -\left(\frac{1.8y^2 + 24.43}{3.2} \right)^{1/3} \\ y &= g_2(x, y) = \left(\frac{2x^2 + 5.92}{3} \right)^{1/3} \end{aligned}$$

Based on [Figure 4.6](#), a reasonable rectangular region R is chosen as $-4 \leq x \leq -2$, $0 \leq y \leq 2$. We next examine the conditions listed in Equation 4.43 in relation to our choices of g_1 and g_2 . Noting $n = 2$ in this example, the four conditions to be met are

$$\left| \frac{\partial g_1}{\partial x} \right| < \frac{1}{2}, \quad \left| \frac{\partial g_1}{\partial y} \right| < \frac{1}{2}, \quad \left| \frac{\partial g_2}{\partial x} \right| < \frac{1}{2}, \quad \left| \frac{\partial g_2}{\partial y} \right| < \frac{1}{2}$$

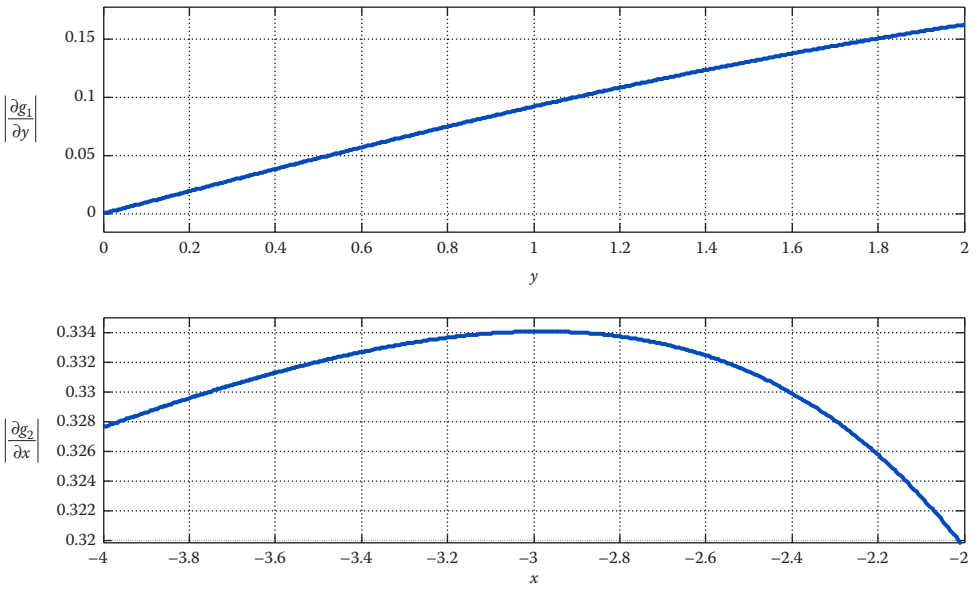


FIGURE 4.7

Graphical inspection of upper bounds for $|\partial g_1/\partial y|$ and $|\partial g_2/\partial x|$.

Of course, $|\partial g_1/\partial x| = 0 < \frac{1}{2}$ and $|\partial g_2/\partial y| = 0 < \frac{1}{2}$ satisfy two of the above. The other two may be inspected with the aid of MATLAB as follows:

```
>> syms x y
>> g1 = sym(' -((1.8*y^2+24.43)/3.2)^(1/3) ');
>> g2 = sym(' ((2*x^2+5.92)/3)^(1/3) ');
>> subplot(2,1,1), ezplot(abs(diff(g1,'y')), [0 2]) % First plot in Figure 4.7
>> subplot(2,1,2), ezplot(abs(diff(g2,'x')), [-4 -2]) % Complete Figure 4.7
```

The two plots in Figure 4.7 clearly indicate that the two remaining partial derivatives satisfy their respective conditions as well. This means that the vector function

$$\mathbf{g} = \begin{Bmatrix} g_1 \\ g_2 \end{Bmatrix}$$

has a fixed point in region R , and the fixed-point iteration in Equation 4.42 is guaranteed to converge to this fixed point.

The following code will use the fixed-point iteration to generate a sequence of values for x and y and terminates the iterations as soon as the tolerance is met. For simplicity, we define a vector

$$\mathbf{v} = \begin{Bmatrix} x \\ y \end{Bmatrix}$$

and subsequently define g_1 and g_2 as functions of the components of vector \mathbf{v} .

```
% Define the iteration functions g1 and g2
g1 = @(v) ( -((24.43+1.8*v(2,1)^2)/3.2)^(1/3) );
g2 = @(v) ( ((5.92+2*v(1,1)^2)/3)^(1/3) );
```

```

tol = 1e-4; kmax = 20;
v(:,1) = [-1;-2]; % Initial estimate
for k = 1:kmax,
    v(:,k+1) = [g1(v(:,k));g2(v(:,k))]; % Fixed-point iteration
    if norm(v(:,k+1)-v(:,k)) < tol,
        break
    end
end

```

Execution of this code results in

```
>> v
```

```
v =
```

```


-1.0000 -2.1461 -2.0574 -2.1021 -2.0980 -2.1000 -2.0998 -2.0999 -2.0999
-2.0000  1.3821  1.7150  1.6863  1.7007  1.6994  1.7000  1.7000  1.7000

```

Convergence to the true solution is observed after eight (8) iterations. The estimated solution agrees with that in Example 4.20.

PROBLEM SET (CHAPTER 4)

Gauss Elimination Method (Section 4.3)

 In Problems 1 through 12 solve the linear system using basic Gauss elimination with partial pivoting, if necessary.

$$1. \begin{cases} 3x_1 + 2x_2 = 0 \\ -x_1 + 2x_2 = 8 \end{cases}$$

$$2. \begin{cases} 0.6x_1 + 1.3x_2 = 0.2 \\ 2.1x_1 - 3.2x_2 = 3.8 \end{cases}$$

$$3. \begin{cases} 2x_1 + 6x_2 = 5 \\ 3x_1 - 4x_2 = 1 \end{cases}$$

$$4. \begin{cases} 2x_1 - 3x_2 + 2x_3 = 1 \\ x_2 + 3x_3 = -11 \\ -6x_1 + 9x_2 - 7x_3 = 1 \end{cases}$$

$$5. \begin{cases} 3x_1 + 5x_3 = -1 \\ -x_1 + 5x_2 - 2x_3 = 12 \\ 2x_1 - 7x_2 + 4x_3 = -18 \end{cases}$$

$$6. \begin{cases} x_1 + 3x_2 - 2x_3 = -7 \\ -2x_1 + x_2 + 2x_3 = 8 \\ 3x_1 + 8x_2 + x_3 = 0 \end{cases}$$

$$7. \begin{bmatrix} -3 & -3 & 2 \\ 0 & 9 & 2 \\ 1 & 4 & -1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} -12 \\ -6 \\ 5 \end{Bmatrix}$$

$$8. \begin{bmatrix} 4 & -1 & 0 \\ 1 & 8 & 1 \\ 2 & 5 & 6 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} -9 \\ 10 \\ 25 \end{Bmatrix}$$

$$9. \begin{bmatrix} 3 & -4 & -1 \\ 0 & 7 & 3 \\ 6 & -29 & -1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 0 \\ -1 \\ -47 \end{Bmatrix}$$

$$10. \begin{bmatrix} 3 & 5 & 15.2 \\ 1 & 0 & 6 \\ -2 & 2.5 & -8 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 9.3 \\ 0 \\ 6 \end{Bmatrix}$$

$$11. \begin{bmatrix} 1.5 & 1 & -2.5 \\ 3 & 2 & -6.2 \\ -2.4 & 0.4 & 5 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 0.5 \\ -0.2 \\ -1.3 \end{Bmatrix}$$

$$12. \begin{bmatrix} -5 & 1 & 16 & -12 \\ 1 & 0 & -4 & 3 \\ 0 & -3 & 10 & -5 \\ 4 & 8 & -24 & -3 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} -28 \\ 6 \\ -2 \\ 1 \end{Bmatrix}$$

Small Pivot


13. Consider the linear 2×2 system


$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} 2 \\ 1 \end{Bmatrix}, \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix}$$

where $\varepsilon > 0$ is a very small constant.

- Solve by Gauss elimination without partial pivoting.
- Solve by Gauss elimination with partial pivoting. Compare the results and discuss their validity.

In Problems 14 through 18, a linear system is given.

a.  Solve using Gauss elimination with partial pivoting and row scaling.

b.  Solve by executing the user-defined function `GaussPivotScale`.

$$14. \begin{bmatrix} 5 & 8 & 12 \\ -3 & 5 & -8 \\ 2 & -1 & 4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 15 \\ 11 \\ -2 \end{Bmatrix}$$

$$15. \begin{bmatrix} 2 & 1 & -6 \\ 3 & -2 & 2 \\ 1 & 5 & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 8 \\ 1 \\ 1 \end{Bmatrix}$$


$$16. \begin{bmatrix} -3 & 3 & 5 \\ 3 & -2 & -2 \\ 9 & 10 & -1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} -14 \\ 3 \\ 5 \end{Bmatrix}$$

$$17. \begin{cases} 3x_1 + 2x_2 = -7 \\ -3x_1 + 5x_2 - x_3 = 9 \\ 6x_1 + 5x_2 + 12x_3 = 47 \end{cases}$$

$$18. \begin{cases} 2x_1 - x_2 + 3x_3 = 17 \\ 5x_1 + 6x_2 + 4x_3 = 3 \\ x_1 + x_3 = 5 \end{cases}$$

Tridiagonal Systems

In Problems 19 through 24, a tridiagonal system is given.

a.  Solve using the Thomas method.

b.  Solve by executing the user-defined function `ThomasMethod`.

$$19. \begin{bmatrix} -2 & 1 & 0 \\ 3 & 2 & 1 \\ 0 & -1 & 3 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 3 \\ 1 \\ 5 \end{Bmatrix}$$

$$20. \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} -1 \\ 5 \\ 5 \end{Bmatrix}$$

$$21. \begin{bmatrix} 1 & -2 & 0 & 0 \\ 2 & 3 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} -4 \\ 5 \\ 7 \\ 13 \end{Bmatrix}$$



$$22. \begin{bmatrix} 0.1 & 0.09 & 0 & 0 \\ 0.12 & 1.2 & 0.8 & 0 \\ 0 & 1.1 & 0.9 & 0.6 \\ 0 & 0 & -1.3 & 0.9 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} -0.01 \\ 0.28 \\ 1.4 \\ 3.1 \end{Bmatrix}$$

$$23. \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 0 & -2 & 3 & 1 & 0 \\ 0 & 0 & 1 & -4 & 0 \\ 0 & 0 & 0 & 3 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} 2 \\ 7 \\ -8 \\ 8 \\ -7 \end{Bmatrix}$$

$$24. \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 1 & -3 & -1 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & -2 & 3 & 4 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} -1 \\ 1 \\ -4 \\ 9 \\ 4 \end{Bmatrix}$$


25. Finite difference methods are used to numerically solve boundary-value problems; [Chapter 8](#). These methods are designed so that tridiagonal systems are generated in their solution process. In one such application, the following tridiagonal system has been created:

$$\begin{bmatrix} -6 & 3.5 & 0 & 0 & 0 \\ 3.5 & -8 & 4.5 & 0 & 0 \\ 0 & 4.5 & -10 & 5.5 & 0 \\ 0 & 0 & 5.5 & -12 & 6.5 \\ 0 & 0 & 0 & 4 & -4 \end{bmatrix} \begin{Bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{Bmatrix} = \begin{Bmatrix} -4 \\ -2 \\ -2.5 \\ -3 \\ -1 \end{Bmatrix}$$

-  Solve using the Thomas method. Use 4-digit rounding up.
-  Solve by executing the user-defined function `ThomasMethod`. Compare with the results in (a).

26. Alternating direct implicit (ADI) methods are used to numerically solve a certain type of partial differential equation in a rectangular region; [Chapter 10](#). These methods are specifically designed to generate tridiagonal systems in their solution process. In one such application, the following tridiagonal system has been created:

$$\begin{bmatrix} -4 & 1 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 1 & -4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix} = \begin{Bmatrix} -0.6056 \\ -1.0321 \\ -1.1389 \\ -0.2563 \\ -0.4195 \\ -0.3896 \end{Bmatrix}$$

-  Solve using the Thomas method. Use 4-digit rounding up.
- Solve by executing the user-defined function `ThomasMethod`. Compare with the results in (a).

LU Factorization Methods (Section 4.4)

Doolittle Factorization

 In Problems 27 through 30 find the Doolittle factorization of each matrix using the steps of Gauss elimination method.

$$27. \mathbf{A} = \begin{bmatrix} -1 & 2 & 2 \\ 3 & -4 & -5 \\ -2 & 6 & 3 \end{bmatrix}$$

$$28. \mathbf{A} = \begin{bmatrix} -1 & 3 & -5 \\ 4 & -1 & 0 \\ 2 & 5 & 6 \end{bmatrix}$$

$$29. \mathbf{A} = \begin{bmatrix} 2 & 4 & -2 & 6 \\ 1 & 3 & 2 & 5 \\ 4 & 7 & -3 & 10 \\ 3 & 5 & 1 & 11 \end{bmatrix}$$

$$30. \mathbf{A} = \begin{bmatrix} 3 & 6 & 3 & 9 \\ 1 & 5 & 4 & 7 \\ -2 & -1 & 2 & -3 \\ 3 & 0 & 1 & 7 \end{bmatrix}$$

In Problems 31 through 38,

- Using Doolittle's method solve each linear system $\mathbf{Ax} = \mathbf{b}$.
- Confirm the results by executing `DoolittleMethod`.

$$31. \mathbf{A} = \begin{bmatrix} 3 & 1 & 1 \\ -3 & -3 & 1 \\ 3 & -3 & 6 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 2 \\ -4 \\ 0 \end{bmatrix}$$

$$32. \mathbf{A} = \begin{bmatrix} 2 & 2 & 1 \\ 1 & -1 & \frac{13}{2} \\ -2 & -\frac{10}{3} & 6 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 6 \\ -15 \\ -24 \end{bmatrix}$$

$$33. \mathbf{A} = \begin{bmatrix} 1 & 3 & -3 \\ \frac{1}{3} & -5 & 2 \\ \frac{2}{3} & 14 & -3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -9 \\ 0 \\ 3 \end{bmatrix}$$

$$34. \mathbf{A} = \begin{bmatrix} 4 & -2 & 8 \\ -4 & 5 & -13 \\ 1 & -\frac{19}{2} & 19 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -8 \\ 19 \\ -37 \end{bmatrix}$$

$$35. \mathbf{A} = \begin{bmatrix} -1 & 3 & -5 \\ 4 & -1 & 0 \\ 2 & 5 & 6 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -14 \\ 5 \\ 9 \end{bmatrix}$$

$$36. \mathbf{A} = \begin{bmatrix} -1 & 2 & 2 \\ 3 & -4 & -5 \\ -2 & 6 & 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -8 \\ 21 \\ -13 \end{bmatrix}$$

$$37. \mathbf{A} = \begin{bmatrix} 3 & 0 & -1 & 2 \\ -3 & 2 & 2 & 1 \\ 0 & -2 & -5 & -2 \\ 6 & 6 & -7 & 20 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 4 \\ -1 \\ 6 \\ 40 \end{bmatrix}$$

$$38. \mathbf{A} = \begin{bmatrix} -2 & 1 & 3 & -1 \\ -4 & 5 & 6 & 0 \\ 4 & -2 & -1 & -1 \\ -2 & 13 & -12 & 18 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0 \\ 7 \\ -11 \\ 65 \end{bmatrix}$$

Cholesky Factorization Method

In Problems 39 through 44,

- Using Cholesky's method solve each linear system $\mathbf{Ax} = \mathbf{b}$.
- Confirm the results by executing `CholeskyMethod`.

$$39. \mathbf{A} = \begin{bmatrix} 1 & 1 & -2 \\ 1 & 10 & 4 \\ -2 & 4 & 24 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} -3 \\ 33 \\ 78 \end{Bmatrix}$$

$$40. \mathbf{A} = \begin{bmatrix} 9 & -6 & 3 \\ -6 & 13 & 1 \\ 3 & 1 & 6 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} -30 \\ 53 \\ 9 \end{Bmatrix}$$

$$41. \mathbf{A} = \begin{bmatrix} 4 & 2 & -6 \\ 2 & 17 & 5 \\ -6 & 5 & 17 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} -14 \\ 17 \\ 45 \end{Bmatrix}$$

$$42. \mathbf{A} = \begin{bmatrix} 1 & -2 & -3 \\ -2 & 5 & 7 \\ -3 & 7 & 26 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} -6 \\ 13 \\ 83 \end{Bmatrix}$$

$$43. \mathbf{A} = \begin{bmatrix} 4 & 2 & 6 & -4 \\ 2 & 2 & 2 & -6 \\ 6 & 2 & 11 & -3 \\ -4 & -6 & -3 & 25 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} -16 \\ -26 \\ -12 \\ 114 \end{Bmatrix}$$

$$44. \mathbf{A} = \begin{bmatrix} 9 & 6 & 3 & 6 \\ 6 & 5 & 6 & 7 \\ 3 & 6 & 21 & 18 \\ 6 & 7 & 18 & 18 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} 3 \\ 1 \\ -3 \\ 1 \end{Bmatrix}$$

Crout Factorization

45. Crout LU factorization requires the diagonal entries of \mathbf{U} be 1's, while \mathbf{L} is a general lower triangular matrix. Perform direct calculation of the entries of \mathbf{L} and \mathbf{U} for the case of a 3×3 matrix, similar to that in Example 4.6. Based on the findings, write a user-defined function with function call `[L, U] = Crout_Factor(A)` that returns the desired lower and upper triangular matrices for any $n \times n$ matrix. Apply `Crout_Factor` to

$$\mathbf{A} = \begin{bmatrix} 4 & 1 & 7 \\ 2 & 3 & -6 \\ -5 & -1 & -5 \end{bmatrix}$$

Crout's Method

46. Crout's method uses Crout factorization (Problem 45) of the coefficient matrix \mathbf{A} of the linear system $\mathbf{Ax} = \mathbf{b}$ and generates two triangular systems, which

can be solved by back and forward substitution. Write a user-defined function with function call $\mathbf{x} = \text{Crout_Method}(\mathbf{A}, \mathbf{b})$. Apply Crout_Method to

$$\begin{bmatrix} 4 & 1 & 7 \\ 2 & 3 & -6 \\ -5 & -1 & -5 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 11 \\ -11 \\ -6 \end{Bmatrix}$$

Iterative Solution of Linear Systems (Section 4.5)

Vector/Matrix Norms

In Problems 47 through 54,

- Calculate the three norms of each vector or matrix.
- Verify the results by using the MATLAB built-in function `norm`.

$$47. \mathbf{v} = \frac{1}{\sqrt{2}} \begin{Bmatrix} 1 \\ 1 \end{Bmatrix}$$

$$48. \mathbf{v} = \sqrt{3} \begin{Bmatrix} 2 \\ -3 \\ 1 \end{Bmatrix}$$

$$49. \mathbf{v} = \begin{Bmatrix} \frac{1}{3} \\ \frac{2}{3} \\ \frac{1}{4} \end{Bmatrix}$$

$$50. \mathbf{v} = \begin{Bmatrix} -1 \\ 0 \\ 3 \\ -4 \end{Bmatrix}$$

$$51. \mathbf{A} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 1 & 0 \\ -1 & \frac{1}{\sqrt{2}} & 1 \\ 0 & -2 & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$52. \mathbf{A} = \begin{bmatrix} 10 & 0.3 & -0.7 \\ 0.2 & 7 & 1.2 \\ 0.9 & -1.1 & 5 \end{bmatrix}$$

$$53. \mathbf{A} = \begin{bmatrix} -\frac{1}{5} & \frac{1}{2} & \frac{1}{3} & 0 \\ \frac{2}{3} & 1 & -\frac{1}{5} & \frac{1}{3} \\ \frac{1}{3} & -\frac{1}{5} & 1 & \frac{2}{3} \\ 0 & \frac{2}{5} & \frac{1}{3} & -\frac{1}{5} \end{bmatrix}$$

$$54. \mathbf{A} = \begin{bmatrix} 2 & -\frac{1}{2} & \frac{1}{3} & 0 \\ \frac{1}{2} & 1 & -\frac{1}{5} & \frac{1}{3} \\ -\frac{1}{3} & \frac{1}{5} & 3 & -\frac{2}{3} \\ 0 & -\frac{1}{3} & \frac{2}{3} & 4 \end{bmatrix}$$

55. Write a user-defined function with function call $[x, k, Mnorm] = \text{GenIter}_1(A, b, x_0, tol, kmax)$ to solve $\mathbf{Ax} = \mathbf{b}$ using the general iterative method as follows: The coefficient matrix is split as $\mathbf{A} = \mathbf{Q} - \mathbf{P}$ where \mathbf{Q} has the same diagonal and upper diagonal (one level higher than the diagonal) entries as \mathbf{A} with all other entries zero. The input/output arguments, as well as the terminating condition are as in functions `Jacobi` and `GaussSeidel` with the same default values. The output $Mnorm$ is the infinite norm of the corresponding iteration matrix. Apply `GenIter_1` to the linear system

$$\begin{bmatrix} -6 & 2 & 0 & 1 & -1 \\ 1 & 7 & -2 & 1 & 1 \\ 2 & 0 & 9 & -3 & 3 \\ 0 & 2 & -3 & 8 & 2 \\ -2 & 4 & 1 & -5 & 14 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 16 \\ 1 \\ 8 \\ 2 \\ -9 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{Bmatrix}, \quad \varepsilon = 10^{-6}$$

56. Write a user-defined function with function call $[x, k, Mnorm] = \text{GenIter}_2(A, b, x_0, tol, kmax)$ to solve $\mathbf{Ax} = \mathbf{b}$ using the general iterative method as follows: The coefficient matrix must be split as $\mathbf{A} = \mathbf{Q} - \mathbf{P}$ where \mathbf{Q} has the same diagonal, upper diagonal (one level higher than the diagonal), and lower diagonal (one level lower than the diagonal) entries as \mathbf{A} with all other entries zero. The input/output arguments, as well as the terminating condition are as in functions `Jacobi` and `GaussSeidel` with the same default values. The output $Mnorm$ is the infinite norm of the corresponding iteration matrix. Apply `GenIter_2` to the linear system

$$\begin{bmatrix} 8 & 0 & -1 & 1 & 4 \\ 0 & 6 & 0 & 1 & 4 \\ 2 & -1 & -5 & 0 & -1 \\ 3 & 2 & 1 & -7 & 0 \\ -1 & 3 & 4 & -1 & 11 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} -11 \\ 16 \\ -9 \\ -10 \\ 1 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{Bmatrix}, \quad \varepsilon = 10^{-6}$$

Jacobi Iteration Method

In Problems 57 through 60,

- For each linear system find the components of the first vector generated by the Jacobi method.
- Find the solution vector by executing the user-defined function `Jacobi` with default values for `tol` and `kmax`.

$$57. \begin{bmatrix} 1.9 & -0.7 & 0.9 \\ 0.6 & 2.3 & 1.2 \\ -0.8 & 1.3 & 3.2 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 1.5 \\ 0.7 \\ 9.1 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 1 \\ 0 \\ 1 \end{Bmatrix}$$

$$58. \begin{bmatrix} -4 & 1 & 0 \\ 1 & 3 & -1 \\ 0 & -1 & 5 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} -6 \\ -6 \\ 7 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \mathbf{0}_{3 \times 1}$$

$$59. \begin{bmatrix} 3 & 0 & 1 & -1 \\ 0 & -4 & 2 & 1 \\ 1 & -2 & 5 & 0 \\ -1 & 3 & 2 & 6 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 5 \\ -3 \\ -4 \\ 16 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{Bmatrix}$$

$$60. \begin{bmatrix} 6 & 2 & 1 & -2 \\ 2 & 5 & -1 & 0 \\ -1 & 3 & 7 & 1 \\ -2 & 1 & 4 & -8 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 5 \\ 7 \\ 28 \\ 6 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{Bmatrix}$$



 In Problems 61 and 62, calculate the first two vectors generated by the Jacobi method.

$$61. \begin{bmatrix} -3 & 1 & 2 \\ 2 & 4 & -1 \\ 1 & -2 & 4 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 24 \\ -5 \\ 12 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 1 \\ 0 \\ 1 \end{Bmatrix}$$

$$62. \begin{bmatrix} -5 & 4 & 0 \\ 2 & 6 & -3 \\ -1 & 2 & 3 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 18 \\ 11 \\ 3 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 1 \\ 0 \end{Bmatrix}$$

Gauss–Seidel Iteration Method

In Problems 63 through 66,


-  For each linear system find the components of the first vector generated by the Gauss–Seidel method.
-  Solve the system by executing the user-defined function `GaussSeidel` with default values for `tol` and `kmax`.

$$63. \begin{bmatrix} 4 & 2 & -6 \\ 2 & 12 & 5 \\ -6 & 5 & 17 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 2 \\ 15 \\ 5 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 1 \\ 0 \end{Bmatrix}$$

$$64. \begin{bmatrix} 1 & 1 & -2 \\ 1 & 10 & 4 \\ -2 & 4 & 18 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} -6 \\ 15 \\ 46 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 1 \\ 1 \\ 0 \end{Bmatrix}$$


$$65. \begin{bmatrix} 6 & 3 & -2 & 0 \\ 3 & 7 & 1 & -2 \\ -2 & 1 & 8 & 3 \\ 0 & -2 & 3 & 9 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 9 \\ -5 \\ 1 \\ 20 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{Bmatrix}$$

$$66. \begin{bmatrix} 5 & 1 & -1 & 2 \\ -2 & 10 & 1 & 3 \\ 0 & -4 & 10 & 1 \\ 3 & 3 & -2 & 10 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 6.5 \\ 7 \\ 9 \\ 12.5 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{Bmatrix}$$

 In Problems 67 and 68, calculate the first two vectors generated by the Gauss–Seidel method.

$$67. \begin{bmatrix} 1 & 2 & -3 \\ 2 & 10 & 4 \\ -3 & 4 & 30 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 1 \\ 30 \\ 48 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 1 \\ 0 \\ 0 \end{Bmatrix}$$

$$68. \begin{bmatrix} 6 & 2 & 6 \\ 2 & 18 & 5 \\ 6 & 5 & 16 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} -28 \\ 2 \\ -45 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 1 \\ 1 \\ 0 \end{Bmatrix}$$

 In Problems 69 and 70 solve the linear system by executing user-defined functions `Jacobi` and `GaussSeidel` with the initial vector and tolerance as indicated, and default `kmax`. Compare the results and discuss convergence.



$$69. \begin{bmatrix} 9 & -2 & -1 & 0 & 3 \\ 0 & 7 & 3 & -1 & 0 \\ 1 & -2 & 8 & 2 & -1 \\ 1 & -3 & 1 & 9 & -1 \\ 4 & -1 & 2 & -2 & 10 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 21 \\ 0 \\ 17 \\ -3 \\ 25 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \end{Bmatrix}, \quad \varepsilon = 10^{-6}$$

$$70. \begin{bmatrix} -6.5 & 1 & 0 & 1 & 3 \\ 1 & 6 & -1 & 0 & 1 \\ 0 & -1 & 10 & -1 & 2 \\ 1 & 0 & -1 & 9 & 1 \\ 3 & 1 & 2 & 1 & 10 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} -23.5 \\ -14 \\ 39 \\ 7 \\ 6 \end{Bmatrix}, \quad \mathbf{x}^{(0)} = \begin{Bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{Bmatrix}, \quad \varepsilon = 10^{-6}$$

III-Conditioning and Error Analysis (Section 4.6)

Condition Number

In Problems 71 through 76,

-  Calculate the condition number using all three matrix norms.
-  Verify the results using the MATLAB built-in function `cond`.

$$71. \mathbf{A} = \begin{bmatrix} 1 & 0.4 \\ 3 & 1.1 \end{bmatrix}$$

$$72. \mathbf{A} = \begin{bmatrix} 3 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 2 & 1 \end{bmatrix}$$

$$73. \mathbf{A} = \begin{bmatrix} 4 & 2 & 0 \\ 2 & 5 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$




$$74. \mathbf{A} = \begin{bmatrix} 2 & 7 & 4 \\ 2 & 1 & 2 \\ 5 & -1 & 2 \end{bmatrix}$$

$$75. \mathbf{A} = \begin{bmatrix} 1 & -1 & 1 & 2 \\ -2 & 1 & -3 & -4 \\ 1 & 1 & 4 & 2 \\ 3 & -6 & 4 & 5 \end{bmatrix}$$

$$76. \mathbf{A} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix} \text{4}\times\text{4 Hilbert matrix}$$

III-Conditioning

In Problems 77 through 80, a linear system $\mathbf{Ax} = \mathbf{b}$, its actual solution \mathbf{x}_a , and a poor approximation $\hat{\mathbf{x}}$ of the solution are given. Perform all of the following to inspect the ill-conditioning or well-conditioning of the system.

-  Perturb the second component of \mathbf{b} by a small $\epsilon > 0$ and find the solution of the ensuing system.
-  Find the condition number of \mathbf{A} using the 1-norm.
-  Calculate the 1-norm of the residual vector corresponding to the poor approximation $\hat{\mathbf{x}}$.



$$77. \begin{bmatrix} 1 & 2 \\ 2 & 4.0001 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} -1 \\ -2.0002 \end{Bmatrix}, \quad \mathbf{x}_a = \begin{Bmatrix} 3 \\ -2 \end{Bmatrix}, \quad \hat{\mathbf{x}} = \begin{Bmatrix} -1 \\ 0 \end{Bmatrix}$$

$$78. \begin{bmatrix} 2 & 2 \\ 1.0002 & 0.9998 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 4 \\ 2.0012 \end{Bmatrix}, \quad \mathbf{x}_a = \begin{Bmatrix} 4 \\ -2 \end{Bmatrix}, \quad \hat{\mathbf{x}} = \begin{Bmatrix} 2 \\ 0 \end{Bmatrix}$$

$$79. \begin{bmatrix} 5 & 9 \\ 6 & 11 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} -1 \\ -1 \end{Bmatrix}, \quad \mathbf{x}_a = \begin{Bmatrix} -2 \\ 1 \end{Bmatrix}, \quad \hat{\mathbf{x}} = \begin{Bmatrix} 7.2 \\ -4.1 \end{Bmatrix}$$

$$80. \begin{bmatrix} 13 & 14 & 14 \\ 11 & 12 & 13 \\ 12 & 13 & 14 \end{bmatrix} \mathbf{x} = \begin{Bmatrix} 55 \\ 48 \\ 52 \end{Bmatrix}, \quad \mathbf{x}_a = \begin{Bmatrix} 1 \\ 2 \\ 1 \end{Bmatrix}, \quad \hat{\mathbf{x}} = \begin{Bmatrix} -0.51 \\ 3.61 \\ 0.79 \end{Bmatrix}$$

Percent Change

81.  Consider the system in Example 4.19 and suppose the second component of vector \mathbf{b} is increased by 0.0001, while the coefficient matrix is unchanged from its original form. Find the upper bound for, and the actual value of, the percent change in solution. Use the vector and matrix 1-norm.
82.  Consider

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} 5 & 9 \\ 6 & 11 \end{bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} -1 \\ -1 \end{Bmatrix}$$

- a. Increase each of the second-column entries of \mathbf{A} by 0.01, solve the ensuing system, and calculate the actual percent change in solution. Also find an upper bound for the percent change. Use matrix and vector 1-norm.
- b. In the original system, increase each of the components of \mathbf{b} by 0.01, and repeat Part (a).

Systems of Nonlinear Equations (Section 4.7)

83.  Consider

$$\begin{cases} x^2 + (y-1)^2 = 2 \\ xy + (x-1)^2 = 0 \end{cases}$$

Starting with the initial estimate $(x_1, y_1) = (1.4, 1)$, use Newton's method to find (x_2, y_2) .

84.  Consider


$$\begin{cases} y - \frac{1}{x} = 1 \\ \frac{1}{2}x^2 + \frac{1}{3}(y-1)^2 = 2 \end{cases}$$

Starting with the initial estimate $(x_1, y_1) = (2, 0)$, use Newton's method to find (x_2, y_2) and (x_3, y_3) .

85.  Consider

$$\begin{cases} (x-1)^3 y = -1 \\ xy^2 = \sin x \end{cases}$$

First locate the roots graphically. Then find an approximate value for one of the roots using Newton's method with initial estimate $(2, -2)$, a terminating condition with $\varepsilon = 10^{-4}$, and allow a maximum of 20 iterations.

 In Problems 86 through 91, solve the nonlinear system as follows: Locate the roots graphically. Then find the approximate values for all the roots using Newton's method

with suitable initial estimates. In all cases, use a terminating condition with $\varepsilon = 10^{-4}$ and allow a maximum of 20 iterations.

$$86. \begin{cases} y - x^2 + 4 = 0 \\ x^2 + y^2 = 10 \end{cases}$$


$$87. \begin{cases} 1.4x^2 + y^2 = 1.6 \\ 2x^2 + 3y^2 = 2.8 \end{cases}$$

$$88. \begin{cases} 3e^{0.8x} + y = 0 \\ 2x^2 + 4y^2 = 8 \end{cases}$$

$$89. \begin{cases} 2x^2 + 2\sqrt{2}xy + y^2 = 14 \\ x^2 - \sqrt{2}xy + 3y^2 = 9 \end{cases}$$

$$90. \begin{cases} \sin(\alpha + \beta) - \cos\beta = 0.17 \\ \cos(\alpha - \beta) + \sin\alpha = 1.8 \end{cases}$$

$$91. \begin{cases} x^2 + y - 2x = 0.4 \\ 2y + 3xy - 2x^3 = 0 \end{cases}$$

92.  A two-link robot arm in plane motion is shown in [Figure 4.8](#). The coordinate system xy is the tool frame and is attached to the end-effector. The coordinates of the end-effector relative to the base frame are expressed as

$$\begin{cases} x = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2) \\ y = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2) \end{cases}$$

Suppose the lengths, in consistent physical units, of the two links are $L_1 = 1$ and $L_2 = 2$, and that $x = 2.5$, $y = 1.4$. Find the joint angles θ_1 and θ_2 (in radians) using Newton's method with an initial estimate of $(0.8, 0.9)$, tolerance $\varepsilon = 10^{-4}$, and maximum number of iterations set to 10.

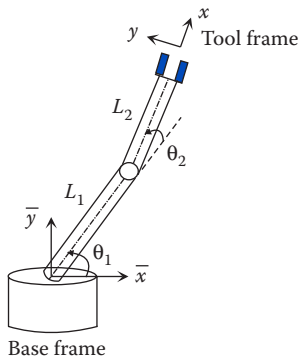


FIGURE 4.8

A two-link arm in plane motion.

93. Solve the following system of three nonlinear equations in three unknowns

$$\begin{cases} x^2 + y^2 = 2z \\ x^2 + z^2 = \frac{1}{3} \\ x^2 + y^2 + z^2 = 1 \end{cases}$$

using Newton's method with initial estimate (1, 1, 0.1), tolerance $\varepsilon = 10^{-4}$, and a maximum of 10 iterations.

94. Solve the following nonlinear system using Newton's method, with initial estimate (0, 1, 1), tolerance $\varepsilon = 10^{-3}$, and kmax = 10.

$$\begin{cases} xy - \cos x + z^2 = 3.6 \\ x^2 - 2y^2 + z = 2.8 \\ 3x + y \sin z = 2.8 \end{cases}$$

95. Consider the nonlinear system

$$\begin{cases} xy^2 - \sin x = 0 \\ (x-1)^3 y + 1 = 0 \end{cases}$$

With iteration functions

$$g_1(x, y) = \frac{\sin x}{y^2}, \quad g_2(x, y) = \frac{-1}{(x-1)^3}$$

use the fixed-point iteration method with initial estimate (2, -2), tolerance $\varepsilon = 10^{-3}$, and maximum 20 iterations to estimate one solution of the system.

96. Consider the system

$$\begin{cases} y = x^2 - 4 \\ x^2 + y^2 = 10 \end{cases}$$

- Locate the roots graphically.
- Find the root in the first quadrant by using the fixed-point iteration with iteration functions

$$g_1(x, y) = \sqrt{y+4}, \quad g_2(x, y) = \sqrt{10-x^2}$$

with (2, 0) as initial estimate, $\varepsilon = 10^{-3}$, and maximum number of iterations set to 20.

- There are three other roots, one in each quadrant. Find these roots by using combinations of the same g_1 and g_2 with positive and negative square roots.

97.  Consider the nonlinear system

$$\begin{cases} 2e^x + y = 0 \\ 3x^2 + 4y^2 = 8 \end{cases}$$

- Locate the roots graphically.
- Based on the location of the roots, select suitable iteration functions g_1 and g_2 and apply the fixed-point iteration method with $(-1, -2)$ as the initial estimate, $\varepsilon = 10^{-4}$, and number of iterations not to exceed 20, to find one of the two roots.
- To find the other root, write the original equations in reverse order, suitably select g_1 and g_2 , and apply fixed-point iteration with all information as in (b).

98.  Consider the nonlinear system

$$\begin{cases} x^2 + y^2 = 1.2 \\ 2x^2 + 3y^2 = 3 \end{cases}$$

- Locate the roots using a graphical approach.
- Select iteration functions

$$g_1(x, y) = \sqrt{1.2 - y^2}, \quad g_2(x, y) = \sqrt{\frac{3 - 2x^2}{3}}$$

and apply the fixed-point iteration method with initial estimate $(0.25, 1)$ and tolerance $\varepsilon = 10^{-4}$ to find a root. Decide the maximum number of iterations to be performed.

- There are three other roots, one in each of the remaining quadrants. Find these roots by using combinations of the same g_1 and g_2 with positive and negative square roots.

Curve Fitting and Interpolation

A set of data may emanate from various sources. In many engineering and scientific applications, the data originates from conducting experiments that involve measurement of physical quantities; for instance, measuring the displacement of a coiled spring when subjected to tensile or compressive force. In other cases, the data may be generated as a result of using numerical methods; for instance, numerical solution of differential equations (Chapters 7, 8, and 10).

An available set of data can be used for different purposes. In some cases, the data is represented by a function, which in turn can be used for numerical differentiation or integration (Chapter 6). Such function may be obtained through curve fitting, or approximation, of the data. Curve fitting is a procedure where a function is used to fit a given set of data in the “best” possible manner without having to match the data exactly. As a result, while the function does not necessarily yield the exact value at the data points, overall it fits the set of data well. Several types of functions and polynomials of different degrees can be used for curve fitting purposes. Curve fitting is normally used when the data has substantial inherent error, such as data gathered from experimental measurements. The aforementioned function or polynomial can then be used for interpolation purposes; that is, to find estimates of values at intermediate points (points between the given data points) where the data is not directly available.

In other situations, a single interpolating polynomial is sought that agrees exactly with the data points, and used to find estimates of values at intermediate points. For a small set of data, a single interpolating polynomial may be adequate. For large sets of data, however, different polynomials are used in different intervals of the whole data. This is referred to as spline interpolation.

5.1 Least-Squares Regression

As mentioned above, a single polynomial may be sufficient for interpolation of a small set of data. However, when the data has substantial error, even if the size of data is small, this may no longer be appropriate. Consider Figure 5.1, which shows a set of seven data points collected from an experiment. The nature of the data suggests that, for the most part, the y values increase with the x values. A single interpolating polynomial goes through all of the data points, but displays large oscillations in some regions. As a result, the interpolated values near $x = 1.2$ and $x = 2.85$, for instance, will be well outside of the range of the original data.

In these types of situations, it makes more sense to find a function that does not necessarily go through all of the data points, but fits the data well overall. One option, for example, is to fit the “best” straight line into the data. This line is not random and can be generated systematically via least-squares regression.

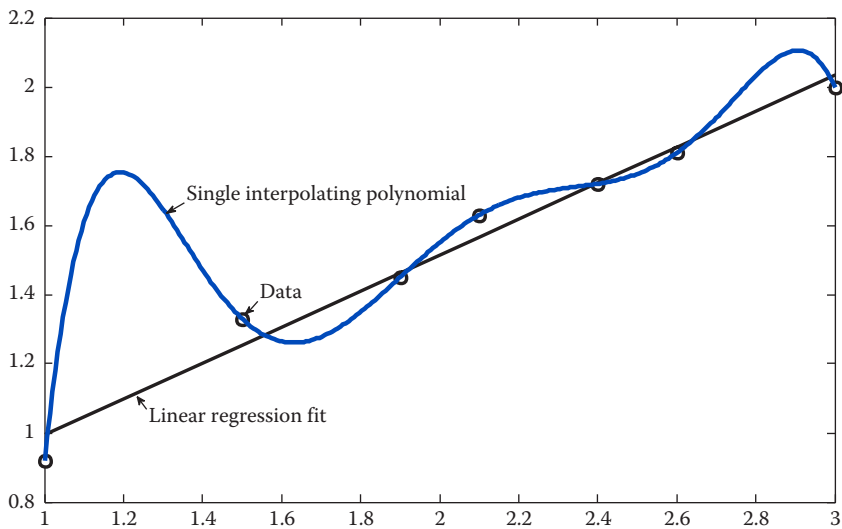


FIGURE 5.1
Interpolation by a single polynomial, and linear regression fit of a set of data.

5.2 Linear Regression

The simplest case of a least-squares regression involves finding a straight line (linear function) in the form

$$y = a_1x + a_0 \quad (5.1)$$

that best fits a set of n data points $(x_1, y_1), \dots, (x_n, y_n)$. Of course, the data first needs to be plotted to see whether the independent and dependent variables exhibit a somewhat linear relationship. If this is the case, then the coefficients a_1 and a_0 are determined such that the error associated with the line is minimized. As shown in Figure 5.2, at each data point

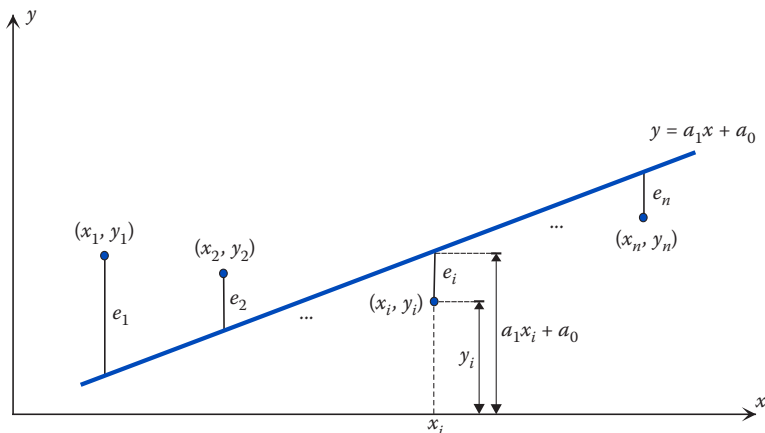


FIGURE 5.2
A linear fit of data, and individual errors.

(x_i, y_i) the error e_i is defined as the difference between the true value y_i and the approximate value $a_1x_i + a_0$,

$$e_i = y_i - (a_1x_i + a_0) \tag{5.2}$$

These individual errors will be used to calculate a total error associated with the line $y = a_1x + a_0$.

5.2.1 Deciding a “Best” Fit Criterion

Different strategies can be considered for determining the best linear fit of a set of n data points $(x_1, y_1), \dots, (x_n, y_n)$. One strategy is to minimize the sum of all the individual errors,

$$E = \sum_{i=1}^n e_i = \sum_{i=1}^n [y_i - (a_1x_i + a_0)] \tag{5.3}$$

This criterion, however, does not offer a good measure of how well the line fits the data because, as shown in [Figure 5.3](#), it allows for positive and negative individual errors—even very large errors—to cancel out and yield a zero sum.

Another strategy is to minimize the sum of the absolute values of the individual errors,

$$E = \sum_{i=1}^n |e_i| = \sum_{i=1}^n |y_i - (a_1x_i + a_0)| \tag{5.4}$$

As a result, the individual errors can no longer cancel out and the total error is always positive. This criterion, however, is not able to uniquely determine the coefficients that describe the best line fit because for a given set of data, several lines can have the same total error. [Figure 5.4](#) shows a set of four data points with two line fits that have the same total error.

The third strategy is to minimize the sum of the squares of the individual errors,

$$E = \sum_{i=1}^n e_i^2 = \sum_{i=1}^n [y_i - (a_1x_i + a_0)]^2 \tag{5.5}$$

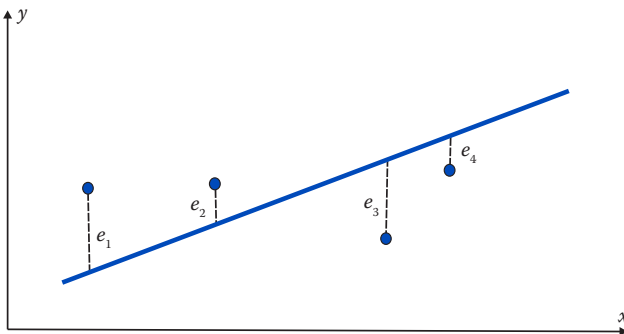


FIGURE 5.3
Zero total error based on the criterion defined by Equation 5.3.

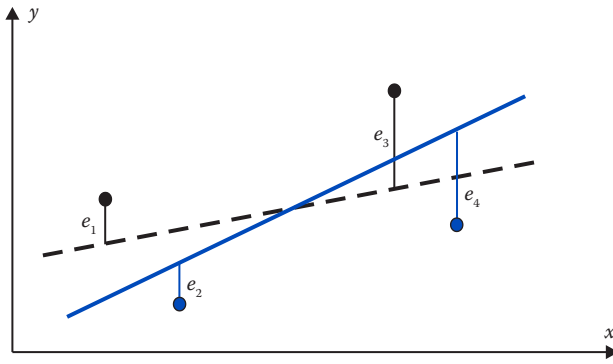


FIGURE 5.4

Two linear fits with the same total error calculated by Equation 5.4.

This criterion uniquely determines the coefficients that describe the best line fit for a given set of data. As in the second strategy, individual errors cannot cancel each other out and the total error is always positive. Also note that small errors get smaller and large errors get larger. This means that larger individual errors have larger contributions to the total error being minimized so that this strategy essentially minimizes the maximum distance that an individual data point is located relative to the line.

5.2.2 Linear Least-Squares Regression

As decided above, the criterion to find the line $y = a_1x + a_0$ that best fits the data $(x_1, y_1), \dots, (x_n, y_n)$ is to determine the coefficients a_1 and a_0 that minimize

$$E = \sum_{i=1}^n [y_i - (a_1x_i + a_0)]^2 \quad (5.6)$$

Noting that E is a (nonlinear) function of a_0 and a_1 , it attains its minimum where $\partial E/\partial a_0$ and $\partial E/\partial a_1$ vanish, that is,

$$\begin{aligned} \frac{\partial E}{\partial a_0} &= -2 \sum_{i=1}^n [y_i - (a_1x_i + a_0)] = 0 \quad \Rightarrow \quad \sum_{i=1}^n [y_i - (a_1x_i + a_0)] = 0 \\ \frac{\partial E}{\partial a_1} &= -2 \sum_{i=1}^n x_i [y_i - (a_1x_i + a_0)] = 0 \quad \Rightarrow \quad \sum_{i=1}^n \{x_i [y_i - (a_1x_i + a_0)]\} = 0 \end{aligned}$$

Expanding and rearranging the above equations, yield a system of two linear equations to be solved for a_0 and a_1 :

$$\begin{aligned} na_0 + \left(\sum_{i=1}^n x_i \right) a_1 &= \sum_{i=1}^n y_i \\ \left(\sum_{i=1}^n x_i \right) a_0 + \left(\sum_{i=1}^n x_i^2 \right) a_1 &= \sum_{i=1}^n x_i y_i \end{aligned}$$

By Cramer's rule, the solutions are found as

$$a_1 = \frac{n \left(\sum_{i=1}^n x_i y_i \right) - \left(\sum_{i=1}^n x_i \right) \left(\sum_{i=1}^n y_i \right)}{n \left(\sum_{i=1}^n x_i^2 \right) - \left(\sum_{i=1}^n x_i \right)^2}, \quad a_0 = \frac{\left(\sum_{i=1}^n x_i^2 \right) \left(\sum_{i=1}^n y_i \right) - \left(\sum_{i=1}^n x_i \right) \left(\sum_{i=1}^n x_i y_i \right)}{n \left(\sum_{i=1}^n x_i^2 \right) - \left(\sum_{i=1}^n x_i \right)^2} \quad (5.7)$$

The user-defined function `LinearRegression` uses the linear least-squares regression approach to find the straight line that best fits a set of data. The function plots this line, as well as the original data.

```
function [a1, a0] = LinearRegression(x,y)
%
% LinearRegression uses linear least-squares approximation to fit a data
% by a line in the form y = a1*x + a0. It also returns the plot of the
% original data together with the best line fit.
%
% [a1, a0] = LinearRegression(x,y), where
%
% x, y are n-dimensional row or column vectors of data,
%
% a1 and a0 are the coefficients that describe the linear fit.
%
n = length(x);
Sumx = sum(x); Sumy = sum(y); Sumxx = sum(x.*x); Sumxy = sum(x.*y);
den = n*Sumxx - Sumx^2;
a1 = (n*Sumxy - Sumx*Sumy)/den; a0 = (Sumxx*Sumy - Sumxy*Sumx)/den;
% Plot the data and the line fit
l = zeros(n,1); % Pre-allocate
for i = 1:n,
    l(i) = a1*x(i) + a0; % Calculate n points on the line
end
plot(x,y,'o')
hold on
plot(x,l)
end
```

EXAMPLE 5.1: LINEAR LEAST-SQUARES REGRESSION

Consider the data in [Table 5.1](#).

TABLE 5.1

Data in Example 5.1

x_i	y_i
0.2	8.2
0.4	8.4
0.6	8.5
0.8	8.6
1.0	8.8
1.2	8.7

- Using least-squares regression, find a straight line that best fits the data.
- Confirm the results by executing the user-defined function `LinearRegression`.

Solution

- Noting $n = 6$, we first calculate all the essential sums involved in Equation 5.7:

$$\sum_{i=1}^6 x_i = 0.2 + 0.4 + \dots + 1.2 = 4.2, \quad \sum_{i=1}^6 y_i = 8.0 + 8.4 + \dots + 8.7 = 51.2$$

$$\sum_{i=1}^6 x_i^2 = (0.2)^2 + (0.4)^2 + \dots + (1.2)^2 = 3.64$$

$$\sum_{i=1}^6 x_i y_i = (0.2)(8.0) + (0.4)(8.4) + \dots + (1.2)(8.7) = 36.22$$

Then, following Equation 5.7, the coefficients are found as

$$a_1 = \frac{(6)(36.22) - (4.2)(51.2)}{(6)(3.64) - (4.2)^2} = 0.5429, \quad a_0 = \frac{(3.64)(51.2) - (4.2)(36.22)}{(6)(3.64) - (4.2)^2} = 8.1533$$

Therefore, the line that best fits the data is described by

$$y = 0.5429x + 8.1533$$

- Execution of `LinearRegression` yields the coefficients a_1 and a_0 , which describe the best line fit, as well as the plot of the line and the original set of data; [Figure 5.5](#).

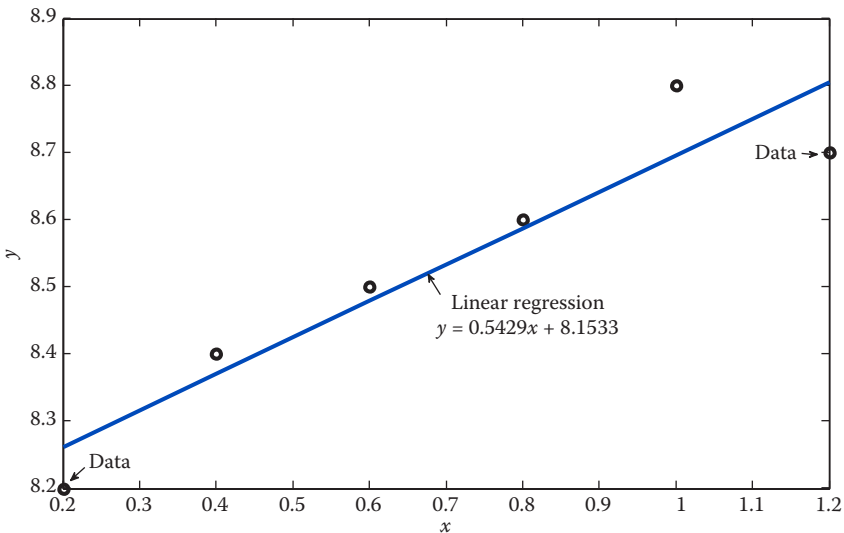


FIGURE 5.5

Data and the best line fit in Example 5.1.

```
>> x = 0.2:0.2:1.2; y = [8.2 8.4 8.5 8.6 8.8 8.7];
>> [a1, a0] = LinearRegression(x,y)

a1 =
    0.5429

a0 =
    8.1533
```

5.3 Linearization of Nonlinear Data

If the relationship between the independent and dependent variables is not linear, curve-fitting techniques other than linear regression must be used. One such method is polynomial regression, to be discussed in Section 5.4. Others involve conversion of the data into a form that could be handled by linear regression. Three examples of nonlinear functions that are used for curve fitting are as follows.

5.3.1 Exponential Function

The exponential function is in the form

$$y = ae^{bx} \quad (a, b = \text{const}) \quad (5.8)$$

Because differentiation of the exponential function returns a constant multiple of the exponential function, this technique applies to situations where the rate of change of a quantity is directly proportional to the quantity itself; for instance, radioactive decay. Conversion into linear form is made by taking the natural logarithm of Equation 5.8 to obtain

$$\ln y = bx + \ln a \quad (5.9)$$

Therefore, the plot of $\ln y$ versus x is a straight line with slope b and intercept $\ln a$; see [Figure 5.6a](#) and [d](#).

5.3.2 Power Function

Another example of a nonlinear function is the power function

$$y = ax^b \quad (a, b = \text{const}) \quad (5.10)$$

Linearization is achieved by taking the standard (base 10) logarithm of Equation 5.10,

$$\log y = b \log x + \log a \quad (5.11)$$

so that the plot of $\log y$ versus $\log x$ is a straight line with slope b and intercept $\log a$; see [Figure 5.6b](#) and [e](#).

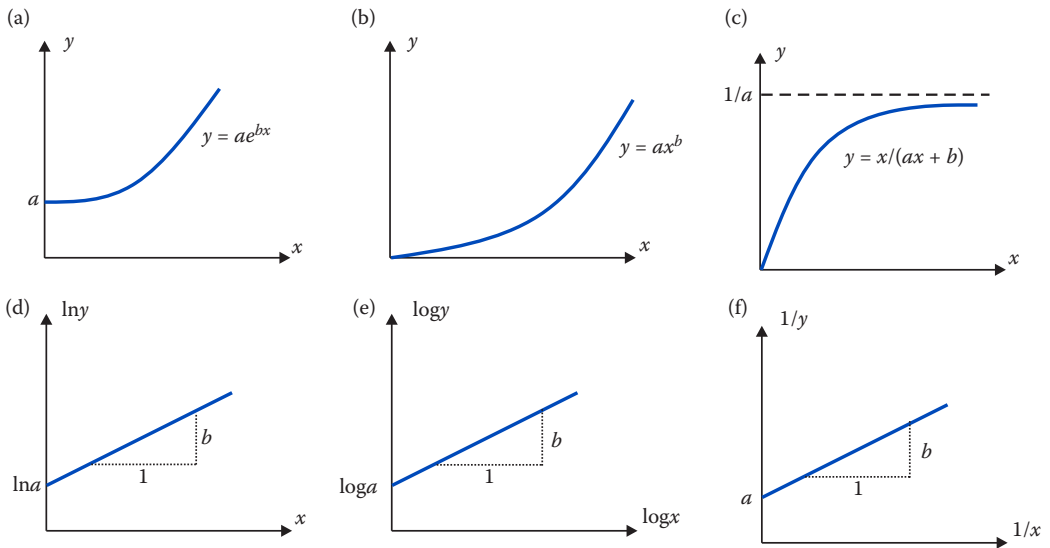


FIGURE 5.6

Linearization of three nonlinear functions for curve fitting. (a,d) Exponential function, (b,e) Power function, (c,f) Saturation function.

5.3.3 Saturation Function

The saturation function is in the form

$$y = \frac{x}{ax + b} \quad (a, b = \text{const}) \quad (5.12)$$

Inverting Equation 5.12 yields

$$\frac{1}{y} = b \left(\frac{1}{x} \right) + a \quad (5.13)$$

so that the plot of $1/y$ versus $1/x$ is a straight line with slope b and intercept a ; see [Figure 5.6c](#) and [f](#).

EXAMPLE 5.2: LINEARIZATION OF NONLINEAR DATA

Consider the data in [Table 5.2](#).

The data must first be plotted before any specific approach is selected. Plot of the data ([Figure 5.7](#) [left]) reveals that the saturation function *may be* suitable for a curve fit. To confirm that the saturation function is indeed the right choice, we need to determine whether the plot of $1/y$ versus $1/x$ is somewhat linear.

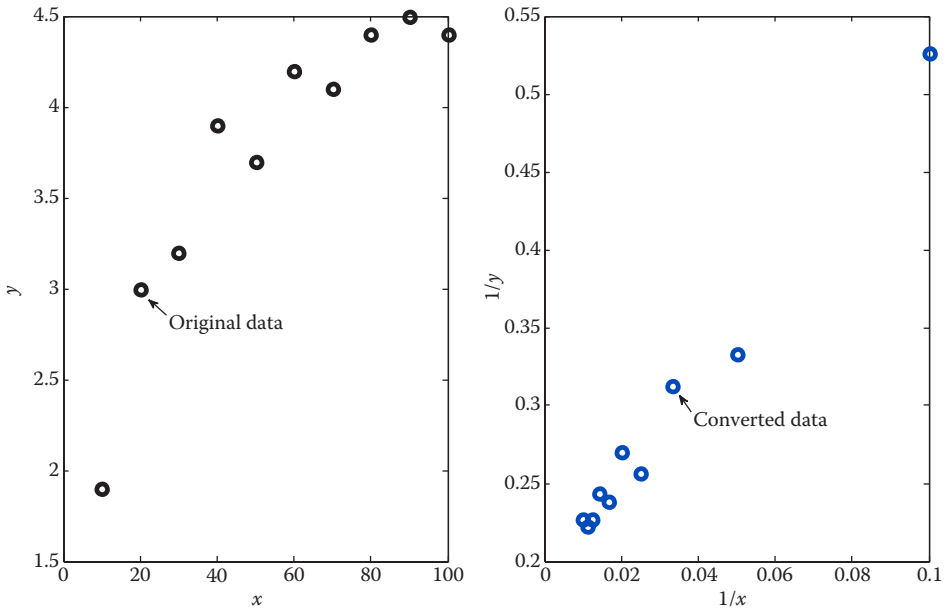
```
>> x = 10:10:100; y = [1.9 3.0 3.2 3.9 3.7 4.2 4.1 4.4 4.5 4.4];
>> xx = 1./x; yy = 1./y; % Element-by-element reciprocals
>> subplot(1,2,1), plot(x,y,'o') % Figure 5.7
>> subplot(1,2,2), plot(xx,yy,'o')
```

The plot of $1/y$ versus $1/x$ ([Figure 5.7](#) [right]) in fact shows that the converted data behaves in a somewhat linear manner. Therefore, we will proceed with the saturation function fit. We will apply linear regression to this converted data to find the slope and

TABLE 5.2

Data in Example 5.2

x	y
10	1.9
20	3.0
30	3.2
40	3.9
50	3.7
60	4.2
70	4.1
80	4.4
90	4.5
100	4.4

**FIGURE 5.7**

Plot of the data in Table 5.2 (left) and converted data (right).

the intercept of the line fit. Execution of `LinearRegression` provides this information and also plots the result; Figure 5.8.

```
>> [a1, a0] = LinearRegression(xx,yy)
a1 =
    3.3052
a0 =
    0.1890    % Figure 5.8 is returned by the function
```

Based on the form in Equation 5.13, we have $b = 3.3052$ (slope) and $a = 0.1890$ (intercept). Consequently, the saturation function of interest is formed as

$$y = \frac{x}{ax + b} = \frac{x}{0.1890x + 3.3052}$$

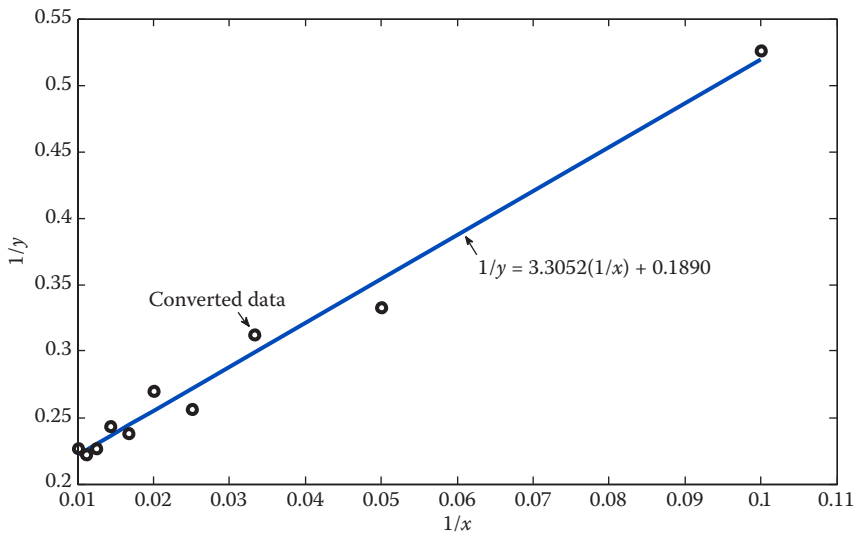


FIGURE 5.8
Linear fit of the converted data in Example 5.2.

The plot of this function together with the original data (Figure 5.9) clearly shows that the saturation function provides a reasonable fit of the given data.

```
x_fit = linspace(10,100);
for i = 1:100,
y_fit(i) = x_fit(i)/(0.1890*x_fit(i)+3.3052);
end
plot(x,y,'o')
hold on
plot(x_fit,y_fit) % Figure 5.9
```

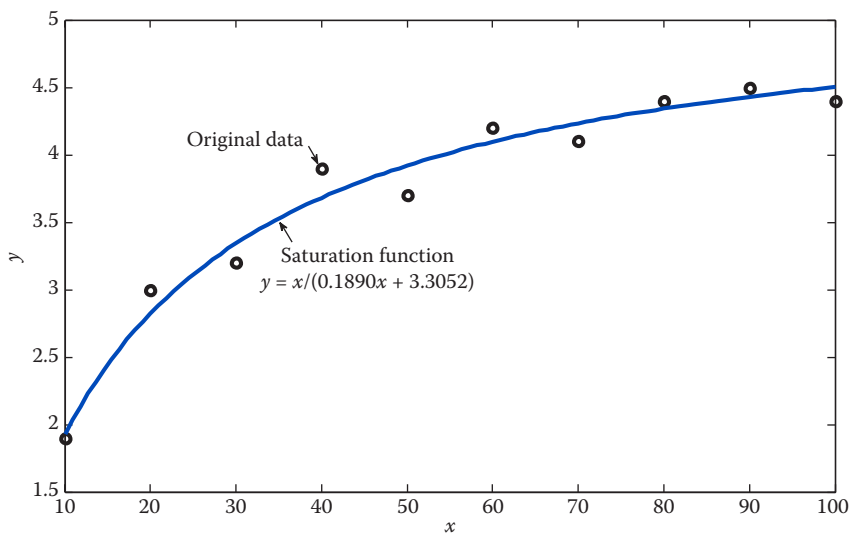


FIGURE 5.9
Curve fit using the saturation function; Example 5.2.

EXAMPLE 5.3: LINEARIZATION OF NONLINEAR DATA

Consider the data in [Table 5.3](#).

The data is first plotted before any specific approach is selected. Plot of the data ([Figure 5.10](#) [left]) suggests that the power function *maybe* suitable for a curve fit. To confirm this, we need to determine whether the plot of $\log y$ versus $\log x$ is somewhat linear.

```
>> x = 0.1:0.1:1; y = [0.02 0.1 0.2 0.35 0.56 0.75 1.04 1.3 1.7 2.09];  
>> xx = log10(x); yy = log10(y);  
>> subplot(1,2,1), plot(x,y,'o') % Figure 5.10  
>> subplot(1,2,2), plot(xx,yy,'o')
```

Since the plot of $\log y$ versus $\log x$ ([Figure 5.10](#) [right]) confirms a somewhat linear relationship, we proceed with the power function fit. We will apply linear regression

TABLE 5.3

Data in Example 5.3

x	y
0.1	0.02
0.2	0.10
0.3	0.20
0.4	0.35
0.5	0.56
0.6	0.75
0.7	1.04
0.8	1.30
0.9	1.70
1.0	2.09

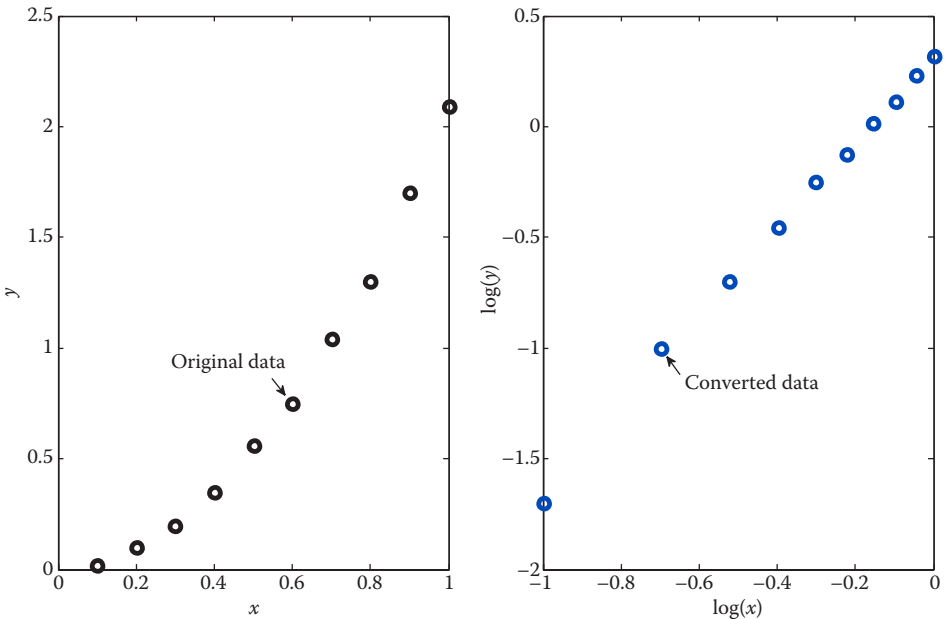


FIGURE 5.10

Plot of the data in [Table 5.3](#) (left) and converted data (right).

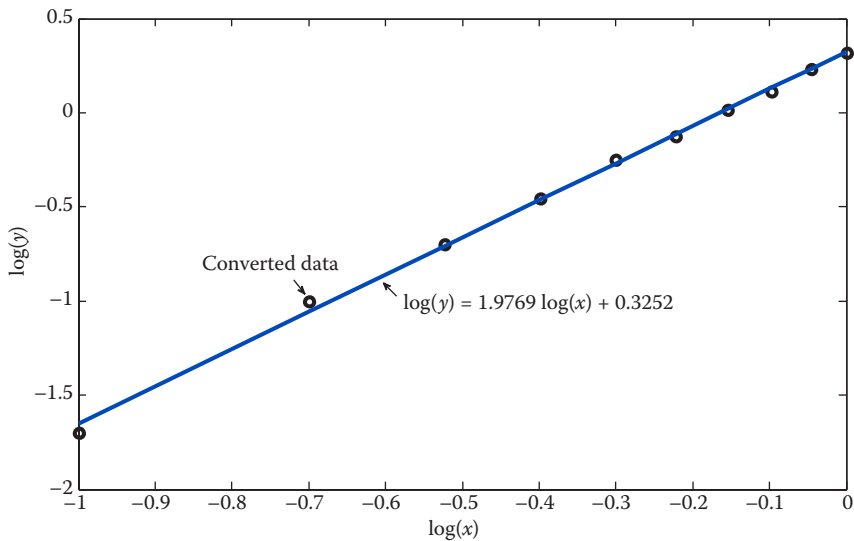


FIGURE 5.11

Linear fit of the converted data in Example 5.3.

to this converted data to find the slope and the intercept of the line fit. Execution of `LinearRegression` returns this information and also plots the result; [Figure 5.11](#).

```
>> [a1, a0] = LinearRegression(xx,yy)
a1 =
    1.9769
a0 =
    0.3252
```

Based on the form in Equation 5.11, we have $b = 1.9769$ (slope) and $\log a = 0.3252$ (intercept) so that $a = 2.1145$. The corresponding power function is

$$y = ax^b = 2.1145x^{1.9769}$$

The plot of this function together with the original data ([Figure 5.12](#)) clearly shows that the power function provides a reasonable fit of the given data.

```
x_fit = linspace(0.1,1);
for i = 1:100,
y_fit(i) = 2.1145*x_fit(i)^1.9769;
end
plot(x,y,'o')
hold on
plot(x_fit,y_fit) % Figure 5.12
```

5.4 Polynomial Regression

In the previous section we learned that a curve can fit into nonlinear data by transforming the data into a form that can be handled by linear regression. Another method is to fit polynomials of different orders to the data by means of polynomial regression.

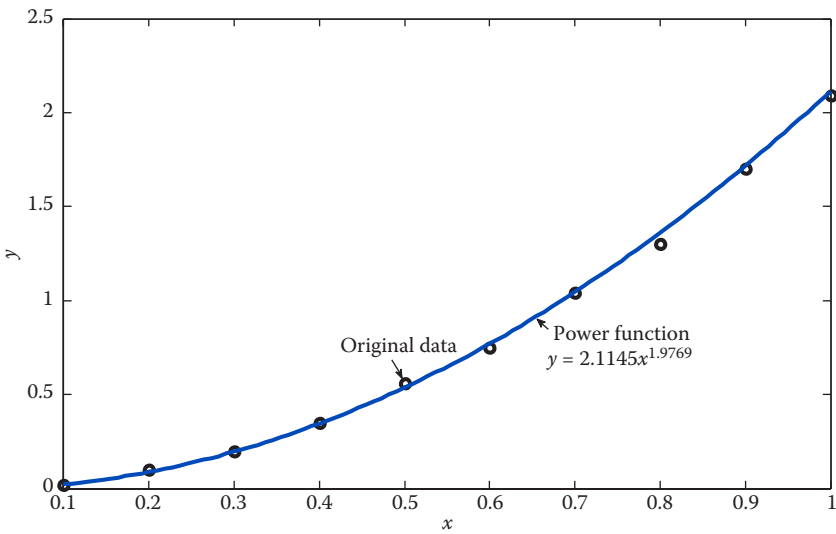


FIGURE 5.12

Curve fit using the power function; Example 5.3.

The linear least-squares regression of Section 5.2 can be extended to fit a set of n data points $(x_1, y_1), \dots, (x_n, y_n)$ with an m th-degree polynomial in the form

$$y = a_m x^m + a_{m-1} x^{m-1} + \dots + a_2 x^2 + a_1 x + a_0$$

with a total error

$$E = \sum_{i=1}^n \left[y_i - (a_m x_i^m + \dots + a_2 x_i^2 + a_1 x_i + a_0) \right]^2 \quad (5.14)$$

The coefficients $a_m, \dots, a_2, a_1, a_0$ are determined such that E is minimized. A necessary condition for E to attain a minimum is that its partial derivative with respect to each of these coefficients vanishes, that is,

$$\begin{aligned} \frac{\partial E}{\partial a_0} &= -2 \sum_{i=1}^n \left[y_i - (a_m x_i^m + \dots + a_2 x_i^2 + a_1 x_i + a_0) \right] = 0 \\ \frac{\partial E}{\partial a_1} &= -2 \sum_{i=1}^n \left\{ x_i \left[y_i - (a_m x_i^m + \dots + a_2 x_i^2 + a_1 x_i + a_0) \right] \right\} = 0 \\ \frac{\partial E}{\partial a_2} &= -2 \sum_{i=1}^n \left\{ x_i^2 \left[y_i - (a_m x_i^m + \dots + a_2 x_i^2 + a_1 x_i + a_0) \right] \right\} = 0 \\ &\dots \qquad \qquad \qquad \dots \qquad \qquad \dots \\ \frac{\partial E}{\partial a_m} &= -2 \sum_{i=1}^n \left\{ x_i^m \left[y_i - (a_m x_i^m + \dots + a_2 x_i^2 + a_1 x_i + a_0) \right] \right\} = 0 \end{aligned} \quad (5.15)$$

Manipulation of these equations yields a system of $m + 1$ linear equations to be solved for $a_m, \dots, a_2, a_1, a_0$:

$$\begin{aligned}
 na_0 + \left(\sum_{i=1}^n x_i \right) a_1 + \left(\sum_{i=1}^n x_i^2 \right) a_2 + \dots + \left(\sum_{i=1}^n x_i^m \right) a_m &= \sum_{i=1}^n y_i \\
 \left(\sum_{i=1}^n x_i \right) a_0 + \left(\sum_{i=1}^n x_i^2 \right) a_1 + \left(\sum_{i=1}^n x_i^3 \right) a_2 + \dots + \left(\sum_{i=1}^n x_i^{m+1} \right) a_m &= \sum_{i=1}^n x_i y_i \\
 \left(\sum_{i=1}^n x_i^2 \right) a_0 + \left(\sum_{i=1}^n x_i^3 \right) a_1 + \left(\sum_{i=1}^n x_i^4 \right) a_2 + \dots + \left(\sum_{i=1}^n x_i^{m+2} \right) a_m &= \sum_{i=1}^n x_i^2 y_i \\
 \dots & \dots \dots \\
 \left(\sum_{i=1}^n x_i^m \right) a_0 + \left(\sum_{i=1}^n x_i^{m+1} \right) a_1 + \left(\sum_{i=1}^n x_i^{m+2} \right) a_2 + \dots + \left(\sum_{i=1}^n x_i^{2m} \right) a_m &= \sum_{i=1}^n x_i^m y_i
 \end{aligned} \tag{5.16}$$

5.4.1 Quadratic Least-Squares Regression

The objective is to fit a set of n data points $(x_1, y_1), \dots, (x_n, y_n)$ with a second-degree polynomial

$$y = a_2x^2 + a_1x + a_0$$

such that the total error

$$E = \sum_{i=1}^n \left[y_i - (a_2x_i^2 + a_1x_i + a_0) \right]^2$$

is minimized. Following the procedure outlined above, which ultimately led to Equation 5.16, the coefficients a_2, a_1, a_0 are determined by solving a system of three linear equations:

$$\begin{aligned}
 na_0 + \left(\sum_{i=1}^n x_i \right) a_1 + \left(\sum_{i=1}^n x_i^2 \right) a_2 &= \sum_{i=1}^n y_i \\
 \left(\sum_{i=1}^n x_i \right) a_0 + \left(\sum_{i=1}^n x_i^2 \right) a_1 + \left(\sum_{i=1}^n x_i^3 \right) a_2 &= \sum_{i=1}^n x_i y_i \\
 \left(\sum_{i=1}^n x_i^2 \right) a_0 + \left(\sum_{i=1}^n x_i^3 \right) a_1 + \left(\sum_{i=1}^n x_i^4 \right) a_2 &= \sum_{i=1}^n x_i^2 y_i
 \end{aligned} \tag{5.17}$$

The user-defined function `QuadraticRegression` uses the quadratic least-squares regression approach to find the second-degree polynomial that best fits a set of data. The coefficients a_2, a_1, a_0 are found by writing Equation 5.17 in matrix form and applying the built-in backslash “\” operator in MATLAB. The function also returns the plot of the data and the best quadratic polynomial fit.

```

function [a2, a1, a0] = QuadraticRegression(x,y)
%
% QuadraticRegression uses quadratic least-squares approximation to fit a
% data by a 2nd-degree polynomial in the form  $y = a_2x^2 + a_1x + a_0$ .
%
% [a2, a1, a0] = QuadraticRegression(x,y), where
%
% x, y are n-dimensional row or column vectors of data,
%
% a2, a1 and a0 are the coefficients that describe the quadratic fit.
%
n = length(x);
Sumx = sum(x); Sumy = sum(y);
Sumx2 = sum(x.^2); Sumx3 = sum(x.^3); Sumx4 = sum(x.^4);
Sumxy = sum(x.*y); Sumx2y = sum(x.*x.*y);

% Form the coefficient matrix and the vector of right-hand sides
A = [n Sumx Sumx2; Sumx Sumx2 Sumx3; Sumx2 Sumx3 Sumx4];
b = [Sumy; Sumxy; Sumx2y];
w = A\b; % Solve for the coefficients
a2 = w(3); a1 = w(2); a0 = w(1);
% Plot the data and the quadratic fit
xx = linspace(x(1),x(end)); % Generate 100 points for plotting purposes
p = zeros(100,1); % Pre-allocate
for i = 1:100,
    p(i) = a2*xx(i)^2+a1*xx(i)+a0; % Calculate 100 points
end
plot(x,y,'o')
hold on
plot(xx,p)
end

```

EXAMPLE 5.4: QUADRATIC REGRESSION

Using quadratic least-squares regression find the second-degree polynomial that best fits the data in [Table 5.4](#).

Solution

```

>> x = 0:0.4:1.6; y = [2.90 3.10 3.56 4.60 6.70];
>> [a2, a1, a0] = QuadraticRegression(x,y)

a2 =
    1.9554

a1 =
   -0.8536

a0 =
    2.9777

```

The plot returned by the user-defined function is shown in [Figure 5.13](#).

TABLE 5.4

Data in Example 5.4

x	y
0	2.90
0.4	3.10
0.8	3.56
1.2	4.60
1.6	6.70

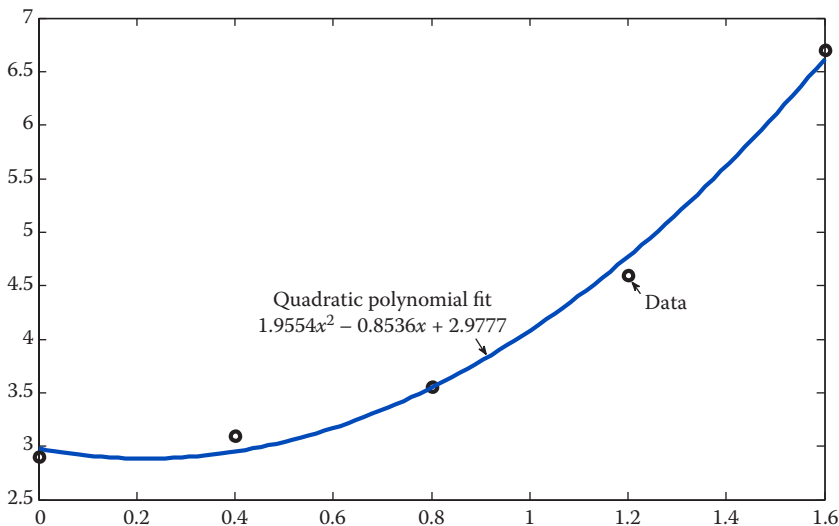


FIGURE 5.13

Quadratic polynomial fit in Example 5.4.

5.4.2 Cubic Least-Squares Regression

The objective is to fit a set of n data points $(x_1, y_1), \dots, (x_n, y_n)$ with a third-degree polynomial

$$y = a_3x^3 + a_2x^2 + a_1x + a_0$$

such that the total error

$$E = \sum_{i=1}^n \left[y_i - (a_3x_i^3 + a_2x_i^2 + a_1x_i + a_0) \right]^2$$

is minimized. Proceeding as always, a_3, a_2, a_1, a_0 are determined by solving a system of four linear equations generated by Equation 5.16.

A user-defined function (see Problem Set) with function call $[a_3, a_2, a_1, a_0] = \text{Cubic Regression}(x, y)$ can then be written that uses the cubic least-squares regression approach to find the third-degree polynomial that best fits a set of data. The coefficients a_3, a_2, a_1, a_0 are found by expressing the appropriate 4×4 system of equations in matrix form and solving by “\” in MATLAB. The function should also return the plot of the original data and the best cubic polynomial fit.

EXAMPLE 5.5: CUBIC REGRESSION

Find the cubic polynomial that best fits the data in Example 5.4. Plot the quadratic and cubic polynomial fits in one graph and compare.

Solution

```
>> x = 0:0.4:1.6; y = [2.90 3.10 3.56 4.60 6.70];  
>> [a3, a2, a1, a0] = CubicRegression(x,y) % See Problem Set  
  
a3 =  
    1.0417  
  
a2 =  
   -0.5446  
  
a1 =  
    0.5798  
  
a0 =  
    2.8977  
  
>> hold on  
>> [a2, a1, a0] = QuadraticRegression(x,y) % Previously done in Example 5.4  
  
a2 =  
    1.9554  
  
a1 =  
   -0.8536  
  
a0 =  
    2.9777
```

Figure 5.14 clearly shows that the cubic polynomial fit is superior to the quadratic one. The cubic polynomial fit almost goes through all five data points, but not exactly.

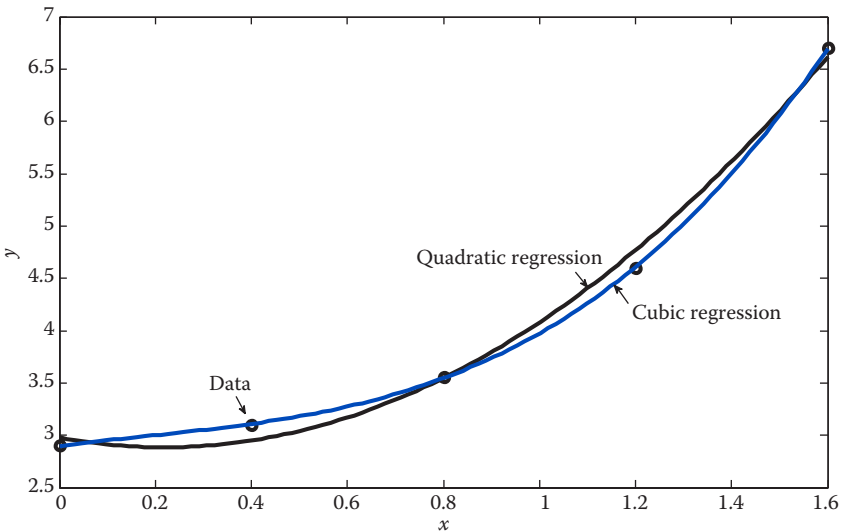


FIGURE 5.14 Quadratic and cubic polynomial fits; Example 5.5.

This is because a third-degree polynomial has four coefficients but the data contains five points. A fourth-degree polynomial, which has five coefficients, would exactly agree with the data in this example. In general, if a set of n data points is given, then the $(n - 1)$ th-degree polynomial that best fits the data will agree exactly with the data points. This is the main idea behind interpolation, and such polynomial is called an interpolating polynomial. We will discuss this topic in Section 5.5.

5.4.3 MATLAB Built-In Functions `polyfit` and `polyval`

A brief description of the MATLAB built-in function `polyfit` is given as:

`POLYFIT` Fit polynomial to data.

`P = POLYFIT(X,Y,N)` finds the coefficients of a polynomial $P(X)$ of degree N that fits the data Y best in a least-squares sense. P is a row vector of length $N+1$ containing the polynomial coefficients in descending powers, $P(1)*X^N + P(2)*X^{(N-1)} + \dots + P(N)*X + P(N+1)$.

This polynomial is then evaluated at any x using the built-in function `polyval`. Specifically, $y_i = \text{polyval}(P, x_i)$ returns the value of polynomial P evaluated at x_i .

EXAMPLE 5.6: CURVE FIT—`POLYFIT` FUNCTION

Using the `polyfit` and `polyval` functions find and plot the second-degree polynomial that best fits the data in [Table 5.5](#). Apply the user-defined function `QuadraticRegression` and compare the results.

Solution

```
>> x = 0:0.3:1.2; y = [3.6 4.8 5.9 7.6 10.9];  
>> P = polyfit(x,y,2)
```

`P =`

```
3.8095 1.2286 3.7657 % Coefficients of the second-deg polynomial fit
```

```
>> xi = linspace(0,1.2); % Generate 100 points for plotting purposes  
>> yi = polyval(P,xi); % Evaluate the polynomial at these points  
>> plot(xi,yi,x,y,'o') % Figure 5.15
```

Execution of `QuadraticRegression` yields:

```
>> [a2, a1, a0] = QuadraticRegression(x,y)
```

`a2 =`

```
3.8095
```

`a1 =`

```
1.2286
```

`a0 =`

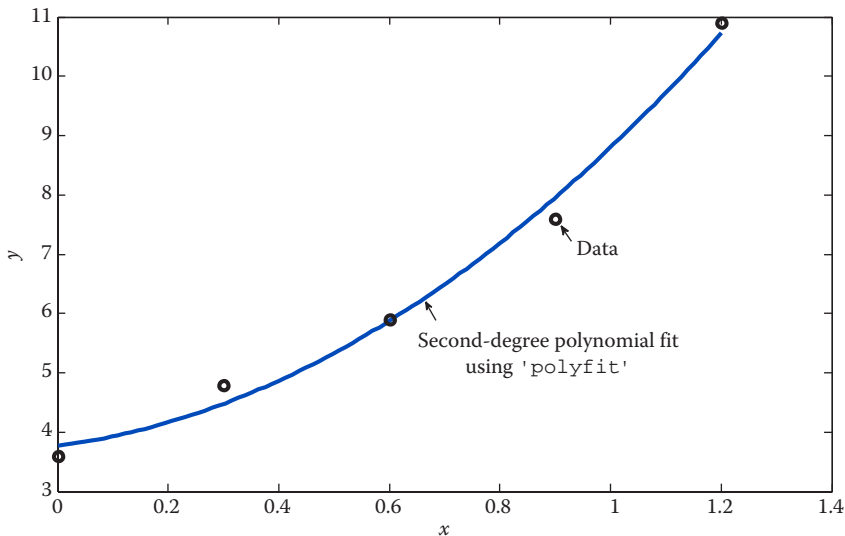
```
3.7657
```

The coefficients of the second-degree polynomial are precisely those returned by `polyfit`.

TABLE 5.5

Data in Example 5.6

x	y
0	3.6
0.3	4.8
0.6	5.9
0.9	7.6
1.2	10.9

**FIGURE 5.15**

Cubic polynomial fit using polyfit.

5.5 Polynomial Interpolation

Given a set of $n + 1$ data points $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$, there is only one polynomial of degree at most n in the form

$$p(x) = a_{n+1}x^n + a_nx^{n-1} + \dots + a_3x^2 + a_2x + a_1$$

that goes through all the points. Although this polynomial is unique, it can be expressed in different forms. The two most commonly used forms are provided by Lagrange interpolating polynomials and Newton interpolating polynomials, which are presented in this section.

5.5.1 Lagrange Interpolating Polynomials

The first-degree Lagrange interpolating polynomial that goes through the two points (x_1, y_1) and (x_2, y_2) is in the form

$$p_1(x) = L_1(x)y_1 + L_2(x)y_2$$

where $L_1(x)$ and $L_2(x)$ are the Lagrange coefficient functions described by

$$L_1(x) = \frac{x - x_2}{x_1 - x_2}, \quad L_2(x) = \frac{x - x_1}{x_2 - x_1}$$

Then $L_1(x_1) = 1$ and $L_1(x_2) = 0$, while $L_2(x_1) = 0$ and $L_2(x_2) = 1$. Consequently, $p_1(x_1) = y_1$ and $p_1(x_2) = y_2$, which means the polynomial, in this case a straight line, passes through the two points; see [Figure 5.16a](#).

The second-degree Lagrange interpolating polynomial that goes through the three points (x_1, y_1) , (x_2, y_2) , and (x_3, y_3) is in the form

$$p_2(x) = L_1(x)y_1 + L_2(x)y_2 + L_3(x)y_3$$

where

$$L_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)}, \quad L_2(x) = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)}, \quad L_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}$$

Therefore, $L_1(x_1) = 1 = L_2(x_2) = L_3(x_3)$, while all other $L_i(x_j) = 0$ for $i \neq j$. This yields $p_2(x_1) = y_1$, $p_2(x_2) = y_2$, and $p_2(x_3) = y_3$ so that the polynomial goes through the three points; see [Figure 5.16b](#).

In general, the n th-degree Lagrange interpolating polynomial that goes through $n + 1$ points $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$ is formed as

$$p_n(x) = L_1(x)y_1 + \dots + L_{n+1}(x)y_{n+1} = \sum_{i=1}^{n+1} L_i(x)y_i \tag{5.18}$$

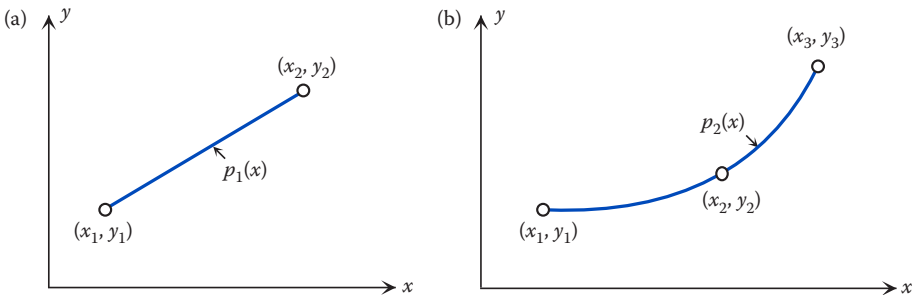


FIGURE 5.16

(a) First-degree and (b) second-degree Lagrange interpolating polynomials.

where each $L_i(x)$ is defined as

$$L_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{x - x_j}{x_i - x_j} \quad (5.19)$$

and “ Π ” denotes the product of terms.

The user-defined function `LagrangeInterp` finds the Lagrange interpolating polynomial that fits a set of data (x, y) and uses this polynomial to calculate the interpolated value y_i at a specified point x_i .

```
function yi = LagrangeInterp(x,y,xi)
%
% LagrangeInterp finds the Lagrange interpolating polynomial that goes
% through the data (x,y) and uses it to find the interpolated value
% at xi.
%
% yi = LagrangeInterp(x,y,xi), where
%
% x, y are n-dimensional row or column vectors of data,
% xi is a specified point,
%
% yi is the interpolated value at xi.
%
n = length(x);
L = zeros(1,n); % Pre-allocate
for i = 1:n,
    L(i) = 1;
    for j = 1:n,
        if j ~= i,
            L(i) = L(i)*(xi - x(j))/(x(i) - x(j));
        end
    end
end
yi = sum(y.*L);
```

EXAMPLE 5.7: LAGRANGE INTERPOLATION

Consider the data in [Table 5.6](#) generated by the function $y = 1 + e^{-x}$.

1. Find the first-degree Lagrange interpolating polynomial that goes through the first two data points, and use it to find the interpolated value at $x = 0.35$. Confirm the result by executing `LagrangeInterp`.

TABLE 5.6

Data in Example 5.7

x	$y = 1 + e^{-x}$
0.1	1.9048
0.5	1.6065
0.8	1.4493

- Find the second-degree Lagrange interpolating polynomial that goes through all three data points, and use it to find the interpolated value at $x = 0.35$. Confirm the result by executing `LagrangeInterp`.
- Calculate the % relative errors associated with the two estimates in (1) and (2), and comment.
- Using `LagrangeInterp`, plot the first-degree Lagrange interpolating polynomial generated in (1), the second-degree polynomial generated in (2), and the original data in the same graph.

Solution

- The two Lagrange coefficient functions are

$$L_1(x) = \frac{x - x_2}{x_1 - x_2} = \frac{x - 0.5}{0.1 - 0.5}, \quad L_2(x) = \frac{x - x_1}{x_2 - x_1} = \frac{x - 0.1}{0.5 - 0.1}$$

The first-degree polynomial is then formed as

$$p_1(x) = L_1(x)y_1 + L_2(x)y_2 = \frac{x - 0.5}{0.1 - 0.5}(1.9048) + \frac{x - 0.1}{0.5 - 0.1}(1.6065) \stackrel{\text{simplify}}{=} -0.7458x + 1.9794$$

Using this polynomial, we can interpolate at $x = 0.35$ to find

$$p_1(0.35) = 1.7184$$

The result can be readily verified in MATLAB as follows.

```
% Input the first two data points only
>> x = [0.1 0.5]; y = [1.9048 1.6065];
>> yi = LagrangeInterp(x,y,0.35)
```

```
yi =
    1.7184
```

- The three Lagrange coefficient functions are

$$L_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} = \frac{(x - 0.5)(x - 0.8)}{(0.1 - 0.5)(0.1 - 0.8)}$$

$$L_2(x) = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} = \frac{(x - 0.1)(x - 0.8)}{(0.5 - 0.1)(0.5 - 0.8)}$$

$$L_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} = \frac{(x - 0.1)(x - 0.5)}{(0.8 - 0.1)(0.8 - 0.5)}$$

The second-degree polynomial is then formed as

$$p_2(x) = L_1(x)y_1 + L_2(x)y_2 + L_3(x)y_3 = \frac{(x - 0.5)(x - 0.8)}{(0.1 - 0.5)(0.1 - 0.8)}(1.9048) + \frac{(x - 0.1)(x - 0.8)}{(0.5 - 0.1)(0.5 - 0.8)}(1.6065) + \frac{(x - 0.1)(x - 0.5)}{(0.8 - 0.1)(0.8 - 0.5)}(1.4493) \stackrel{\text{simplify}}{=} 0.3168x^2 - 0.9358x + 1.9952$$

Using this polynomial, we can interpolate at $x = 0.35$ to find

$$p_2(0.35) = 1.7065$$

The result is confirmed in MATLAB as

```
>> x = [0.1 0.5 0.8]; y = [1.9048 1.6065 1.4493];
>> yi = LagrangeInterp(x,y,0.35)

yi =
    1.7065
```

3. In calculating the errors we will use the entire values as stored in a calculator, as opposed to the values displayed showing only four decimal places. Noting $y_{exact} = 1 + e^{-0.35}$, the % relative errors are found as

$$\frac{p_1(0.35) - y_{exact}}{y_{exact}} \times 100 = \boxed{0.8026\%}$$

$$\frac{p_2(0.35) - y_{exact}}{y_{exact}} \times 100 = \boxed{0.1050\%}$$

The estimate provided by $p_2(x)$ is clearly superior because the data was more effectively utilized for interpolation.

4.

```
x = [0.1 0.5 0.8]; y = [1.9048 1.6065 1.4493];

% Specific data to build the first-degree polynomial
x1 = [0.1 0.5]; y1 = [1.9048 1.6065];

% 100 points for plotting over the entire (original) data
xi = linspace(0.1,0.8);

% Use the first-degree polynomial [based on (x1,y1)] to evaluate at
% the 100 points
for i = 1:100,
    yi(i) = LagrangeInterp(x1,y1,xi(i));
end

% Plot
plot(xi,yi,x,y,'o') % Figure 5.17
hold on

% Use the second-degree polynomial [based on (x,y)] to evaluate at
% the 100 points
for i = 1:100,
    yi(i) = LagrangeInterp(x,y,xi(i));
end

% Complete the figure
plot(xi,yi)
```

5.5.2 Drawbacks of Lagrange Interpolation

As observed in Example 5.7, none of the information gathered in the construction of $p_1(x)$ was saved and used in the construction of $p_2(x)$. This is always true when working with

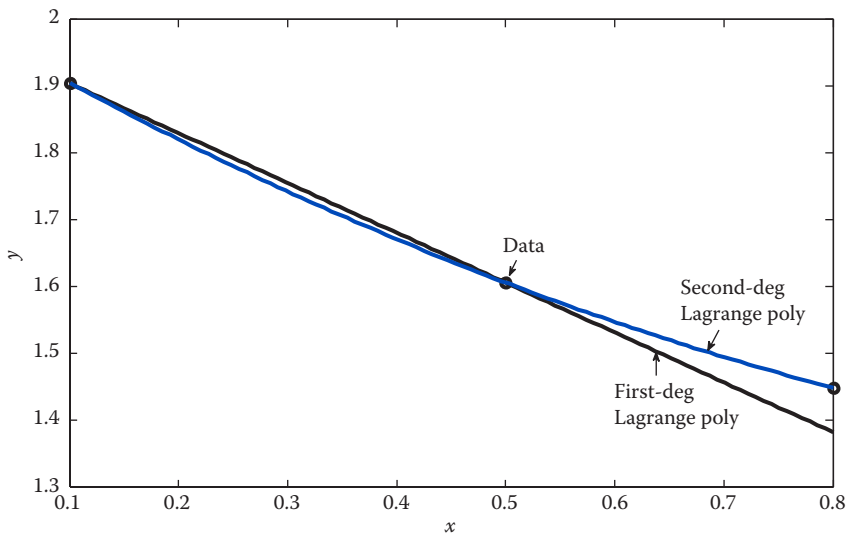


FIGURE 5.17

First-degree and second-degree Lagrange interpolating polynomials in Example 5.7.

Lagrange interpolating polynomials; that the quantities calculated while constructing a Lagrange polynomial cannot be stored and used in the construction of a higher-degree polynomial. This is particularly inconvenient in two situations: (1) when the exact degree of the interpolating polynomial is not known in advance; for instance, it might be better to use a portion of the set of data, or (2) when additional points are added to the data. In these cases, a more suitable form is provided by the Newton interpolating polynomials.

5.5.3 Newton Divided-Difference Interpolating Polynomials

Recall the notations we used in relation to [Figure 5.16](#), that is, $p_1(x)$ is the first-degree polynomial that goes through (x_1, y_1) and (x_2, y_2) , while $p_2(x)$ is the second-degree polynomial that goes through (x_1, y_1) , (x_2, y_2) , and (x_3, y_3) , etc. The Newton interpolating polynomials are *recursively* constructed as

$$\begin{aligned}
 p_1(x) &= a_1 + a_2(x - x_1) \\
 p_2(x) &= p_1(x) + a_3(x - x_1)(x - x_2) \\
 p_3(x) &= p_2(x) + a_4(x - x_1)(x - x_2)(x - x_3) \\
 &\dots \\
 p_n(x) &= P_{n-1}(x) + a_{n+1}(x - x_1)(x - x_2) \dots (x - x_n)
 \end{aligned}
 \tag{5.20}$$

where the coefficients a_1, a_2, \dots, a_{n+1} are determined inductively as follows: Since $p_1(x)$ must agree with the data at (x_1, y_1) and (x_2, y_2) , we have

$$p_1(x_1) = y_1, \quad p_1(x_2) = y_2$$

so that

$$\begin{aligned} a_1 + a_2(x_1 - x_1) &= y_1 \\ a_1 + a_2(x_2 - x_1) &= y_2 \end{aligned} \Rightarrow a_1 = y_1, \quad a_2 = \frac{y_2 - y_1}{x_2 - x_1}$$

Similarly, $p_2(x)$ must agree with the data at (x_1, y_1) , (x_2, y_2) , and (x_3, y_3) , hence

$$p_2(x_1) = y_1, \quad p_2(x_2) = y_2, \quad p_2(x_3) = y_3$$

But the added term in $p_2(x)$ is $a_3(x - x_1)(x - x_2)$, which vanishes when $x = x_1$ or $x = x_2$. Thus,

$$p_2(x_1) = p_1(x_1), \quad p_2(x_2) = p_1(x_2)$$

As a result, the two conditions $p_2(x_1) = y_1$ and $p_2(x_2) = y_2$ are simply $p_1(x_1) = y_1$ and $p_1(x_2) = y_2$, which already led to a_1 and a_2 in constructing $p_1(x)$. The third condition $p_2(x_3) = y_3$ yields

$$\boxed{p_1(x_3)} + a_3(x_3 - x_1)(x_3 - x_2) = y_3 \quad \Rightarrow \quad \boxed{a_1 + a_2(x_3 - x_1)} + a_3(x_3 - x_1)(x_3 - x_2) = y_3$$

Using the values of a_1 and a_2 obtained earlier, this equation then gives

$$a_3 = \frac{(y_3 - y_2)/(x_3 - x_2) - (y_2 - y_1)/(x_2 - x_1)}{x_3 - x_1}$$

Continuing this process yields all remaining coefficients. In the meantime, we observe that these coefficients follow a systematic pattern. First off, a_1 is simply the first data y_1 . Then, a_2 is the difference quotient involving the first two data points, a_3 is the difference quotient of difference quotients involving the first three data points, and so on. Even though these coefficients can be readily formed using the cited pattern, they do tend to get very complicated beyond the first few. To remedy this, we introduce Newton's divided differences for easier handling of the coefficients.

The *first divided difference* at x_i is denoted by $f[x_{i+1}, x_i]$ and defined as the slope of the line connecting (x_i, y_i) and (x_{i+1}, y_{i+1}) , that is,

$$f[x_{i+1}, x_i] = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

The *second divided difference* at x_i is denoted by $f[x_{i+2}, x_{i+1}, x_i]$ and defined as

$$\begin{aligned} f[x_{i+2}, x_{i+1}, x_i] &= \frac{f[x_{i+2}, x_{i+1}] - f[x_{i+1}, x_i]}{x_{i+2} - x_i} \\ &= \frac{(y_{i+2} - y_{i+1})/(x_{i+2} - x_{i+1}) - (y_{i+1} - y_i)/(x_{i+1} - x_i)}{x_{i+2} - x_i} \end{aligned}$$

Note that the first divided differences are directly involved in the construction of the second divided difference. If instead of x_i we focus on x_1 , it is then clear that the first divided difference at x_1 is

$$f[x_2, x_1] = \frac{y_2 - y_1}{x_2 - x_1} \stackrel{\text{As shown earlier}}{=} a_2$$

The second divided difference at x_1 is

$$f[x_3, x_2, x_1] = \frac{f[x_3, x_2] - f[x_2, x_1]}{x_3 - x_1} = \frac{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1} \stackrel{\text{As shown earlier}}{=} a_3$$

In general, the k th divided difference at x_1 is described by

$$f[x_{k+1}, x_k, \dots, x_2, x_1] = \frac{f[x_{k+1}, x_k, \dots, x_3, x_2] - f[x_k, x_{k-1}, \dots, x_2, x_1]}{x_{k+1} - x_1} = a_{k+1} \quad (5.21)$$

Ultimately, the n th-degree Newton divided-difference interpolating polynomial for the data $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$ is formed as

$$p_n(x) = \boxed{a_1}_{y_1} + \boxed{a_2}_{f[x_2, x_1]}(x - x_1) + \boxed{a_3}_{f[x_3, x_2, x_1]}(x - x_1)(x - x_2) + \dots + \boxed{a_{n+1}}_{f[x_{n+1}, \dots, x_1]}(x - x_1)(x - x_2) \dots (x - x_n) \quad (5.22)$$

where the coefficients a_1, \dots, a_{n+1} are best calculated with the aid of a *divided-differences table*. A typical such table is presented in [Table 5.7](#) corresponding to a set of five data points.

TABLE 5.7

Divided Differences Table

x_i	y_i	First Divided Diff	Second Divided Diff	Third Divided Diff	Fourth Divided Diff
x_1	$y_1 = \boxed{a_1}$				
		$f[x_2, x_1] = \boxed{a_2}$			
x_2	y_2		$f[x_3, x_2, x_1] = \boxed{a_3}$		
		$f[x_3, x_2]$		$f[x_4, x_3, x_2, x_1] = \boxed{a_4}$	
x_3	y_3		$f[x_4, x_3, x_2]$		$f[x_5, x_4, x_3, x_2, x_1] = \boxed{a_5}$
		$f[x_4, x_3]$		$f[x_5, x_4, x_3, x_2]$	
x_4	y_4		$f[x_5, x_4, x_3]$		
		$f[x_5, x_4]$			
x_5	y_5				

The user-defined function `NewtonInterp` finds the Newton divided-difference interpolating polynomial that fits a set of data and uses this polynomial to calculate the interpolated value at a specific point.

```
function yi = NewtonInterp(x,y,xi)
%
% NewtonInterp finds the Newton divided-difference interpolating
% polynomial that agrees with the data (x,y) and uses it to find the
% interpolated value at xi.
%
% yi = NewtonInterp(x,y,xi), where
%
% x, y are n-dimensional row or column vectors of data,
% xi is a specified point,
%
% yi is the interpolated value at xi.
%
n = length(x);
a = zeros(1,n); % Pre-allocate
a(1) = y(1);
DivDiff = zeros(1,n-1); % Pre-allocate
for i = 1:n-1,
    DivDiff(i,1) = (y(i+1) - y(i))/(x(i+1) - x(i));
end
for j = 2:n-1,
    for i = 1:n-j,
        DivDiff(i,j) = (DivDiff(i+1,j-1) - DivDiff(i,j-1))/(x(j+i) - x(i));
    end
end
for k = 2:n,
    a(k) = DivDiff(1,k-1);
end
yi = a(1);
xprod = 1;
for m = 2:n,
    xprod = xprod*(xi - x(m-1));
    yi = yi + a(m)*xprod;
end
```

EXAMPLE 5.8: NEWTON INTERPOLATION, DIVIDED DIFFERENCES

Consider the data in [Table 5.8](#).

1. Use the second-degree Newton interpolating polynomial that goes through the second, third, and fourth data points to find the interpolated value at $x = 0.35$. Confirm the result by executing the user-defined function `NewtonInterp`.
2. Use the fourth-degree Newton interpolating polynomial that goes through all five data points to find the interpolated value at $x = 0.35$. Confirm the result by executing `NewtonInterp`.

TABLE 5.8

Data in Example 5.8

x	y
0	0
0.1	0.1210
0.2	0.2258
0.5	0.4650
0.8	0.6249

Solution

1. If a second-degree polynomial is to be used for interpolation at $x = 0.35$, then the one going through the second, third, and fourth data points is the most suitable due to the location of $x = 0.35$ relative to the data. The corresponding divided differences are calculated according to Equation 5.21 and recorded in Table 5.9. The second-degree interpolating polynomial is then formed via Equation 5.22, as

$$\begin{aligned}
 p_2(x) &= a_1 + a_2(x - x_1) + a_3(x - x_1)(x - x_2) \\
 &= 0.1210 + 1.0480(x - 0.1) - 0.6268(x - 0.1)(x - 0.2)
 \end{aligned}$$

Substitution of $x = 0.35$ yields $p_2(0.35) = 0.3595$. The result is confirmed in MATLAB as follows.

```
>> x = [0.1 0.2 0.5]; y = [0.1210 0.2258 0.4650];
>> yi = NewtonInterp(x,y,0.35)
```

```
yi =
    0.3595
```

TABLE 5.9

Divided Differences Table for Example 5.8, Part (1)

x_i	y_i	1st Divided Diff	2nd Divided Diff
0.1	$\boxed{0.1210}$		
		$\frac{0.2258 - 0.1210}{0.2 - 0.1}$	
		$= \boxed{1.0480}$	
0.2	0.2258		$\frac{0.7973 - 1.0480}{0.5 - 0.1}$
			$= \boxed{-0.6268}$
		$\frac{0.4650 - 0.2258}{0.5 - 0.2} = 0.7973$	
0.5	0.4650		

TABLE 5.10

Divided Differences Table for Example 5.8, Part (2)

x_i	y_i	1st Divided Diff	2nd Divided Diff	3rd Divided Diff	4th Divided Diff
0	$\boxed{0}$				
		$\frac{0.1210 - 0}{0.1 - 0}$ $= \boxed{1.2100}$			
0.1	0.1210		$\frac{1.0480 - 1.2100}{0.2 - 0}$ $= \boxed{-0.8100}$		
		$\frac{0.2258 - 0.1210}{0.2 - 0.1} = 1.0480$		$\frac{-0.6268 - (-0.8100)}{0.5 - 0}$ $= \boxed{0.3664}$	
0.2	0.2258		$\frac{0.7973 - 1.0480}{0.5 - 0.1} = -0.6268$		$\frac{0.2661 - 0.3664}{0.8 - 0}$ $= \boxed{-0.1254}$
		$\frac{0.4650 - 0.2258}{0.5 - 0.2} = 0.7973$		$\frac{-0.4405 - (-0.6268)}{0.8 - 0.1} = 0.2661$	
0.5	0.4650		$\frac{0.5330 - 0.7973}{0.8 - 0.2} = -0.4405$		
		$\frac{0.6249 - 0.4650}{0.8 - 0.5} = 0.5330$			
0.8	0.6249				

2. The divided differences are calculated according to Equation 5.21 and recorded in Table 5.10. The fourth-degree interpolating polynomial is then formed via Equation 5.22, as

$$\begin{aligned}
 p_4(x) &= a_1 + a_2(x - x_1) + a_3(x - x_1)(x - x_2) + a_4(x - x_1)(x - x_2)(x - x_3) \\
 &\quad + a_5(x - x_1)(x - x_2)(x - x_3)(x - x_4) = 0 + 1.2100(x - 0) - 0.8100(x - 0)(x - 0.1) \\
 &\quad + 0.3664(x - 0)(x - 0.1)(x - 0.2) - 0.1254(x - 0)(x - 0.1)(x - 0.2)(x - 0.5)
 \end{aligned}$$

Substitution of $x = 0.35$ yields $p_4(0.35) = 0.3577$. The result is confirmed in MATLAB as follows.

```
>> x = [0 0.1 0.2 0.5 0.8]; y = [0 0.1210 0.2258 0.4650 0.6249];
>> yi = NewtonInterp(x,y,0.35)
```

```
yi =
    0.3577
```

This interpolated value is considered a better estimate at $x = 0.35$ than that obtained in (1) because the data was utilized more effectively.

5.5.4 Special Case: Equally-Spaced Data

In the derivation of Newton divided difference interpolating polynomial, Equation 5.22, no restriction was placed on how the data was spaced. In the event that the data is equally spaced, as it is often the case in practice, the divided differences reduce to simpler forms. Let every two successive x values in the data $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$ be separated by distance h so that

$$x_{i+1} - x_i = h, \quad i = 1, 2, \dots, n$$

Consequently,

$$x_2 = x_1 + h, \quad x_3 = x_1 + 2h, \dots, \quad x_{n+1} = x_1 + nh$$

The *first forward difference* at x_i is denoted by Δy_i and defined as

$$\Delta y_i = y_{i+1} - y_i, \quad i = 1, 2, \dots, n$$

The *second forward difference* at x_i is denoted by $\Delta^2 y_i$ and defined as

$$\Delta^2 y_i = \Delta y_{i+1} - \Delta y_i, \quad i = 1, 2, \dots, n$$

Note that the first forward differences are directly involved in the construction of the second forward difference. In general, the k th forward difference at x_i is described as

$$\Delta^k y_i = \Delta^{k-1} y_{i+1} - \Delta^{k-1} y_i \quad (5.23)$$

If instead of x_i we focus on x_1 , then the first forward difference at x_1 is

$$\Delta y_1 = y_2 - y_1$$

The second forward difference at x_1 is

$$\Delta^2 y_1 = \Delta y_2 - \Delta y_1$$

and so on. We next find out how the divided differences and forward differences are related. The first divided difference at x_1 can be written as

$$f[x_2, x_1] = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y_1}{h}$$

The second divided difference at x_1 is

$$f[x_3, x_2, x_1] = \frac{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1} = \frac{\frac{\Delta y_2 - \Delta y_1}{h}}{2h} = \frac{\Delta y_2 - \Delta y_1}{2h^2} = \frac{\Delta^2 y_1}{2h^2}$$

In general, the k th divided difference at x_1 can be expressed as

$$f[x_{k+1}, x_k, \dots, x_2, x_1] = \frac{\Delta^k y_1}{k! h^k}$$

5.5.5 Newton Forward-Difference Interpolating Polynomials

Any arbitrary x between x_1 and x_{n+1} can be expressed as $x = x_1 + mh$ for a suitable real value m . Then

$$x - x_2 = (x_1 + mh) - x_2 = mh - (x_2 - x_1) = mh - h = (m - 1)h$$

$$x - x_3 = (m - 2)h$$

...

$$x - x_n = (m - (n - 1))h$$

Substitution of these, together with the relations between divided differences and forward differences established above, into Equation 5.22, yields the Newton forward-difference interpolating polynomial as

$$\begin{aligned} p_n(x) &= y_1 + \frac{\Delta y_1}{h}(mh) + \frac{\Delta^2 y_1}{2h^2}(mh)((m-1)h) + \dots + \frac{\Delta^n y_1}{n!h^n}(mh)((m-1)h) \dots (m-(n-1))h \\ &= y_1 + m\Delta y_1 + \frac{m(m-1)}{2!}\Delta^2 y_1 + \dots + \frac{m(m-1) \dots (m-(n-1))}{n!}\Delta^n y_1, \quad m = \frac{x - x_1}{h} \end{aligned} \quad (5.24)$$

This polynomial is best formed with the aid of a forward-differences table, as in [Table 5.11](#).

A user-defined function with function call `yi = Newton_FD(x,Y,xi)` can be written (see Problem Set) that finds the Newton forward-difference interpolating polynomial for the equally spaced data (x,Y) and uses this polynomial to interpolate at x_i and returns the interpolated value in y_i . Of course, `NewtonInterp` works for the equally spaced data as well, but is not recommended as it will perform unnecessary calculations.

TABLE 5.11

Forward Differences Table

x_i	y_i	First Forward Diff	Second Forward Diff	Third Forward Diff
x_1	y_1			
		Δy_1		
x_2	y_2		$\Delta^2 y_1$	
		Δy_2		$\Delta^3 y_1$
x_3	y_3		$\Delta^2 y_2$	
		Δy_3		
x_4	y_4			

EXAMPLE 5.9: NEWTON INTERPOLATION, FORWARD DIFFERENCES

For the data in [Table 5.12](#), interpolate at $x = 0.64$ using the fourth-degree Newton forward-difference interpolating polynomial. Confirm the result by executing the user-defined function `NewtonInterp`.

Solution

The forward differences are calculated according to Equation 5.23 and recorded in [Table 5.13](#). Since we are interpolating at $x = 0.64$, we have

$$m = \frac{x - x_1}{h} = \frac{0.64 - 0.4}{0.1} = 2.4$$

The fourth-degree Newton forward-difference interpolating polynomial is given by Equation 5.24, as

$$p_4(x) = y_1 + m\Delta y_1 + \frac{m(m-1)}{2!}\Delta^2 y_1 + \frac{m(m-1)(m-2)}{3!}\Delta^3 y_1 + \frac{m(m-1)(m-2)(m-3)}{4!}\Delta^4 y_1$$

TABLE 5.12

Data in Example 5.9

x	$y = \cos x$
0.4	0.921061
0.5	0.877583
0.6	0.825336
0.7	0.764842
0.8	0.696707

TABLE 5.13

Forward Differences Table for Example 5.9

x_i	y_i	1st Forward Diff	2nd Forward Diff	3rd Forward Diff	4th Forward Diff
0.4	0.921061				
		-0.043478 Δy_1			
0.5	0.877583		-0.008769 $\Delta^2 y_1$		
		-0.052247		0.000522 $\Delta^3 y_1$	
0.6	0.825336		-0.008247		0.000084 $\Delta^4 y_1$
		-0.060494		0.000606	
0.7	0.764842		-0.007641		
		-0.068135			
0.8	0.696707				

Inserting $m = 2.4$ and the proper (boxed) entries of [Table 5.13](#) into this equation, the interpolated value is found as

$$\begin{aligned}
 p_4(0.64) &= 0.921061 + 2.4(-0.043478) + \frac{(2.4)(2.4-1)}{2}(-0.008769) \\
 &\quad + \frac{(2.4)(2.4-1)(2.4-2)}{6}(0.000522) + \frac{(2.4)(2.4-1)(2.4-2)(2.4-3)}{24}(0.000084) \\
 &= 0.8020959856
 \end{aligned}$$

Noting the actual value is $\cos(0.64) = 0.8020957579$, the above interpolation yielded a 6-decimal place accuracy. The result can be verified in MATLAB using the user-defined function `NewtonInterp`.

```

>> format long
>> x = [0.4 0.5 0.6 0.7 0.8];
>> y = [0.921061 0.877583 0.825336 0.764842 0.696707];
>> yi = NewtonInterp(x,y,0.64)

yi =

    0.802095985600000

```

5.6 Spline Interpolation

In Section 5.5, we used n th-degree polynomials to interpolate $n + 1$ data points. For example, we learned that a set of 11 data points can be interpolated by a single polynomial of at most degree 10. When there are a small number of points in the data, the degree of the interpolating polynomial will also be small, and the interpolated values are generally accurate. However, when a high-degree polynomial is used to interpolate a large number of points, large errors in interpolation are possible, as shown in [Figure 5.18](#). The main

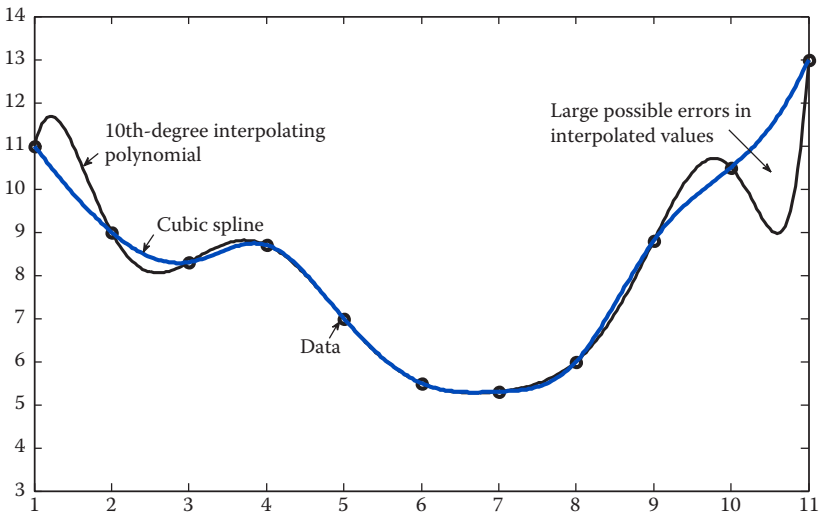


FIGURE 5.18

A 10th-degree interpolating polynomial and cubic splines for a set of 11 data points.

contributing factor is the large number of peaks and valleys that accompany a high-degree polynomial. These situations may be avoided by using several low-degree polynomials, each of which is valid in one interval between one or more data points. The low degree of each polynomial in turn limits the number of peaks and valleys to a low number, hence reducing the possibility of large deviations from the main theme of the data. These low-degree polynomials are known as splines. The term “spline” originated from a thin, flexible strip, known as a spline, used by draftsmen to draw smooth curves over a set of points marked by pegs or nails. The data points at which two splines meet are called knots.

The most commonly used splines are cubic splines, which produce very smooth connections over adjacent intervals. Figure 5.18 shows the clear advantage of using several cubic splines as opposed to one single high-degree polynomial for interpolation of a large set of data.

5.6.1 Linear Splines

With linear splines, straight lines (linear functions) are used for interpolation between the data points. Figure 5.19 shows the linear splines used for a set of four data points, as well as the corresponding third-degree interpolating polynomial. If the data points are labeled (x_1, y_1) , (x_2, y_2) , (x_3, y_3) , and (x_4, y_4) , then, using the Lagrange form, the linear splines are simply three linear functions described by

$$S_1(x) = \frac{x - x_2}{x_1 - x_2} y_1 + \frac{x - x_1}{x_2 - x_1} y_2, \quad x_1 \leq x \leq x_2$$

$$S_2(x) = \frac{x - x_3}{x_2 - x_3} y_2 + \frac{x - x_2}{x_3 - x_2} y_3, \quad x_2 \leq x \leq x_3$$

$$S_3(x) = \frac{x - x_4}{x_3 - x_4} y_3 + \frac{x - x_3}{x_4 - x_3} y_4, \quad x_3 \leq x \leq x_4$$

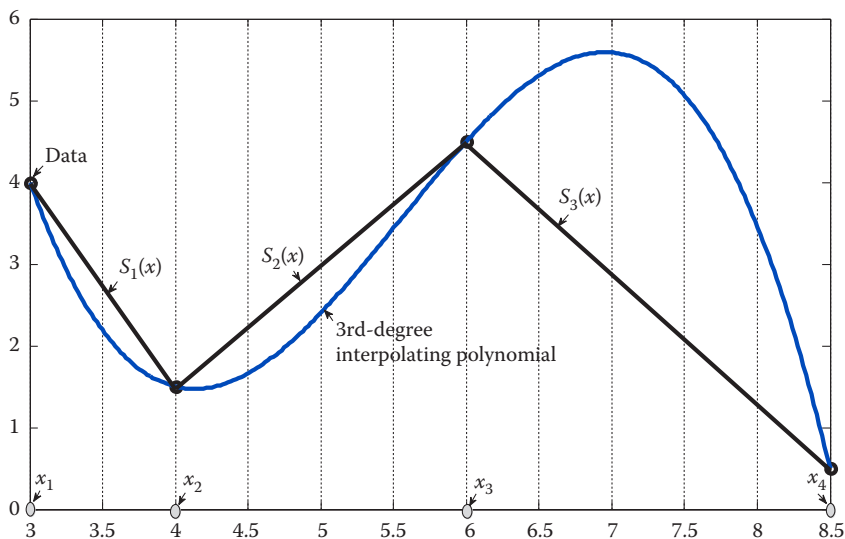


FIGURE 5.19
Linear splines.

This is clearly the same as linear interpolation as discussed in Section 5.5. The obvious drawback of linear splines is that they are not smooth so that the slope experiences sudden changes at the knots. This is because the first derivatives of neighboring linear functions do not agree. To circumvent this problem, higher-degree polynomial splines are used such that the derivatives of every two successive splines agree at the point (knot) they meet. Quadratic splines ensure continuous first derivatives at the knots, but not the second derivatives. Cubic splines ensure continuity of both first and second derivatives at the knots, and are most commonly used in practice.

5.6.2 Quadratic Splines

With quadratic splines, a second-degree polynomial is employed to interpolate over each interval between data points. Suppose there are $n + 1$ data points $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$ so that there are n intervals and thus n quadratic polynomials; see Figure 5.20. Each quadratic polynomial is in the form

$$S_i(x) = a_i x^2 + b_i x + c_i, \quad i = 1, 2, \dots, n \quad (5.25)$$

where a_i, b_i, c_i ($i = 1, 2, \dots, n$) are unknown constants to be determined. Since there are n such polynomials, and each has three unknown constants, there are a total of $3n$ unknown constants. Therefore, exactly $3n$ equations are needed to determine all the unknowns. These equations are generated as follows:

5.6.2.1 Function Values at the Endpoints (2 Equations)

The first polynomial $S_1(x)$ must go through (x_1, y_1) and the last polynomial $S_n(x)$ must go through (x_{n+1}, y_{n+1}) :

$$\begin{aligned} S_1(x_1) &= y_1 \\ S_n(x_{n+1}) &= y_{n+1} \end{aligned}$$

More specifically,

$$\begin{aligned} a_1 x_1^2 + b_1 x_1 + c_1 &= y_1 \\ a_n x_{n+1}^2 + b_n x_{n+1} + c_n &= y_{n+1} \end{aligned} \quad (5.26)$$

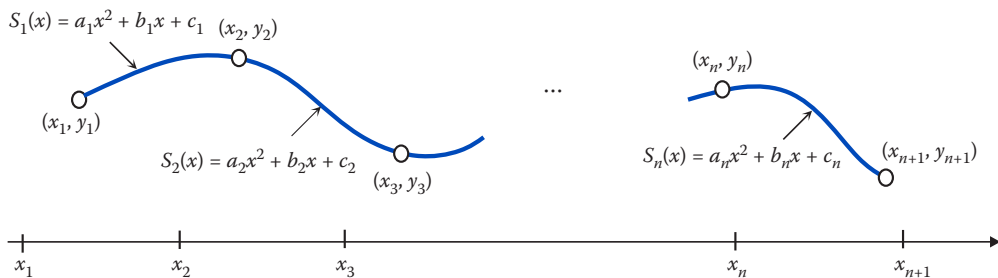


FIGURE 5.20
Quadratic splines.

5.6.2.2 Function Values at the Interior Knots ($2n - 2$ Equations)

At the interior knots, two conditions must hold: (1) polynomials must go through the data points, and (2) adjacent polynomials must agree at the data points:

$$\begin{aligned} S_1(x_2) &= y_2, S_2(x_3) = y_3, \dots, S_{n-1}(x_n) = y_n \\ S_2(x_2) &= y_2, S_3(x_3) = y_3, \dots, S_n(x_n) = y_n \end{aligned}$$

More specifically,

$$\begin{aligned} S_i(x_{i+1}) &= y_{i+1}, \quad i = 1, 2, \dots, n-1 \\ S_i(x_i) &= y_i, \quad i = 2, 3, \dots, n \end{aligned}$$

so that

$$\begin{aligned} a_i x_{i+1}^2 + b_i x_{i+1} + c_i &= y_{i+1}, \quad i = 1, 2, \dots, n-1 \\ a_i x_i^2 + b_i x_i + c_i &= y_i, \quad i = 2, 3, \dots, n \end{aligned} \tag{5.27}$$

Note that each of the above contains $n - 1$ equations so that $2n - 2$ equations are generated in this stage.

5.6.2.3 First Derivatives at the Interior Knots ($n - 1$ Equations)

At the interior knots, the first derivatives of adjacent quadratic polynomials must agree:

$$S'_1(x_2) = S'_2(x_2), S'_2(x_3) = S'_3(x_3), \dots, S'_{n-1}(x_n) = S'_n(x_n)$$

More specifically,

$$S'_i(x_{i+1}) = S'_{i+1}(x_{i+1}), \quad i = 1, 2, \dots, n-1$$

Noting that $S'_i(x) = 2a_i x + b_i$, the above yields

$$2a_i x_{i+1} + b_i = 2a_{i+1} x_{i+1} + b_{i+1}, \quad i = 1, 2, \dots, n-1 \tag{5.28}$$

This contains $n - 1$ equations. So far, we have managed to generate $\boxed{2} + \boxed{2n-2} + \boxed{n-1} = 3n - 1$ equations. One more equation is needed to complete the task. Among several available choices, we select the following:

5.6.2.4 Second Derivative at the Left Endpoint is Zero (1 Equation)

$$S''_1(x_1) = 0$$

Noting that $S''_1(x_1) = 2a_1$, this yields

$$a_1 = 0 \tag{5.29}$$

TABLE 5.14

Data in Example 5.10

x	y
2	5
3	2.3
5	5.1
7.5	1.5

A total of $3n$ equations have therefore been generated.

In summary, one equation simply gives $a_1 = 0$ and the remaining $3n - 1$ unknowns are found by solving the $3n - 1$ equations provided by Equations 5.26 through 5.28.

EXAMPLE 5.10: QUADRATIC SPLINES

Construct the quadratic splines for the data in [Table 5.14](#).

Solution

Since there are four data points, we have $n = 3$ so that there are three quadratic splines with a total of nine unknown constants. Of these, one is given by $a_1 = 0$. The remaining eight equations to solve are provided by Equations 5.26 through 5.28 as follows.

Equation 5.26 yields

$$\begin{aligned} a_1(2)^2 + b_1(2) + c_1 &= 5 \\ a_3(7.5)^2 + b_3(7.5) + c_3 &= 1.5 \end{aligned}$$

Equation 5.27 yields

$$\begin{aligned} a_1(3)^2 + b_1(3) + c_1 &= 2.3 \\ a_2(5)^2 + b_2(5) + c_2 &= 5.1 \\ a_2(3)^2 + b_2(3) + c_2 &= 2.3 \\ a_3(5)^2 + b_3(5) + c_3 &= 5.1 \end{aligned}$$

Finally, Equation 5.28 gives

$$\begin{aligned} 2a_1(3) + b_1 &= 2a_2(3) + b_2 \\ 2a_2(5) + b_2 &= 2a_3(5) + b_3 \end{aligned}$$

Substituting $a_1 = 0$ and writing the above equations in matrix form, we arrive at

$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 56.25 & 7.5 & 1 \\ 3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 25 & 5 & 1 & 0 & 0 & 0 \\ 0 & 0 & 9 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 25 & 5 & 1 \\ 1 & 0 & -6 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 1 & 0 & -10 & -1 & 0 \end{bmatrix} \begin{Bmatrix} b_1 \\ c_1 \\ a_2 \\ b_2 \\ c_2 \\ a_3 \\ b_3 \\ c_3 \end{Bmatrix} = \begin{Bmatrix} 5 \\ 1.5 \\ 2.3 \\ 5.1 \\ 2.3 \\ 5.1 \\ 0 \\ 0 \end{Bmatrix}$$

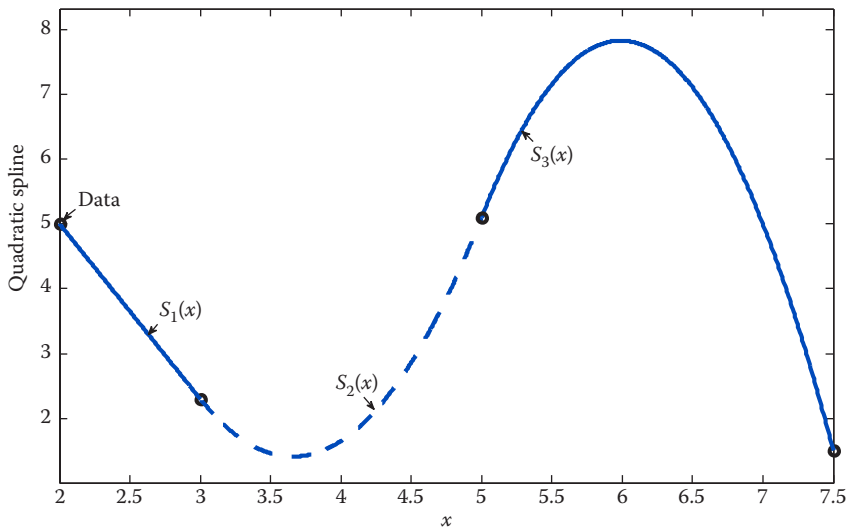


FIGURE 5.21
Quadratic splines in Example 5.10.

This system is subsequently solved to obtain

$$\begin{array}{lll} \boxed{a_1 = 0} & a_2 = 2.05 & a_3 = -2.776 \\ b_1 = -2.7 & b_2 = -15 & b_3 = 33.26 \\ c_1 = 10.4 & c_2 = 28.85 & c_3 = -91.8 \end{array}$$

Therefore, the quadratic splines are completely defined by the following three second-degree polynomials:

$$\begin{aligned} S_1(x) &= -2.7x + 10.4, & 2 \leq x \leq 3 \\ S_2(x) &= 2.05x^2 - 15x + 28.85, & 3 \leq x \leq 5 \\ S_3(x) &= -2.776x^2 + 33.26x - 91.8, & 5 \leq x \leq 7.5 \end{aligned}$$

Results are shown graphically in [Figure 5.21](#). Note that the first spline, $S_1(x)$, describes a straight line since $a_1 = 0$.

5.6.3 Cubic Splines

In cubic splines, third-degree polynomials are used to interpolate over each interval between data points. Suppose there are $n + 1$ data points $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$ so that there are n intervals and thus n cubic polynomials. Each cubic polynomial is conveniently expressed in the form

$$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i, \quad i = 1, 2, \dots, n \quad (5.30)$$

where a_i, b_i, c_i, d_i ($i = 1, 2, \dots, n$) are unknown constants to be determined. Since there are n such polynomials, and each has four unknown constants, there are a total of $4n$ unknown constants. Therefore, $4n$ equations are needed to determine all the unknowns. These

equations are derived based on the same logic as quadratic splines, except that second derivatives of adjacent splines also agree at the interior knots and two boundary conditions are required.

Splines go through the endpoints and interior knots, and adjacent splines agree at the interior knots (2n equations)

$$\begin{aligned} S_1(x_1) &= y_1, & S_n(x_{n+1}) &= y_{n+1} \\ S_{i+1}(x_{i+1}) &= S_i(x_{i+1}), & i &= 1, 2, \dots, n-1 \\ S_i(x_i) &= y_i, & i &= 2, 3, \dots, n \end{aligned} \tag{5.31}$$

First derivatives of adjacent splines agree at the common interior knots (n - 1 equations)

$$S'_i(x_{i+1}) = S'_{i+1}(x_{i+1}), \quad i = 1, 2, \dots, n-1 \tag{5.32}$$

Second derivatives of adjacent splines agree at the common interior knots (n - 1 equations)

$$S''_i(x_{i+1}) = S''_{i+1}(x_{i+1}), \quad i = 1, 2, \dots, n-1 \tag{5.33}$$

A total of $\boxed{2n} + \boxed{n-1} + \boxed{n-1} = 4n - 2$ equations have been generated up to this point. The other two are provided by the boundary conditions. *Boundary conditions indicate the manner in which the first spline departs from the first data point and the last spline arrives at the last data point.* There are two sets of boundary conditions that are generally used for this purpose.

5.6.3.1 Clamped Boundary Conditions

The slopes with which S_1 departs from (x_1, y_1) and S_n arrives at (x_{n+1}, y_{n+1}) are specified:

$$S'_1(x_1) = p, \quad S'_n(x_{n+1}) = q \tag{5.34}$$

5.6.3.2 Free Boundary Conditions

$$S''_1(x_1) = 0, \quad S''_n(x_{n+1}) = 0 \tag{5.35}$$

The clamped boundary conditions generally yield more accurate approximations because they contain more specific information about the splines; see Example 5.12.

5.6.4 Construction of Cubic Splines: Clamped Boundary Conditions

The coefficients a_i, b_i, c_i, d_i ($i = 1, 2, \dots, n$) will be determined by Equations 5.31 through 5.33, together with clamped boundary conditions given by Equation 5.34.

By Equation 5.30 $S_i(x_i) = d_i$ ($i = 1, 2, \dots, n$). The first and last equations in Equation 5.31 yield $S_i(x_i) = y_i$ ($i = 1, 2, \dots, n$). Therefore,

$$d_i = y_i, \quad i = 1, 2, \dots, n \tag{5.36}$$

Let $h_i = x_{i+1} - x_i$ ($i = 1, 2, \dots, n$) define the spacing between the data points. Using this in the second equation in Equation 5.31, while noting $S_{i+1}(x_{i+1}) = d_{i+1}$, we have

$$d_{i+1} = a_i h_i^3 + b_i h_i^2 + c_i h_i + d_i, \quad i = 1, 2, \dots, n-1$$

If we define $d_{n+1} = y_{n+1}$, then the above equation will be valid for the range $i = 1, 2, \dots, n$ since $S_n(x_{n+1}) = y_{n+1}$. Thus,

$$d_{i+1} = a_i h_i^3 + b_i h_i^2 + c_i h_i + d_i, \quad i = 1, 2, \dots, n \quad (5.37)$$

Taking the first derivative of $S_i(x)$ and applying Equation 5.32, we find

$$c_{i+1} = 3a_i h_i^2 + 2b_i h_i + c_i, \quad i = 1, 2, \dots, n-1$$

If we define $c_{n+1} = S'_n(x_{n+1})$, then the above equation will be valid for the range $i = 1, 2, \dots, n$. Therefore,

$$c_{i+1} = 3a_i h_i^2 + 2b_i h_i + c_i, \quad i = 1, 2, \dots, n \quad (5.38)$$

Taking the second derivative of $S_i(x)$ and applying Equation 5.33, yields

$$2b_{i+1} = 6a_i h_i + 2b_i, \quad i = 1, 2, \dots, n-1$$

If we define $b_{n+1} = \frac{1}{2} S''_n(x_{n+1})$, then the above equation will be valid for the range $i = 1, 2, \dots, n$. Therefore,

$$b_{i+1} = 3a_i h_i + b_i, \quad i = 1, 2, \dots, n \quad (5.39)$$

The goal is to derive a system of equations for b_i ($i = 1, 2, \dots, n+1$) only. Solve Equation 5.39 for $a_i = (b_{i+1} - b_i)/3h_i$ and substitute into Equations 5.37 and 5.38 to obtain

$$d_{i+1} = \frac{1}{3}(2b_i + b_{i+1})h_i^2 + c_i h_i + d_i, \quad i = 1, 2, \dots, n \quad (5.40)$$

and

$$c_{i+1} = (b_i + b_{i+1})h_i + c_i, \quad i = 1, 2, \dots, n \quad (5.41)$$

Solve Equation 5.40 for c_i :

$$c_i = \frac{d_{i+1} - d_i}{h_i} - \frac{1}{3}(2b_i + b_{i+1})h_i \quad (5.42)$$

Change i to $i-1$ and rewrite the above as

$$c_{i-1} = \frac{d_i - d_{i-1}}{h_{i-1}} - \frac{1}{3}(2b_{i-1} + b_i)h_{i-1} \quad (5.43)$$

Also change i to $i-1$ and rewrite Equation 5.41 as

$$c_i = (b_{i-1} + b_i)h_{i-1} + c_{i-1} \quad (5.44)$$

Finally, insert Equations 5.42 and 5.43 into Equation 5.44 to derive

$$b_{i-1}h_{i-1} + 2b_i(h_i + h_{i-1}) + b_{i+1}h_i = \frac{3(d_{i+1} - d_i)}{h_i} - \frac{3(d_i - d_{i-1})}{h_{i-1}}, \quad i = 2, 3, \dots, n \quad (5.45)$$

This describes a system whose only unknowns are b_i ($i = 1, 2, \dots, n + 1$) because d_i ($i = 1, 2, \dots, n + 1$) are simply the values at the data points and h_i ($i = 1, 2, \dots, n$) define the spacing of the data. Equation 5.45, however, generates a system of $n - 1$ equations in $n + 1$ unknowns, which means two more equations are still needed. These come from the clamped boundary conditions, Equation 5.34. First, Equation 5.42 with $i = 1$ gives

$$c_1 = \frac{d_2 - d_1}{h_1} - \frac{1}{3}(2b_1 + b_2)h_1$$

But $c_1 = S'_1(x_1) = p$. Then, the above equation can be rewritten as

$$(2b_1 + b_2)h_1 = \frac{3(d_2 - d_1)}{h_1} - 3p \quad (5.46)$$

By Equation 5.41,

$$c_{n+1} = (b_n + b_{n+1})h_n + c_n$$

Knowing that $c_{n+1} = S'_n(x_{n+1}) = q$, we have

$$c_n = q - (b_n + b_{n+1})h_n \quad (5.47)$$

Equation 5.42 with $i = n$ gives

$$c_n = \frac{d_{n+1} - d_n}{h_n} - \frac{1}{3}(2b_n + b_{n+1})h_n$$

Substituting Equation 5.47 into the above, we have

$$(2b_{n+1} + b_n)h_n = -\frac{3(d_{n+1} - d_n)}{h_n} + 3q \quad (5.48)$$

Combining Equations 5.45, 5.46, and 5.48 yields a system of $n + 1$ equations in $n + 1$ unknowns b_i ($i = 1, 2, \dots, n + 1$).

In summary, the coefficients in Equation 5.30 are determined as follows: First,

$$d_i = y_i, \quad i = 1, 2, \dots, n$$

Next, b_i 's are obtained by solving the system

$$\begin{aligned} (2b_1 + b_2)h_1 &= \frac{3(d_2 - d_1)}{h_1} - 3p \\ b_{i-1}h_{i-1} + 2b_i(h_i + h_{i-1}) + b_{i+1}h_i &= \frac{3(d_{i+1} - d_i)}{h_i} - \frac{3(d_i - d_{i-1})}{h_{i-1}}, \quad i = 2, 3, \dots, n \\ (2b_{n+1} + b_n)h_n &= -\frac{3(d_{n+1} - d_n)}{h_n} + 3q \end{aligned} \quad (5.49)$$

This system contains a total of $n + 1$ equations: the first and last are two single equations, while the one in the middle generates $n - 1$ equations. Recall that h_i ($i = 1, 2, \dots, n$) defines the spacing of the data. The system is tridiagonal (see Section 4.3) with a unique solution. Once b_i 's are known, Equation 5.42 is used to find c_i 's:

$$c_i = \frac{d_{i+1} - d_i}{h_i} - \frac{1}{3}(2b_i + b_{i+1})h_i, \quad i = 1, 2, \dots, n \quad (5.50)$$

Finally, Equation 5.39 is used to determine a_i 's:

$$a_i = \frac{b_{i+1} - b_i}{3h_i}, \quad i = 1, 2, \dots, n \quad (5.51)$$

EXAMPLE 5.11: CUBIC SPLINES, CLAMPED BOUNDARY CONDITIONS

For the data in [Table 5.14](#) of Example 5.10 construct the cubic splines with clamped boundary conditions

$$p = -1, \quad q = 1$$

Solution

Since there are four data points, we have $n = 3$ so that there are three cubic polynomials

$$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i, \quad i = 1, 2, 3$$

Following the summarized procedure outlined above, we first find b_i 's by solving the system in Equation 5.49:

$$\begin{aligned} (2b_1 + b_2)h_1 &= \frac{3(d_2 - d_1)}{h_1} - 3p \\ b_1h_1 + 2b_2(h_2 + h_1) + b_3h_2 &= \frac{3(d_3 - d_2)}{h_2} - \frac{3(d_2 - d_1)}{h_1} \\ b_2h_2 + 2b_3(h_3 + h_2) + b_4h_3 &= \frac{3(d_4 - d_3)}{h_3} - \frac{3(d_3 - d_2)}{h_2} \\ (2b_4 + b_3)h_3 &= -\frac{3(d_4 - d_3)}{h_3} + 3q \end{aligned}$$

Note that d_i 's are simply the values at data points, hence

$$d_1 = 5, d_2 = 2.3, d_3 = 5.1, d_4 = 1.5$$

Also $h_1 = 1, h_2 = 2,$ and $h_3 = 2.5.$ Substituting these, together with $p = -1$ and $q = 1,$ the system reduces to

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 6 & 2 & 0 \\ 0 & 2 & 9 & 2.5 \\ 0 & 0 & 2.5 & 5 \end{bmatrix} \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{Bmatrix} = \begin{Bmatrix} -5.1 \\ 12.3 \\ -8.52 \\ 7.32 \end{Bmatrix}$$

which is tridiagonal, as asserted, and its solution is

$$b_1 = -4.3551, b_2 = 3.6103, b_3 = -2.5033, b_4 = 2.7157$$

Next, c_i 's are found by solving Equation 5.50:

$$\begin{aligned} c_1 &= \frac{d_2 - d_1}{h_1} - \frac{1}{3}(2b_1 + b_2)h_1 = -1 \\ c_2 &= \frac{d_3 - d_2}{h_2} - \frac{1}{3}(2b_2 + b_3)h_2 = -1.7449 \\ c_3 &= \frac{d_4 - d_3}{h_3} - \frac{1}{3}(2b_3 + b_4)h_3 = 0.4691 \end{aligned}$$

Finally, a_i 's come from Equation 5.51:

$$\begin{aligned} a_1 &= \frac{b_2 - b_1}{3h_1} = 2.6551 \\ a_2 &= \frac{b_3 - b_2}{3h_2} = -1.0189 \\ a_3 &= \frac{b_4 - b_3}{3h_3} = 0.6959 \end{aligned}$$

Therefore, the three cubic splines are determined as

$$\begin{aligned} S_1(x) &= 2.6551(x-2)^3 - 4.3551(x-2)^2 - (x-2) + 5, \quad 2 \leq x \leq 3 \\ S_2(x) &= -1.0189(x-3)^3 + 3.6103(x-3)^2 - 1.7449(x-3) + 2.3, \quad 3 \leq x \leq 5 \\ S_3(x) &= 0.6959(x-5)^3 - 2.5033(x-5)^2 + 0.4691(x-5) + 5.1, \quad 5 \leq x \leq 7.5 \end{aligned}$$

The results are illustrated in [Figure 5.22](#), where it is clearly seen that cubic splines are much more desirable than the quadratic splines obtained for the same set of data.

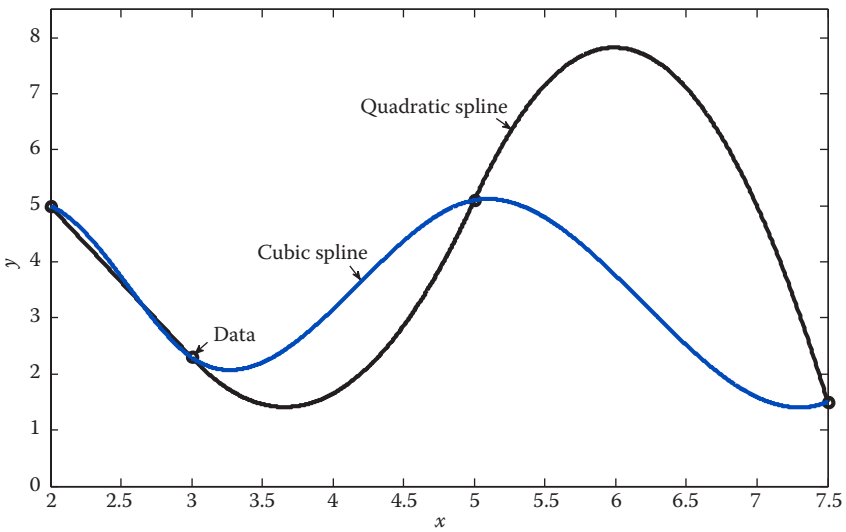


FIGURE 5.22
Cubic and quadratic splines for the same set of data.

5.6.5 Construction of Cubic Splines: Free Boundary Conditions

Recall that free boundary conditions are $S_1''(x_1) = 0$, $S_n''(x_{n+1}) = 0$ so that the first and last data points act as inflection points for the first and last cubic spline, respectively. Knowing $S_i''(x) = 6a_i(x - x_i) + 2b_i$, the first condition yields $b_1 = 0$. From previous work, $b_{n+1} = \frac{1}{2}S_n''(x_{n+1})$ so that the second condition implies $b_{n+1} = 0$. Combining these with Equation 5.45 forms a system of $n + 1$ equations in $n + 1$ unknowns that can be solved for b_i 's:

$$\begin{aligned}
 b_1 &= 0 \\
 b_{i-1}h_{i-1} + 2b_i(h_i + h_{i-1}) + b_{i+1}h_i &= \frac{3(d_{i+1} - d_i)}{h_i} - \frac{3(d_i - d_{i-1})}{h_{i-1}}, \quad i = 2, 3, \dots, n \\
 b_{n+1} &= 0
 \end{aligned} \tag{5.52}$$

Once b_i 's are available, all other unknown constants are determined as in the case of clamped boundary conditions. In summary, d_i ($i = 1, 2, \dots, n + 1$) are the values at the data points, h_i ($i = 1, 2, \dots, n$) define the spacing of the data, b_i 's come from Equation 5.52, c_i 's from Equation 5.50, and a_i 's from Equation 5.51.

EXAMPLE 5.12: CUBIC SPLINES, FREE BOUNDARY CONDITIONS

For the data in Table 5.14 of Examples 5.10 and 5.11 construct the cubic splines with free boundary conditions.

Solution

The free boundary conditions imply $b_1 = 0$, $b_4 = 0$. Consequently, the system in Equation 5.52 simplifies to

$$\begin{aligned}
 b_1 &= 0 \\
 6b_2 + 2b_3 &= 12.3 & b_2 &= 2.5548 \\
 2b_2 + 9b_3 &= -8.52 & \Rightarrow & b_3 &= -1.5144 \\
 b_4 &= 0
 \end{aligned}$$

Next, c_i 's are found by solving Equation 5.50:

$$\begin{aligned}
 c_1 &= \frac{d_2 - d_1}{h_1} - \frac{1}{3}(2b_1 + b_2)h_1 = -3.5516 \\
 c_2 &= \frac{d_3 - d_2}{h_2} - \frac{1}{3}(2b_2 + b_3)h_2 = -0.9968 \\
 c_3 &= \frac{d_4 - d_3}{h_3} - \frac{1}{3}(2b_3 + b_4)h_3 = 1.0840
 \end{aligned}$$

Finally, a_i 's come from Equation 5.51:

$$\begin{aligned}
 a_1 &= \frac{b_2 - b_1}{3h_1} = 0.8516 \\
 a_2 &= \frac{b_3 - b_2}{3h_2} = -0.6782 \\
 a_3 &= \frac{b_4 - b_3}{3h_3} = 0.2019
 \end{aligned}$$

Therefore, the three cubic splines are determined as

$$\begin{aligned}
 S_1(x) &= 0.8516(x-2)^3 - 3.5516(x-2) + 5, & 2 \leq x \leq 3 \\
 S_2(x) &= -0.6782(x-3)^3 + 2.5548(x-3)^2 - 0.9968(x-3) + 2.3, & 3 \leq x \leq 5 \\
 S_3(x) &= 0.2019(x-5)^3 - 1.5144(x-5)^2 + 1.0840(x-5) + 5.1, & 5 \leq x \leq 7.5
 \end{aligned}$$

The graphical results are shown in [Figure 5.23](#), where it is observed that the clamped boundary conditions lead to more accurate approximations, as stated earlier.

5.6.6 MATLAB Built-In Functions `interp1` and `spline`

Brief descriptions of the MATLAB built-in functions `interp1` and `spline` are given as:

`YI = INTERP1(X,Y,XI,METHOD)` specifies alternate methods.

The default is linear interpolation. Use an empty matrix `[]` to specify the default. Available methods are:

```

'nearest'-nearest neighbor interpolation
'linear'-linear interpolation
'spline'-piecewise cubic spline interpolation (SPLINE)
'pchip'-shape-preserving piecewise cubic Hermite interpolation

```

Of the four methods, the nearest neighbor interpolation is the fastest, and does not generate new points. It only returns values that already exist in the `Y` vector. The linear method is slightly slower than the nearest neighbor method and returns values that approximate a continuous function. Each of the `pchip` and `spline` methods generates a different cubic polynomial between any two data points, and uses these points as two

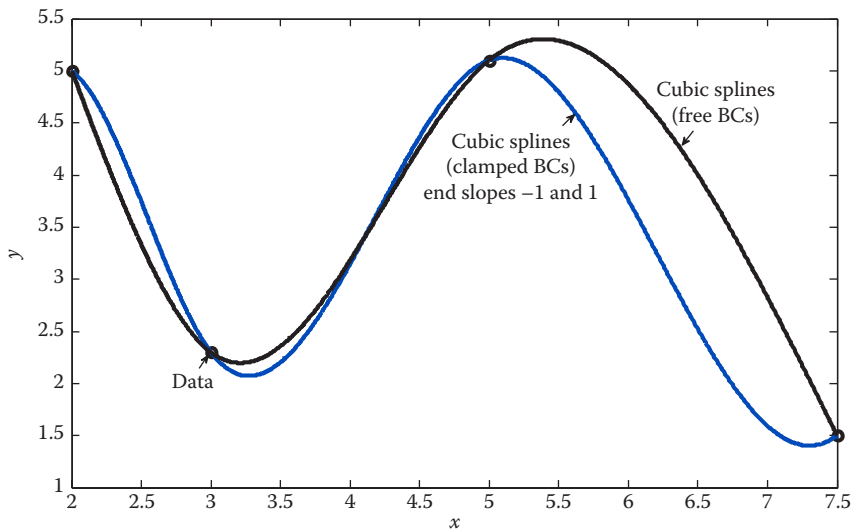


FIGURE 5.23

Cubic splines with clamped and free boundary conditions; Example 5.12.

of the constraints when determining the polynomial. The difference between these two methods is that `pchip` seeks to match the first-order derivatives at these points with those of the intervals before and after, which is a characteristic of Hermite interpolation. The spline method tries to match the second-order derivatives at these points with those of the intervals before and after.

The `pchip` method produces a function whose minimums match the minimums of the data. Also, the function is monotonic over intervals where the data are monotonic. The spline method produces a smooth (twice-continuously differentiable) function, but will overshoot and undershoot the given data.

EXAMPLE 5.13: MATLAB FUNCTION `INTERP1`

Consider the data for $x = -2:0.5:2$ generated by $y = -\frac{1}{4}x^4 + \frac{1}{2}x^2$. Interpolate and plot using the four different methods listed in `interp1`.

Solution

```
>> x = -2:0.5:2;
>> y = -1./4.*x.^4+1./2.*x.^2;
>> xi = linspace(-2, 2);
>> ynear = interp1(x, y, xi, 'nearest');
>> ylin = interp1(x, y, xi, 'linear');
>> ypc = interp1(x, y, xi, 'pchip');
>> yspl = interp1(x, y, xi, 'spline');
% Start Figure 5.24
>> subplot(2,2,1), plot(xi,ynear,x,y,'o'), title('Nearest neighbor
interpolation')
>> hold on
>> subplot(2,2,2), plot(xi,ylin,x,y,'o'), title('Linear interpolation')
>> subplot(2,2,3), plot(xi,ypc,x,y,'o'), title('Piecewise cubic Hermite
interpolation')
>> subplot(2,2,4), plot(xi,yspl,x,y,'o'), title('Cubic spline
interpolation')
```

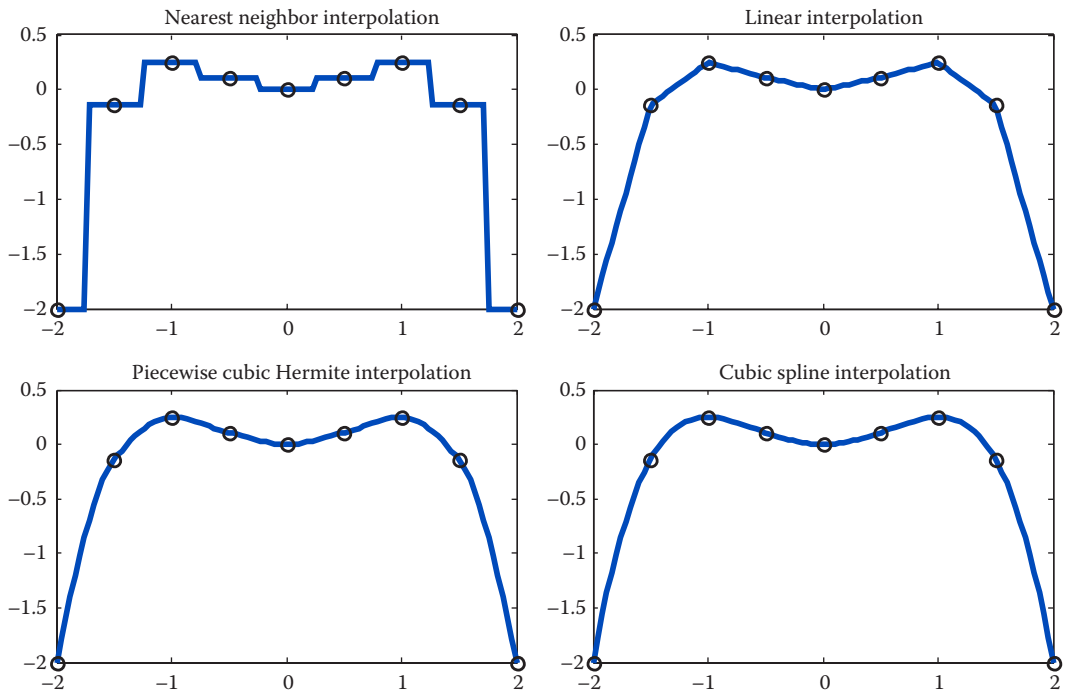


FIGURE 5.24
Interpolation by `interp1` using four methods—Example 5.13.

5.6.7 Boundary Conditions

`YI = INTERP1(X,Y,XI,'spline')` uses piecewise cubic splines interpolation. Note that the option `'spline'` does not allow for specific boundary conditions.

`PP = SPLINE(X,Y)` provides the piecewise polynomial form of the cubic spline interpolant to the data values `Y` at the data sites `X`. Ordinarily, the not-a-knot end conditions* are used. However, if `Y` contains two more values than `X` has entries, then the first and last value in `Y` are used as the end slopes for the cubic spline.

We will apply these two functions to the set of data considered in Examples 5.10 through 5.12.

```
>> x = [2 3 5 7.5]; y = [5 2.3 5.1 1.5];
>> xi = linspace(2,7.5);
>> yi = interp1(x,y,xi,'spline'); % No control over boundary conditions
>> plot(x,y,'o',xi,yi)
>> cs = spline(x,[-1 y 1]); % Specify end slopes of -1 and 1
>> hold on
>> plot(x,y,'o',xi,ppval(cs,xi),'-'); % Figure 5.25
```

* Not-a-knot condition requires that the third derivatives of neighboring splines agree at the second and the one to the last data points. This happens to be the default condition used in MATLAB function `interp1`. “No control over the boundary conditions” refers to this situation.

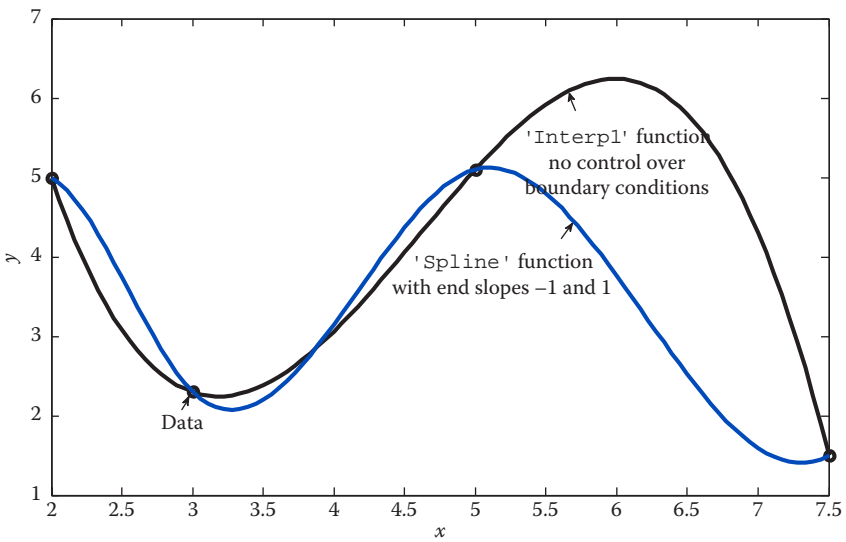


FIGURE 5.25
Cubic splines using MATLAB built-in functions.

5.6.8 Interactive Curve Fitting and Interpolation in MATLAB

The basic fitting interface in MATLAB allows for interactive curve fitting of data. First plot the data. Then, under the “tools” menu choose “basic fitting.” This opens a new window on the side with a list of fitting methods, including spline interpolation and different degree polynomial regression/interpolation. By simply checking the box next to the desired method, the corresponding curve is generated and plotted. [Figure 5.26](#) shows the spline interpolation and 7th-degree polynomial regression of a set of 21 data points.

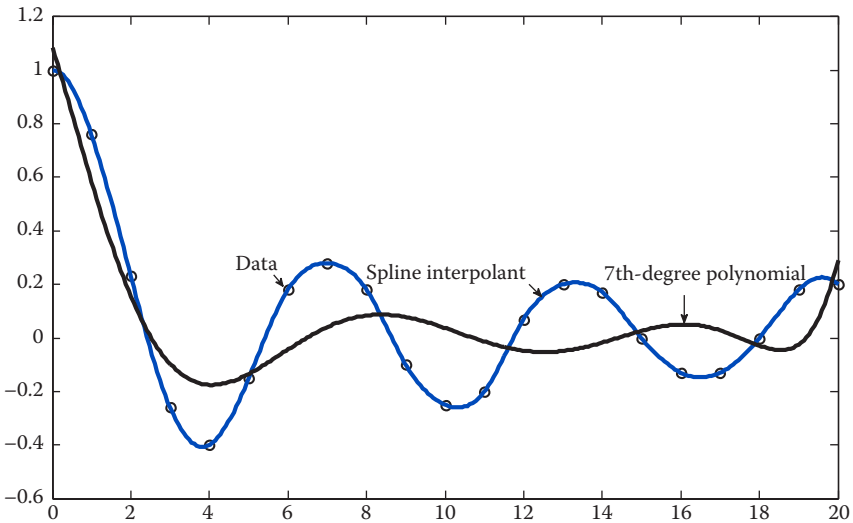


FIGURE 5.26
Basic fitting interface in MATLAB.

5.7 Fourier Approximation and Interpolation

So far in this chapter we have mainly discussed curve fitting and interpolation of data using polynomials. But in many engineering applications we encounter systems that oscillate, and consequently, the collected data exhibits oscillatory behavior. These types of systems are hence modeled via trigonometric functions $1, \cos t, \cos 2t, \dots, \sin t, \sin 2t, \dots$. Fourier approximation/interpolation outlines the systematic use of trigonometric series for this purpose.

5.7.1 Sinusoidal Curve Fitting

To present the idea we first consider a very special set of equally spaced data. Later, we will use a linear transformation to apply the results to any given equally spaced data.

Consider N data points $(\sigma_1, x_1), (\sigma_2, x_2), \dots, (\sigma_N, x_N)$, where σ_k ($k = 1, 2, \dots, N$) are assumed to be equally spaced along the interval $[0, 2\pi)$, that is,

$$\sigma_1 = 0, \sigma_2 = \frac{2\pi}{N}, \sigma_3 = 2\left(\frac{2\pi}{N}\right), \dots, \sigma_N = (N-1)\left(\frac{2\pi}{N}\right)$$

It is desired to interpolate or approximate this set of data by means of a function in the form

$$\begin{aligned} f(\sigma) &= \frac{1}{2}a_0 + \sum_{j=1}^m [a_j \cos j\sigma + b_j \sin j\sigma] \\ &= \frac{1}{2}a_0 + a_1 \cos \sigma + \dots + a_m \cos m\sigma + b_1 \sin \sigma + \dots + b_m \sin m\sigma \end{aligned} \quad (5.53)$$

where $f(\sigma)$ is a trigonometric polynomial of degree m if a_m and b_m are not both zero. Interpolation requires $f(\sigma)$ to go through the data points, while approximation (curve fit) is in the sense of least squares, Section 5.4. More specifically, the coefficients $a_0, a_1, \dots, a_m, b_1, \dots, b_m$ are determined so as to minimize

$$Q = \sum_{k=1}^N \left\{ \left(\frac{1}{2}a_0 + \sum_{j=1}^m [a_j \cos j\sigma_k + b_j \sin j\sigma_k] \right) - x_k \right\}^2 \quad (5.54)$$

A necessary condition for Q to attain a minimum is that its partial derivatives with respect to all coefficients vanish, that is,

$$\frac{\partial Q}{\partial a_0} = 0, \quad \frac{\partial Q}{\partial a_j} = 0 \quad (j = 1, 2, \dots, m), \quad \frac{\partial Q}{\partial b_j} = 0 \quad (j = 1, 2, \dots, m)$$

The ensuing system of $2m + 1$ equations can then be solved to yield

$$\begin{aligned} a_j &= \frac{2}{N} \sum_{k=1}^N x_k \cos j\sigma_k, \quad j = 0, 1, 2, \dots, m \\ b_j &= \frac{2}{N} \sum_{k=1}^N x_k \sin j\sigma_k, \quad j = 1, 2, \dots, m \end{aligned} \quad (5.55)$$

5.7.1.1 Fourier Approximation

If $2m + 1 < N$ (there are more data points than unknown coefficients), then the least-squares approximation of the data is described by Equation 5.53 with coefficients given by Equation 5.55.

5.7.1.2 Fourier Interpolation

For interpolation, the suitable form of the trigonometric polynomial depends on whether N is odd or even.

- **Case 1:** $N = \text{odd} = 2m + 1$

In this case, the interpolating trigonometric polynomial is exactly in the form of the approximating polynomial.

- **Case 2:** $N = \text{even} = 2m$

The interpolating polynomial is in the form

$$\begin{aligned} f(\sigma) &= \frac{1}{2} a_0 + a_1 \cos \sigma + a_2 \cos 2\sigma + \dots + a_{m-1} \cos(m-1)\sigma + \frac{1}{2} a_m \cos m\sigma \\ &\quad + b_1 \sin \sigma + b_2 \sin 2\sigma + \dots + b_{m-1} \sin(m-1)\sigma \end{aligned} \quad (5.56)$$

where the coefficients are once again given by Equation 5.55.

5.7.2 Linear Transformation of Data

Fourier approximation or interpolation of an equally spaced data $(t_1, x_1), (t_2, x_2), \dots, (t_N, x_N)$ is handled as follows. First assume the data is $(\sigma_1, x_1), (\sigma_2, x_2), \dots, (\sigma_N, x_N)$ equally spaced over $[0, 2\pi)$ and apply the results presented above. Then transform the data back to its original form. Such linear transformation is completely described by the connecting line from (σ_1, t_1) to (σ_N, t_N) in the $\sigma - t$ coordinate system; see [Figure 5.27](#). The equation of this connecting line is

$$\sigma = \frac{\sigma_N}{t_N - t_1} (t - t_1)$$

Noting $\sigma_N = (N-1) \left(\frac{2\pi}{N} \right)$, this reduces to

$$\sigma = \frac{(N-1)2\pi}{N(t_N - t_1)} (t - t_1) \quad (5.57)$$

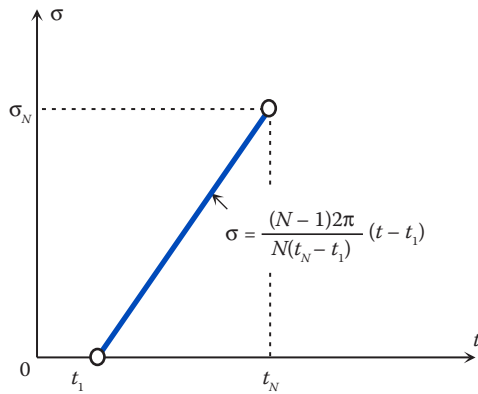


FIGURE 5.27

Linear transformation of data.

The user-defined function `TrigPoly` finds the appropriate Fourier approximation or interpolation of an equally spaced data by first assuming the data is equally spaced over $[0, 2\pi)$ and then transforming the data to agree with the range of the original set. The function also returns the plot of the interpolating/approximating trigonometric polynomial and the given set of data.

```
function [a, b] = TrigPoly(x, m, t1, tN)
%
% TrigPoly approximates or interpolates a set of equally spaced
% data (t1, x1), ..., (tN, xN) by a trigonometric polynomial of degree m.
%
% [a, b] = TrigPoly(x, m, t1, tN), where
%
% x = [x1 x2 ... xN],
% m is the degree of the trigonometric polynomial,
% t1 and tN define the interval endpoints (interval open at tN),
%
% a and b are the vectors of coefficients of the polynomial.
%
% Case(1) Approximation if 2*m + 1 < N,
% Case(2) Interpolation if 2*m + 1 = N or 2*m = N.
%
N = length(x);
% Consider an equally-spaced data from s=0 to s=2*pi
h = 2*pi/N; s = 0:h:2*pi-h; s = s';
a = zeros(m,1); % Pre-allocate
b = a;
for i = 1:m,
    a(i) = x*cos(i*s);
    b(i) = x*sin(i*s);
end
a = 2*a/N; b = 2*b/N; a0 = sum(x)/N;
```

```

if N == 2*m,
    a(m) = a(m)/2;
end
ss = linspace(0,2*pi*(N-1)/N,500); % 500 points for plotting
xx = a0 + a(1)*cos(ss) + b(1)*sin(ss);
for i = 2:m,
    xx = xx + a(i)*cos(i*ss) + b(i)*sin(i*ss);
end
% Transform from s to t
t = N*((tN-t1)/(2*pi*(N-1)))*s + t1;
tt = N*((tN-t1)/(2*pi*(N-1)))*ss + t1;
plot(tt,xx,t,x, 'o')
a = [a0;a];

```

EXAMPLE 5.14: FOURIER APPROXIMATION

Find the first-degree approximating or interpolating trigonometric polynomial for the data in Table 5.15. Confirm the results by executing the user-defined function `TrigPoly`.

Solution

First treat the data as $(\sigma_1, x_1), (\sigma_2, x_2), \dots, (\sigma_5, x_5)$, equally spaced over $[0, 2\pi)$. That is,

$$\sigma_1 = 0, \sigma_2 = \frac{2\pi}{5}, \sigma_3 = \frac{4\pi}{5}, \sigma_4 = \frac{6\pi}{5}, \sigma_5 = \frac{8\pi}{5}$$

Since $m = 1$ and $N = 5$, we have $2m + 1 < N$ so that the polynomial in Equation 5.53 is the suitable form for approximation; in particular, $f(\sigma) = \frac{1}{2}a_0 + a_1 \cos \sigma + b_1 \sin \sigma$. The coefficients are provided by Equation 5.55 as

$$a_0 = \frac{2}{5} \sum_{k=1}^5 x_k = \frac{2}{5}(6.8 + 3.2 - 4.1 - 3.9 + 3.3) = 2.1200$$

$$a_1 = \frac{2}{5} \sum_{k=1}^5 x_k \cos \sigma_k = \frac{2}{5}[6.8 \cos(0) + 3.2 \cos(2\pi/5) - 4.1 \cos(4\pi/5) - 3.9 \cos(6\pi/5) + 3.3 \cos(8\pi/5)] = 6.1123$$

$$b_1 = \frac{2}{5} \sum_{k=1}^5 x_k \sin \sigma_k = \frac{2}{5}[6.8 \sin(0) + 3.2 \sin(2\pi/5) - 4.1 \sin(4\pi/5) - 3.9 \sin(6\pi/5) + 3.3 \sin(8\pi/5)] = -0.0851$$

TABLE 5.15

Data in Example 5.14

t	x
0.5	6.8
0.7	3.2
0.9	-4.1
1.1	-3.9
1.3	3.3

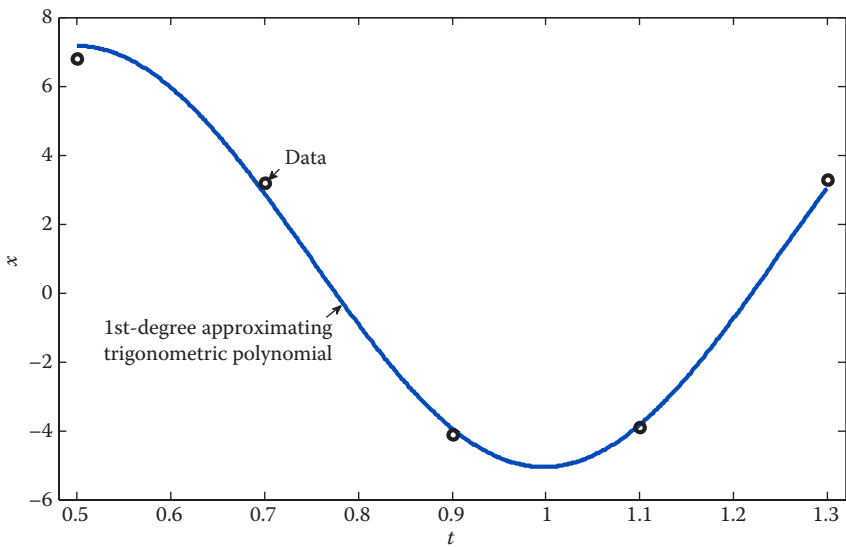


FIGURE 5.28

Fourier approximation of the data in Example 5.14.

Therefore, the least-squares approximating polynomial is $f(\sigma) = 1.06 + 6.1123 \cos \sigma - 0.0851 \sin \sigma$. But variables t and σ are related via Equation 5.57,

$$\sigma = \frac{(5-1)2\pi}{5(1.3-0.5)}(t-0.5) = 6.2832(t-0.5)$$

The approximating trigonometric polynomial is then formed as

$$f(t) = 1.06 + 6.1123 \cos(6.2832(t-0.5)) - 0.0851 \sin(6.2832(t-0.5))$$

Executing the user-defined function `TrigPoly` yields the coefficients of the trigonometric polynomial as well as the plot of this polynomial and the original data. This is shown in [Figure 5.28](#).

```
>> x = [6.8 3.2 -4.1 -3.9 3.3];
>> [a, b] = TrigPoly(x, 1, 0.5, 1.3)

a =
    1.0600
    6.1123

b =
   -0.0851
```

EXAMPLE 5.15: FOURIER INTERPOLATION

Find the third-degree approximating or interpolating trigonometric polynomial for the data in [Table 5.16](#). Confirm the results by executing the user-defined function `TrigPoly`. Find the interpolated value at $t = 0.66$.

TABLE 5.16

Data in Example 5.15

t	x
0.10	0
0.25	0
0.40	0
0.55	1
0.70	1
0.85	1

Solution

First treat the data as $(\sigma_1, x_1), (\sigma_2, x_2), \dots, (\sigma_6, x_6)$, equally spaced over $[0, 2\pi)$. That is,

$$\sigma_1 = 0, \sigma_2 = \frac{\pi}{3}, \sigma_3 = \frac{2\pi}{3}, \sigma_4 = \pi, \sigma_5 = \frac{4\pi}{3}, \sigma_6 = \frac{5\pi}{3}$$

Since $m = 3$ and $N = 6$, we have $2m = N$ so that the trigonometric polynomial interpolates the data and is given by Equation 5.56, more specifically

$$f(\sigma) = \frac{1}{2}a_0 + a_1 \cos \sigma + a_2 \cos 2\sigma + \frac{1}{2}a_3 \cos 3\sigma + b_1 \sin \sigma + b_2 \sin 2\sigma$$

The coefficients are provided by Equation 5.55 as

$$\begin{aligned} a_0 &= \frac{2}{6} \sum_{k=1}^6 x_k = 1, & a_1 &= \frac{2}{6} \sum_{k=1}^6 x_k \cos \sigma_k = -0.3333, & a_2 &= \frac{2}{6} \sum_{k=1}^6 x_k \cos 2\sigma_k = 0 \\ a_3 &= \frac{2}{6} \sum_{k=1}^6 x_k \cos 3\sigma_k = -0.3333, & b_1 &= \frac{2}{6} \sum_{k=1}^6 x_k \sin \sigma_k = -0.5774, & b_2 &= \frac{2}{6} \sum_{k=1}^6 x_k \sin 2\sigma_k = 0 \end{aligned}$$

This yields $f(\sigma) = \frac{1}{2} - 0.3333 \cos \sigma - 0.1667 \cos 3\sigma - 0.5774 \sin \sigma$. But variables t and σ are related via Equation 5.57,

$$\sigma = \frac{(6-1)2\pi}{6(0.85-0.10)}(t-0.10) = 6.9813(t-0.1)$$

Therefore, the interpolating trigonometric polynomial is formed as

$$f(t) = \frac{1}{2} - 0.3333 \cos(6.9813(t-0.10)) - 0.1667 \cos(3 \times 6.9813(t-0.10)) - 0.5774 \sin(6.9813(t-0.10))$$

The interpolated value at $t = 0.66$ is

$$f(0.66) = 1.0293$$

Executing the user-defined function `TrigPoly` yields the coefficients of the trigonometric polynomial as well as the plot of this polynomial and the original data. This is shown in [Figure 5.29](#).

```
>> x = [0 0 0 1 1 1];
>> [a, b] = TrigPoly(x, 3, 0.10, 0.85)
```

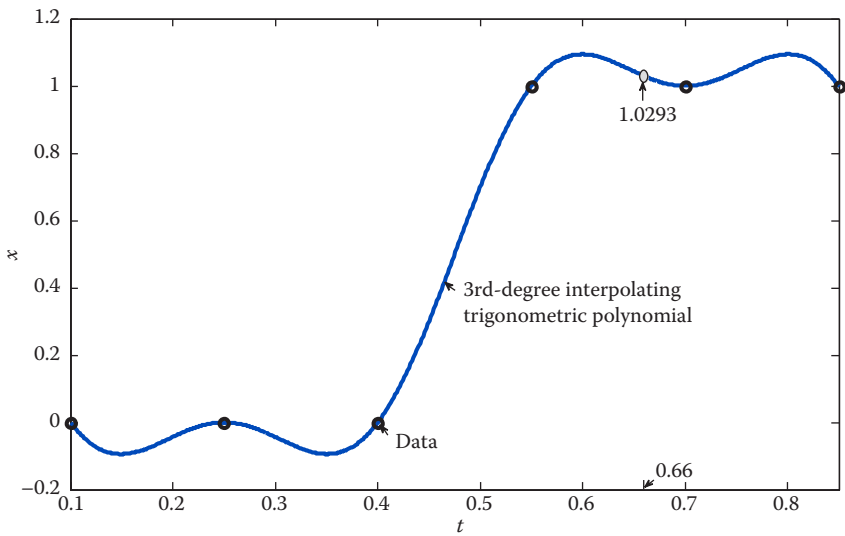


FIGURE 5.29

Fourier interpolation of the data in Example 5.15.

a =
 0.5000
 -0.3333
 -0.0000
 -0.1667

b =
 -0.5774
 0.0000
 0.0000

The numerical results agree with those obtained earlier.

5.7.3 Discrete Fourier Transform

Periodic functions can conveniently be represented by Fourier series. But there are many functions or signals that are not periodic; for example, an impulsive force applied to a mechanical system will normally have a relatively large magnitude and will be applied for a very short period of time. Such non-periodic signals are best represented by the Fourier integral. The Fourier integral of a function can be obtained while taking the Fourier transform of that function. The Fourier transform pair for a continuous function $x(t)$ is defined as

$$\hat{x}(\omega) = \int_{-\infty}^{\infty} x(t)e^{-i\omega t} dt$$

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{x}(\omega)e^{i\omega t} d\omega$$
(5.58)

The first equation gives the Fourier transform $\hat{x}(\omega)$ of $x(t)$. The second equation uses $\hat{x}(\omega)$ to represent $x(t)$ as an integral known as the Fourier integral. In applications, the data is often collected as a discrete set of values and hence $x(t)$ is not available in the form of a continuous function. To that end, a discretized version of Equation 5.58 must be formulated.

Divide an interval $[0, T]$ into N equally spaced subintervals, each of width $h = T/N$. Consider the set of data chosen as $(t_0, x_0), \dots, (t_{N-1}, x_{N-1})$ such that $t_0 = 0, t_1 = t_0 + h, \dots, t_{N-1} = t_0 + (N-1)h$. Note that the point $t_N = T$ is not included.* Equation 5.58 is then discretized as

$$\hat{x}_k = \sum_{n=0}^{N-1} x_n e^{-ik\omega n}, \quad k = 0, 1, \dots, N-1, \quad \omega = \frac{2\pi}{N} \quad (5.59)$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} \hat{x}_k e^{ik\omega n}, \quad n = 0, 1, \dots, N-1 \quad (5.60)$$

where \hat{x}_k is known as the discrete Fourier transform (DFT). Equations 5.59 and 5.60 can be used to compute the Fourier and inverse Fourier transform for a set of discrete data. Calculation of the DFT in Equation 5.59 requires N^2 complex operations. Therefore, even for data of moderate size, such calculations can be quite time-consuming. To remedy that, the fast Fourier transform (FFT) is developed for efficient computation of the DFT. What makes FFT computationally attractive is that it reduces the number of operations by using the results of previous calculations.

5.7.4 Fast Fourier Transform

The FFT algorithm requires roughly $N \log_2 N$ operations as opposed to N^2 by the DFT; and it does so by using the fact that trigonometric functions are periodic and symmetric. For instance, for $N = 100$, the FFT is roughly 15 times faster than the DFT. For $N = 500$, it is about 56 times faster. The first major contribution leading to an algorithm for computing the FFT was made by J. W. Cooley and J. W. Tukey in 1965, known as the Cooley–Tukey algorithm. Since then, a number of other methods have been developed that are essentially consequences of their approach.

The basic idea behind all of these techniques is to decompose, or decimate, a DFT of length N into successively smaller length DFTs. This can be achieved via decimation-in-time or decimation-in-frequency techniques. The Cooley–Tukey method, for example, is a decimation-in-time technique. Here, we will discuss an alternative approach, the Sande–Tukey algorithm, which is a decimation-in-frequency method. The two decimation techniques differ in how they are organized, but they both require $N \log_2 N$ operations. We will limit our presentation to the case $N = 2^p$ (integer p) for which the techniques work best, but analogous methods will clearly work for the general case $N = N_1 N_2 \dots N_m$ where each N_i is an integer.

* Refer to R.W. Ramirez, *The FFT, Fundamentals and Concepts*, Prentice-Hall, 1985.

5.7.4.1 Sande–Tukey Algorithm ($N = 2^p$, $p = \text{integer}$)

We will present the simplified algorithm for the special case $N = 2^p$ where p is some integer. Recall from Equation 5.59 that the DFT is given by

$$\hat{x}_k = \sum_{n=0}^{N-1} x_n e^{-ik(2\pi/N)n}, \quad k = 0, 1, \dots, N-1 \quad (5.61)$$

Define the weighting function $W = e^{-(2\pi/N)i}$ so that Equation 5.61 may also be written as

$$\hat{x}_k = \sum_{n=0}^{N-1} x_n W^{kn}, \quad k = 0, 1, \dots, N-1 \quad (5.62)$$

We next divide the sample of length N in half, each half containing $N/2$ points, and write Equation 5.61 as

$$\hat{x}_k = \sum_{n=0}^{(N/2)-1} x_n e^{-ik(2\pi/N)n} + \sum_{n=N/2}^{N-1} x_n e^{-ik(2\pi/N)n}$$

Since summations can only be combined if their indices cover the same range, introduce a change of variables in the second summation and rewrite this last equation as

$$\begin{aligned} \hat{x}_k &= \sum_{n=0}^{(N/2)-1} x_n e^{-ik(2\pi/N)n} + \sum_{n=0}^{(N/2)-1} x_{n+N/2} e^{-ik(2\pi/N)(n+N/2)} \\ &= \text{Combine} \sum_{n=0}^{(N/2)-1} [x_n + e^{-i\pi k} x_{n+N/2}] e^{-2\pi kni/N} \end{aligned} \quad (5.63)$$

But

$$e^{-i\pi k} = \cos k\pi - i \sin k\pi = (-1)^k = \begin{cases} 1 & \text{if } k = \text{even} \\ -1 & \text{if } k = \text{odd} \end{cases}$$

Therefore, the expression for \hat{x}_k will depend on whether the index is even or odd. For even index, Equation 5.63 yields

$$\hat{x}_{2k} \stackrel{\text{Substitute}}{=} \sum_{n=0}^{(N/2)-1} [x_n + x_{n+N/2}] e^{-2\pi(2k)ni/N} \stackrel{\text{Rewrite}}{=} \sum_{n=0}^{(N/2)-1} [x_n + x_{n+N/2}] e^{-[(2\pi/N)i]2kn}$$

For odd index,

$$\hat{x}_{2k+1} \stackrel{\text{Substitute}}{=} \sum_{n=0}^{(N/2)-1} [x_n - x_{n+N/2}] e^{-2\pi(2k+1)ni/N} \stackrel{\text{Rewrite}}{=} \sum_{n=0}^{(N/2)-1} [x_n - x_{n+N/2}] e^{-[(2\pi/N)i]n} e^{-[(2\pi/N)i]2kn}$$

In terms of $W = e^{-(2\pi/N)i}$, defined earlier,

$$\hat{x}_{2k} = \sum_{n=0}^{(N/2)-1} [x_n + x_{n+N/2}]W^{2kn} \quad (5.64)$$

$$\hat{x}_{2k+1} = \sum_{n=0}^{(N/2)-1} \{[x_n - x_{n+N/2}]W^n\}W^{2kn} \quad (5.65)$$

Next, define

$$\begin{aligned} y_n &= x_n + x_{n+N/2} \\ z_n &= [x_n - x_{n+N/2}]W^n, \quad n = 0, 1, \dots, (N/2)-1 \end{aligned} \quad (5.66)$$

Inspired by Equation 5.62, it is easy to see that the summations in Equations 5.64 and 5.65 simply represent the transforms of y_n and z_n . That is,

$$\begin{aligned} \hat{x}_{2k} &= \hat{y}_k \\ \hat{x}_{2k+1} &= \hat{z}_k, \quad k = 0, 1, \dots, (N/2)-1 \end{aligned}$$

Consequently, the original N -point computation has been replaced by two $(N/2)$ -point computations, each requiring $(N/2)^2 = N^2/4$ operations for a total of $N^2/2$. Comparing with N^2 for the original data, the algorithm manages to reduce the number of operations by a factor of 2. The decomposition continues, with the number of sample points divided by two in each step, until $N/2$ two-point DFTs are computed. To better understand how the scheme works, we present the details involving an 8-point sample.

5.7.4.2 Case Study: $N = 2^3 = 8$

An 8-point DFT is to be decomposed successively using Sande–Tukey algorithm (decimation-in-frequency) into smaller DFTs. Figure 5.30 shows the details in the first stage where two 4-point DFTs are generated. The intersections that are accompanied by “+” and/or “–” signs act as summing junctions. For example, by Equation 5.66 we have

$$y_0 = x_0 + x_4, \quad z_0 = (x_0 - x_4)W^0$$

The operation $y_0 = x_0 + x_4$ is handled by a simple addition of two signals. To obtain z_0 , we first perform $x_0 - x_4$, then send the outcome to a block of W^0 . The same logic applies to the remainder of the sample.

Next, each of the four-point DFTs will be decomposed into two 2-point DFTs, which will mark the end of the process for the case of $N = 8$; see Figure 5.31. Also $N = 2^p = 8$ implies $p = 3$, and there are exactly three stages involved in the process. Furthermore, since $N = 8$ we have $W = e^{-(2\pi/N)i} = e^{-(\pi/4)i}$, and

$$W^0 = 1, \quad W^1 = e^{-(\pi/4)i} = \frac{\sqrt{2}}{2}(1-i), \quad W^2 = e^{-(\pi/2)i} = -i, \quad W^3 = e^{-(3\pi/4)i} = \frac{-\sqrt{2}}{2}(1+i)$$

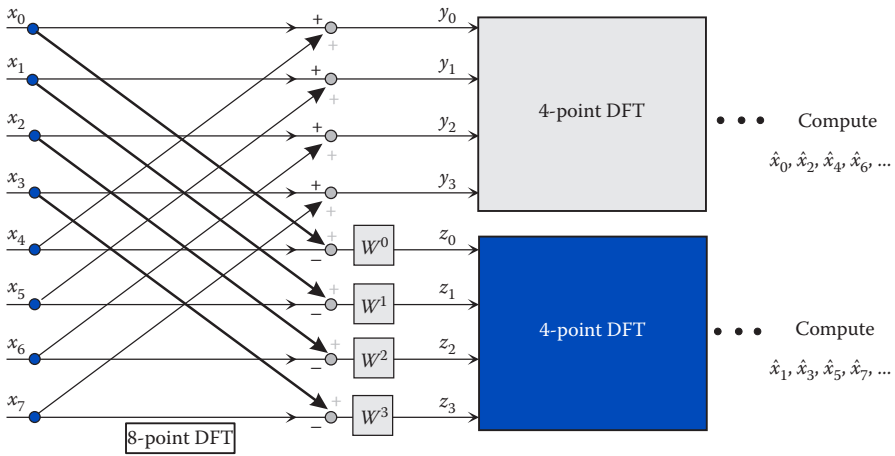


FIGURE 5.30 First stage of decomposition (decimation-in-frequency) of an 8-point DFT into two four-point DFTs.

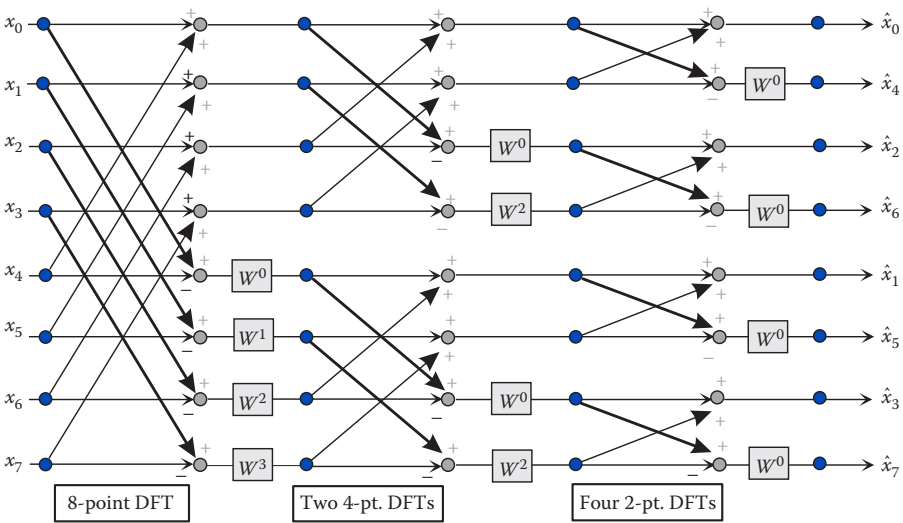


FIGURE 5.31 Complete decomposition (decimation-in-frequency) of an 8-point DFT.

The computed Fourier coefficients appear in a mixed order but can be unscrambled using bit reversal as follows: (1) express the subscripts 0 through 7 in binary form,* (2) reverse the bits, (3) express the reversed bits in decimal form. The details are depicted in [Table 5.17](#).

5.7.4.3 Cooley–Tukey Algorithm ($N = 2^p$, $p = \text{integer}$)

The flow graph for the Cooley–Tukey algorithm is shown in [Figure 5.32](#). The initial sample is divided into groups of even-indexed and odd-indexed data points, but the final outcome appears in correct order.

* For example, $5 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (101)_2$.

TABLE 5.17

Bit Reversal Applied to the Scenario in Figure 5.31

Mixed Order	Binary Subscripts	Reverse Bits	Unscrambled Order
\hat{x}_0	0→000	000→0	\hat{x}_0
\hat{x}_4	4→100	001→1	\hat{x}_1
\hat{x}_2	2→010	010→2	\hat{x}_2
\hat{x}_6	6→110	011→3	\hat{x}_3
\hat{x}_1	1→001	100→4	\hat{x}_4
\hat{x}_5	5→101	101→5	\hat{x}_5
\hat{x}_3	3→011	110→6	\hat{x}_6
\hat{x}_7	7→111	111→7	\hat{x}_7

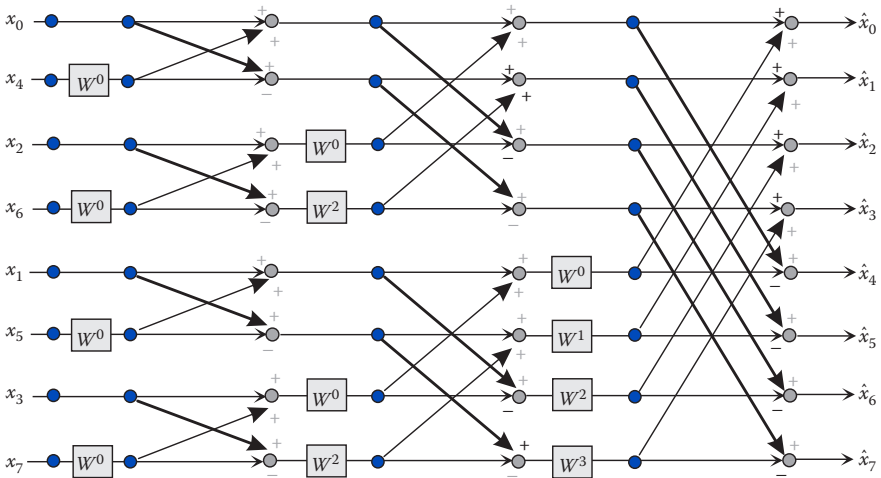


FIGURE 5.32

Complete decomposition (decimation-in-time) of an 8-point DFT.

5.7.5 MATLAB Built-In Function `fft`

The MATLAB built-in function `fft` computes the DFT of an N -dimensional vector using the efficient FFT method. The DFT of the evenly spaced data points $x(1), x(2), \dots, x(N)$ is another N -dimensional vector $X(1), X(2), \dots, X(N)$ where

$$X(k) = \sum_{n=1}^N x(n)e^{-2\pi i(k-1)(n-1)/N}, \quad k = 1, 2, \dots, N$$

5.7.5.1 Interpolation Using `fft`

A set of equally spaced data $(t_k, x_k), k = 1, 2, \dots, N$ is interpolated using `fft` as follows. The data is first treated as

$$(\sigma_1, x_1), (\sigma_2, x_2), \dots, (\sigma_N, x_N)$$

where σ_k ($k = 1, 2, \dots, N$) are equally spaced along the interval $[0, 2\pi)$, that is,

$$\sigma_1 = 0, \sigma_2 = \frac{2\pi}{N}, \sigma_3 = 2\left(\frac{2\pi}{N}\right), \dots, \sigma_N = (N-1)\left(\frac{2\pi}{N}\right)$$

The FFT of the vector $[x_1 \ x_2 \ \dots \ x_N]$ is computed using MATLAB built-in function `fft`. The resulting data is then used in Equation 5.60, with index ranging from 1 to N , to reconstruct x_k . Finally, the data is transformed to its original form and plotted.

EXAMPLE 5.16: INTERPOLATION USING FFT

Table 5.18 contains the equally spaced data for one period of a periodic waveform. Construct and plot the interpolating function for this data using the MATLAB function `fft`.

Solution

We will accomplish this in two steps: First we compute the FFT of the given data, and then use the transformed data to find the interpolating function by essentially reconstructing the original waveform. Note that the reconstruction is done via Equation 5.60, rewritten as

$$x_n = \frac{1}{16} \sum_{k=1}^{16} \hat{x}_k e^{2\pi i(k-1)(n-1)/16}$$

where we will use $n = 1:200$ for plotting purposes. The 16 values of \hat{x}_k are obtained as follows:

```
>> x = [2.95 2.01 0.33 .71 .11 .92 -0.16 .68 -1.57 -1.12 -0.58 -0.69 -0.21
-.54 -0.63 -2.09];
>> Capx = fft(x)'
```

Capx =

```
0.1200
5.1408 + 6.1959i
```

TABLE 5.18

Data in Example 5.16

t	x	t	x
0.0	2.95	0.8	-1.57
0.1	2.01	0.9	-1.12
0.2	0.33	1.0	-0.58
0.3	0.71	1.1	-0.69
0.4	0.11	1.2	-0.21
0.5	0.92	1.3	-0.54
0.6	-0.16	1.4	-0.63
0.7	0.68	1.5	-2.09

0.8295 + 1.9118i
 4.4021 + 5.0122i
 2.3200 + 2.6600i
 4.0157 + 3.7006i
 2.1305 + 0.8318i
 4.5215 + 3.6043i
 0.3600
 4.5215 - 3.6043i
 2.1305 - 0.8318i
 4.0157 - 3.7006i
 2.3200 - 2.6600i
 4.4021 - 5.0122i
 0.8295 - 1.9118i
 5.1408 - 6.1959i

Let $\hat{x}_k = \alpha_k + i\beta_k$ so that

$$x_n = \frac{1}{16} \sum_{k=1}^{16} \hat{x}_k e^{2\pi i(k-1)(n-1)/16} = \frac{1}{16} \sum_{k=1}^{16} [\alpha_k + i\beta_k][\cos(2\pi(k-1)(n-1)/16) + i\sin(2\pi(k-1)(n-1)/16)]$$

Note that $\alpha_2 = \alpha_{16}, \dots, \alpha_8 = \alpha_{10}$ and $\beta_2 = -\beta_{16}, \dots, \beta_8 = -\beta_{10}$. Also α_9 multiplies $\cos((n-1)\pi)$ which alternates between 1 and -1 so that over a range of 200 values for n will cancel out. Finally, α_1 multiplies $\cos 0 = 1$. Then, the above can be written as

$$x_n = \frac{1}{16} \alpha_1 + \frac{1}{8} \sum_{k=2}^8 [\alpha_k \cos(2\pi(k-1)(n-1)/16) - \beta_k \sin(2\pi(k-1)(n-1)/16)]$$

The following MATLAB code will use this to reconstruct the original waveform.

```

x = [2.95 2.01 .33 .71 .11 .92 -.16 .68 -1.57 -1.12 -.58 -.69 -.21 -.54 -.63 -2.09];
N = length(x);
tN = 1.5; t1 = 0;
Capx = fft(x); % Compute FFT of data
% Treat data as equally spaced on [0, 2*pi)
h = 2*pi/N; s = 0:h:2*pi-h; s = s';
ss = linspace(0,2*pi*(N-1)/N,200); % 200 points for plotting purposes
y = zeros(200,1); % Pre-allocate
% Start reconstruction & interpolation
for i = 1:200,
    y(i) = Capx(1)/2;
    for k = 1:8,
        y(i) = y(i) + real(Capx(k+1))*cos(k*ss(i)) - imag(Capx(k+1))*sin(k*ss(i));
    end
    y(i) = (1/8)*y(i);
end
% Transform data to original form
t = N*((tN-t1)/(2*pi*(N-1)))*s + t1;
tt = N*((tN-t1)/(2*pi*(N-1)))*ss + t1;

plot(tt,y,t,x,'o') % Figure 5.33

```

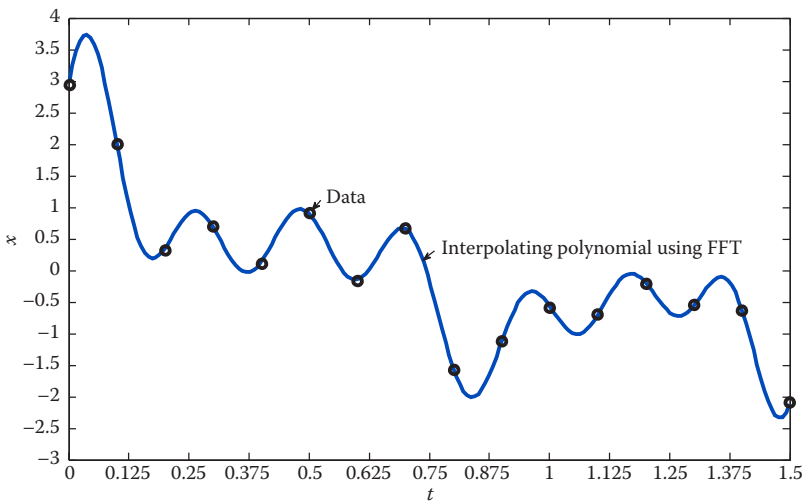


FIGURE 5.33
Interpolations using FFT in Example 5.16.

PROBLEM SET (CHAPTER 5)
Linear Regression (Section 5.2)

In Problems 1 through 9, for each set of data,

- Using least-squares regression, find a straight line that best fits the data.
- Confirm the results by executing `LinearRegression`.

1. [Table P1](#)

TABLE P1

x	y
0.5	0.22
0.6	0.41
0.7	0.69
0.8	0.88
0.9	1.14

2. [Table P2](#)

TABLE P2

x	y
0.1	3.4
0.4	3.78
0.7	4.01
1.0	4.35
1.3	4.57
1.6	4.88

3. Table P3

TABLE P3

x	y
-2	2.2
0	1.5
2	0.8
4	-0.1
6	-0.8
8	-1.3

4. Table P4

TABLE P4

x	y
-1.2	-0.43
-0.8	0.02
-0.4	0.51
0.0	1.05
0.4	1.60

5. Table P5

TABLE P5

x	y
2	2.5
3	3.1
5	3.4
7	4.7
8	5.1
9	4.9
11	6.1

6. Table P6

TABLE P6

x	y
5	6.8
11	19.7
14	26
26	46.5
30	60
41	80.4
46	88.1

7. Table P7

TABLE P7

x	y
1	7.27
2	9.7
3	10.2
4	12.3
5	16.6
6	19
7	21.4
8	24
9	25.9
10	29

8. Table P8

TABLE P8

x	y
-1	2.3
1	2.9
2	3.5
4	4.7
5	5.0
7	5.9
8	6.7
10	7.4
11	8.1
13	9.3

9. Table P9

TABLE P9

x	y
-3.2	5.89
-2.1	4.57
-0.5	2.82
0.8	1.23
1.9	-0.07
3.2	-1.69
4.2	-2.92
4.9	-3.74
5.5	-4.51
6.8	-5.91


10.  The yield of a chemical reaction (%) at several temperatures (°C) is recorded in [Table P10](#).
- Execute the user-defined function `LinearRegression` to find and plot a straight line that best fits the data.
 - Using the line fit, find an estimate for the yield at 280°C.

TABLE P10

x (Temperature °C)	y (Yield %)
165	79.4
190	83.5
215	84.7
230	86.2
245	88.1
260	89.4
275	91.9
290	92.9
300	95.1
310	96.3


11. In a linear coiled spring, the relation between spring force (F) and displacement (x) is described by $F = kx$, where k is the (constant) coefficient of stiffness of the spring. Testing on a certain spring has led to the data recorded in [Table P11](#). All parameter values are in consistent physical units.
-  Execute the user-defined function `LinearRegression` to find and plot a straight line that best fits the data.
 - Using (a), find the estimated value for the coefficient of stiffness, and the displacement corresponding to $F = 150$.

TABLE P11

x (Displacement)	F (Force)
0.2	43.5
0.3	65.7
0.5	109.8
0.6	133
0.8	176.2
0.9	198.2
1.1	242.3
1.3	285.8
1.4	307.8
1.6	352.2


12.  Students' scores on the mathematics portion of the SAT exam and their GPA follow a linear probabilistic model. Data from 10 students have been collected and recorded in [Table P12](#).
- Execute the user-defined function `LinearRegression` to find and plot a straight line that best fits the data.
 - Using the line fit, find an estimate for the GPA of a student whose test score was 560.

TABLE P12

x (Test score)	y (GPA)
360	1.70
400	1.80
450	1.90
480	1.95
500	2.15
520	2.30
590	2.80
610	3.00
640	3.25
740	3.80

Linearization of Nonlinear Data (Section 5.3)



13.  Show that in Example 5.3 an exponential function is not a suitable fit for the data.
-  In Problems 14 through 18, fit an appropriate function (exponential, power, or saturation) to the given data.
14. Data in [Table P14](#)

TABLE P14

x	y
0.5	0.11
1	0.19
1.5	0.27
2	0.34
2.7	0.40
3.5	0.50
4.2	0.52
5	0.56
6	0.60
7	0.66

15. Data in [Table P15](#)

TABLE P15

x	y
0	1.3
0.2	2.1
0.4	3.0
0.6	5.2
0.8	8.4
1.0	13.5
1.2	22
1.4	33.5
1.6	53
1.8	85.4

16. Data in [Table P16](#)

TABLE P16

x	y
0.2	2.8
0.3	3.6
0.5	5.3
0.6	6.4
0.8	8.5
1.0	12.4
1.1	15.3
1.3	22.2

17. Data in [Table P17](#)

TABLE P17

x	y
1	0.27
2	0.51
3.5	0.60
5	0.74
6	0.79
8	0.82
9.5	0.90
10	0.87

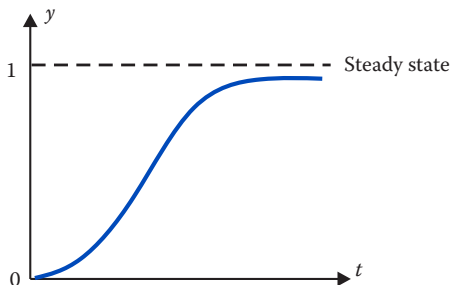
TABLE P18

x	y
1	3.0
1.6	3.4
2	3.7
2.5	3.8
3	4.2
3.4	4.5
4	4.7
4.6	4.8
5	5.0
5.8	5.2

19. In many applications involving chemical processes, the experimental data follows an s-shaped curve as in [Figure 5.34](#), where the data approaches a steady-state value of 1. For these cases, curve fitting is done by approximating the s-shaped curve by $y = 1 + Ae^{-\alpha t}$ where $A < 0$ since $y < 1$ for all data, and $\alpha > 0$. Rearrange and take the natural logarithm to obtain

$$\ln|y - 1| = -\alpha t + \ln|A|$$

so that $\ln|y - 1|$ versus t is linear with a slope of $-\alpha$ and an intercept of $\ln|A|$. The slope and the intercept can be found by linear regression. Apply this procedure to the data in [Table P19](#) to determine the parameters A and α . Plot the original data and the curve fit just obtained.

**FIGURE 5.34**

An s-shaped curve—chemical process.

TABLE P19

t	y
1	0.38
1.5	0.43
2.5	0.68
3	0.79
4	0.90
5.5	0.94
7	0.96
8	0.97
9	0.98
10	0.99

20. Repeat Problem 19 for the data in [Table P20](#).

TABLE P20

t	y
1	0.68
3	0.79
5	0.83
7	0.92
9	0.95
11	0.97
13	0.97
15	0.98
17	0.98
19	0.99

Polynomial Regression (Section 5.4)

21. Write a user-defined function $[a_3, a_2, a_1, a_0] = \text{CubicRegression}(x, y)$ that uses the cubic least-squares regression approach to find the third-degree polynomial that best fits a set of data. The coefficients a_3, a_2, a_1, a_0 are found by expressing the appropriate 4×4 system of equations in matrix form and solving by “\” in MATLAB. The function also should return the plot of the data and the best cubic polynomial fit.
22. Using `LinearRegression` and `CubicRegression` (Problem 21) find and plot (same figure) the first- and third-order polynomials that best fit the data in [Table P22](#).

TABLE P22

x	y
0.8	4.9
2.6	5.7
4	6.3
4.8	6.1
6.5	7.2

23.  Repeat Problem 22 for the data in [Table P23](#).

TABLE P23

x	y
0.0	1.2
0.3	1.8
0.6	2.8
0.9	3.6
1.1	4.4
1.4	4.1


24.  Using the user-defined functions of Section 5.4 find and plot (same figure) the straight line and the second-order polynomial that best fit the data in [Table P24](#). Discuss the results.

TABLE P24

x	y
0	2.0
1	4.2
2	5.7
3	6.7
4	7.9
5	9.1

25.  Repeat Problem 24 for the data in [Table P25](#).

TABLE P25

x	y
0.1	0.8
0.3	1.0
0.5	1.5
0.7	2.2
0.9	3.3

26. Using the `polyfit` and `polyval` functions find and plot (same figure) the third- and fourth-degree polynomials that best fit the data in [Table P26](#).

TABLE P26

x	y
1	1.2
2	4
3	4.8
4	6
5	7.1
6	8

27. Using the `polyfit` and `polyval` functions find and plot (same figure) the third- and fifth-degree polynomials that best fit the data in [Table P27](#). Discuss the results.

TABLE P27

x	y
0	3.4
1	5.1
2	6.0
3	7.2
4	9.3
5	10.1

28. During the free fall of a heavy object, the relationship between the velocity v of the object and the force r resisting its motion—both in consistent physical units—is described by the data in [Table P28](#).
- Using the `polyfit` and `polyval` functions find and plot the second-degree polynomial that best fits the data.
 - Using the result of (a) find the force of resistance corresponding to a velocity of 1.75.

TABLE P28

Velocity (v)	Resistance (r)
0	0
0.4	0.11
0.8	0.52
1.2	1.03
1.6	1.78
2.0	2.72
2.4	4.03
2.8	5.46
3.2	7.24


29.  Consider the data in [Table P29](#) generated by the function $y = 3^{x-0.4}$. Using the `polyfit` and `polyval` functions find the estimated value at $x = 1.3$ given by a third-degree and a fourth-degree polynomial fit, calculate the % relative error for each estimate, and comment on accuracy.

TABLE P29

x	$y = 3^{x-0.4}$
0.20	0.8027
0.50	1.1161
0.90	1.7321
1.20	2.4082
1.40	3.0000
1.70	4.1712


30.  Using the `polyfit` and `polyval` functions find and plot the second-degree polynomial that best fits the data in [Table P30](#). Also find the estimated value at $x = 1.85$.

TABLE P30

x	y
0.53	8.03
0.95	6.69
1.30	5.62
1.72	4.61
2.01	3.89
2.34	3.38
2.93	2.61
3.12	2.36
4.08	1.98
4.40	2.12

Polynomial Interpolation (Section 5.5)

Lagrange Interpolation



31. Given the data in [Table P31](#),
-  Interpolate at $x = 0.75$ using the second-degree Lagrange interpolating polynomial.
 -  Confirm the results by executing `LagrangeInterp`.

TABLE P31

x	y
0.2	0.43
0.5	0.32
0.9	0.13

32. Given the data in [Table P32](#),

- Interpolate at $x = 0.85$ using the second-degree Lagrange interpolating polynomial.
- Confirm the results by executing `LagrangeInterp`.

TABLE P32

x	y
0.3	-0.25
0.7	-0.41
1.0	-0.16

33. Given the data in [Table P33](#),

- Interpolate at $x = 3$ with a first-degree Lagrange polynomial using two most suitable data points.
- Interpolate at $x = 3$ with a second-degree Lagrange polynomial using three most suitable data points.
- Calculate the % relative errors for the results of (a) and (b), and discuss.

TABLE P33

x	$y = \sin(x/3)$
0	0
1	0.3272
2	0.6184
4	0.9719

34. Given the data in [Table P34](#),

- Interpolate at $x = 2.5$ with a first-degree Lagrange polynomial using two most suitable data points.
- Interpolate at $x = 2.5$ with a second-degree Lagrange polynomial using three most suitable data points.
- Calculate the % relative errors for the results of (a) and (b), and discuss.

TABLE P34

x	$y = \log_{10}(x)$
1	0
1.5	0.1761
2	0.3010
3	0.4771
5	0.6990

35. Using `format long` and the user-defined function `LagrangeInterp`, given the data in [Table P35](#), determine the four most suitable data points to interpolate at $x = 0.6$ with a third-degree Lagrange polynomial.

TABLE P35

x	$y = e^{-2x/3}$
0.2	0.8752
0.4	0.7659
0.5	0.7165
0.8	0.5866
1.1	0.4803
1.3	0.4204

36. Using `format long` and the user-defined function `LagrangeInterp`, given the data in [Table P36](#), determine the four most suitable data points to interpolate at $x = 0.5$ with a third-degree Lagrange polynomial.

TABLE P36

x	$y = 3^{-x}$
0.1	0.8960
0.2	0.8027
0.4	0.6444
0.7	0.4635
0.9	0.3720
1.1	0.2987

37. Consider the data in [Table P37](#).

- Interpolate at $x = 1.7$ via a second-degree Lagrange polynomial by using two suitable sets of three data points, and calculate the % relative errors for both cases.
- Confirm the results of (a) in MATLAB.

TABLE P37

x	$y = x^{2/3}$
0.2	0.3420
1.2	1.1292
3	2.0801
6	3.3019

38. The measured velocity of a moving object is recorded in [Table P38](#). It is desired to estimate the velocity at $t = 12$ seconds, between the last two data points where there exists a relatively large gap. Plot the entire data. In the same graph, plot the

second-degree Lagrange interpolating polynomial that goes through the last three data points, and the third-degree Lagrange interpolating polynomial that goes through the last four data points. Also find the interpolated values at $t = 12$ given by the two polynomials.

TABLE P38

Time (t), s	Velocity (v), ft/s
2	120
4	564
7	873
9	1012
15	1670

Newton Interpolation (Divided Differences)

39. Given the data in [Table P39](#),

- ✎ Construct the divided differences table and use Newton interpolating polynomials to interpolate at $x = 0.25$ using the first two points, the first three points, and the entire data.
- 🔍 Confirm the results of (a) by executing `NewtonInterp`.

TABLE P39

x	y
0	1
0.5	0.9098
0.9	0.7725
1.2	0.6626

40. Given the data in [Table P40](#),

- ✎ Construct the divided differences table and use Newton interpolating polynomials to interpolate at $x = 0.3$ using the first two points, the first three points, and the entire data.
- 🔍 Confirm the results of (a) by executing `NewtonInterp`.

TABLE P40

x	y
0	1
0.4	2.68
0.8	5.79
1	8.15

41.  Consider the data in [Table P41](#).

- Construct a divided differences table and interpolate at $x = 1.75$ using the third-degree Newton interpolating polynomial $p_3(x)$.
- Suppose one more point ($x = 3, y = 9.11$) is added to the data. Update the divided differences table of (a) and interpolate at $x = 1.75$ using the fourth-degree Newton interpolating polynomial $p_4(x)$.

TABLE P41

x	y
1	1.22
1.5	2.69
2	4.48
2.5	6.59

42.  Consider the data in [Table P42](#).

- Construct a divided differences table and interpolate at $x = 4$ using the third-degree Newton interpolating polynomial $p_3(x)$.
- Suppose one more point ($x = 7, y = 0.18$) is added to the data. Update the divided-difference table from (a) and interpolate at $x = 4$ using the fourth-degree Newton interpolating polynomial $p_4(x)$.

TABLE P42

x	y
1	1
3	0.45
5	0.26
6	0.21

43. Given the data in [Table P43](#),




-  Construct a divided differences table and interpolate at $x = 2.6$ and $x = 4.4$ using the fourth-degree Newton interpolating polynomial $p_4(x)$.
-  Confirm the results by executing the user-defined function `NewtonInterp`.

TABLE P43

x	y
1	0.69
2	1.10
3	1.39
4	1.61
6	1.95

44. Given the data in [Table P44](#),

-  Construct a divided differences table and interpolate at $x = 2.7$ and $x = 5.3$ using the fourth-degree Newton interpolating polynomial $p_4(x)$.

- b.  Confirm the results by executing the user-defined function `NewtonInterp`.

TABLE P44

x	y
1	0.89
2	1.81
3	2.94
4	4.38
6	8.72


45.  Consider the data in [Table P45](#). It is desired to find an estimate at $x = 10$, between the last two data points where there exists a relatively large gap. Plot the entire data. In the same graph, plot the second-degree Newton interpolating polynomial that goes through the last three data points, and the third-degree Newton interpolating polynomial that goes through the last four data points. Also find the interpolated values at $x = 10$ given by the two polynomials.

TABLE P45

x	y
1	1
2	1.4142
3.5	1.8708
5	2.2361
7	2.6458
12	3.4641


46.  Consider the data in [Table P46](#). It is desired to find an estimate at $x = 7$. Plot the entire data. In the same graph, plot the first-degree Newton interpolating polynomial that goes through the last two data points, the second-degree polynomial that goes through the last three data points, and the third-degree polynomial that goes through the last four data points. Also find the interpolated values at $x = 7$ given by the three polynomials. Comment on the accuracy of each estimate.

TABLE P46

x	$y = e^{x/3}$
1	1.3956
3	2.7183
4	3.7937
5	5.2945
6	7.3891
8	14.3919

Newton Interpolation (Forward Differences)


47.  For the data in [Table P47](#), construct a forward-differences table and interpolate at $x = 2.3$ using Newton interpolating polynomial $p_4(x)$.

TABLE P47

x	y
1	1.25
2	3.25
3	7.25
4	13.25
5	21.25

48. ✎ Consider the data in [Table P48](#).

- Construct a forward-differences table and interpolate at $x = 1.26$ using Newton interpolating polynomials $p_3(x)$, going through the first four data points, and $p_4(x)$.
- Suppose a new point ($x = 1.5$, $y = 4.27$) is added to the data. Interpolate at $x = 1.26$ using Newton interpolating polynomial $p_5(x)$.

TABLE P48

x	y
1.0	1.30
1.1	1.75
1.2	2.27
1.3	2.86
1.4	3.52

49. ✎ For the data in [Table P49](#), construct a forward-differences table and interpolate at $x = 2.75$ using Newton interpolating polynomial $p_5(x)$.

TABLE P49

x	y
1	0.92
1.5	0.80
2	0.64
2.5	0.46
3	0.29
3.5	0.14

50. 🐛 Write a user-defined function with syntax `yi = Newton_FD(x, y, xi)` that finds the Newton forward-difference interpolating polynomial for the equally-spaced data (x, y) and uses this polynomial to interpolate at x_i and returns the interpolated value in y_i . For the data in [Table P50](#), use the entire data to find the interpolated values at $x = 3.7$ and $x = 7.3$ by executing `Newton_FD`. Confirm both results by executing `NewtonInterp`.

TABLE P50

x	y
1	1.0000
2	0.4414
3	0.2598
4	0.1818
5	0.1381
6	0.1095
7	0.0939
8	0.0753
9	0.0675
10	0.0497


51.  The user-defined function `Newton_FD` (Problem 50) is to be used throughout. Given the data in [Table P51](#), interpolate at $x = 11$ by using $p_5(x)$, which goes through the first six data points, and $p_9(x)$.

TABLE P51

x	y
2	1.6840
4	2.1012
6	2.3465
8	2.6913
10	2.8469
12	3.1246
14	3.4723
16	3.6327
18	3.9543
20	4.2185



52.  The user-defined function `Newton_FD` (Problem 50) is to be used throughout. Consider the data in [Table P52](#). Plot the entire data. In the same graph, plot

TABLE P52

x	y
1	1.4422
2	1.8171
3	2.2240
4	2.6207
5	3.0000
6	2.3875
7	2.0986
8	1.6749
9	1.8903
10	2.3458

the third-degree Newton interpolating polynomial that goes through the last four data points, the fourth-degree polynomial that goes through the last five data points, and the ninth-degree polynomial that goes through the entire data. Also find the interpolated values at $x = 7.5$ given by the three polynomials.

Spline Interpolation (Section 5.6)

 In Problems 53 through 56, find the quadratic splines for the given data and interpolate at the specified point(s). Assume $S_1''(x_1) = 0$.

 Plot the resulting splines in MATLAB.

53. [Table P53](#), $x = 0.8$, $x = 2.6$

TABLE P53

x	y
0	1
1	2
2	5
3	9

54. [Table P54](#), $x = 3.75$

TABLE P54

x	y
1	1.2
3	2.3
4	1.2
6	2.9

55. [Table P55](#), $x = 5.5$, $x = 9$

TABLE P55

x	y
3	1
7	4
10	5
13	2

56. [Table P56](#), $x = 5.2$, $x = 7.8$

TABLE P56

x	y
1	10
4	8
6	12
9	14


57.  Consider the data in [Table P57](#).
- Find the quadratic splines, assuming $S_1''(x_1) = 0$.
 - Find the cubic splines with clamped boundary conditions $p = 1, q = -1$.
 - Plot the splines obtained in (a) and (b).
 - Find the interpolated value at $x = 1.6$ using the splines of (a) and (b), compare with the true value, and discuss.

TABLE P57

x	$y = 10 - e^{-x/2}$
0.1	9.0488
0.5	9.2212
1	9.3935
2	9.6321



58.  Repeat Problem 57 but this time assume the cubic splines satisfy free boundary conditions.
59.  For the data in [Table P59](#) construct and plot cubic splines that satisfy
- Clamped boundary conditions $p = -1, q = -0.5$.
 - Free boundary conditions.

TABLE P59

x	y
1	5
4	1
6	2
8	0.5


60.  For the data in [Table P60](#) construct and plot cubic splines that satisfy
- Clamped boundary conditions $p = 0, q = 0.3$.
 - Free boundary conditions.

TABLE P60

x	y
1	1
3	2
5	5
8	6

61.  The yield of a certain chemical reaction at various temperatures is recorded in [Table P61](#). Find the reaction yield at 270°C by using

- interpl with the “spline” option.
- Clamped ($p = -1, q = 1$) cubic spline interpolation. Plot this spline, the one from (a), and the original data in a single graph.
- Clamped ($p = -0.5, q = 0.5$) cubic spline interpolation. Plot this spline, the one from (a), and the original data in a single graph. Compare with (b) and discuss the results.

TABLE P61

Temperature (°C) x	Reaction Yield (%) y
160	78.3
180	81.4
195	84.5
225	85.1
250	89.3
280	91.7
300	94.8


62.  In an exercise session, maximum heart rates for eight individuals of different ages have been recorded as shown in [Table P62](#). Find the maximum heart rate of a 43-year-old individual by using
- interpl.
 - Clamped ($p = -1, q = -1$) cubic spline interpolation.

TABLE P62

Age x	Max. Heart Rate y
15	202
20	195
25	190
30	184
35	178
40	173
45	169
50	160


63.  The data in [Table P63](#) is generated by the function $f(x) = 1/(1 + 2x^2)$.
- Construct and plot the cubic splines with clamped boundary conditions $p = 0.1, q = -0.1$. Also plot the original function and the given data. Interpolate at $x = 1.8$ and compare with the true value at that point.
 - Repeat (a) for boundary conditions $p = 0.2, q = -0.2$. Discuss the results.

TABLE P63

x	y
-2	0.1111
-1	0.3333
0	1.0000
1	0.3333
2	0.1111


64.  For the data in [Table P64](#) construct and plot the cubic splines using `interp1` and find the interpolated value at $x = 3.5$. Repeat for cubic splines with clamped boundary conditions $p = -0.2$, $q = 0.2$, and compare the results.

TABLE P64

x	y
-4	0
-3	0
-2	0
-1	2.3
0	4
1	2.3
2	0
3	0
4	0



65.  The data in [Table P65](#) shows the (experimental) compressibility factor for air at several pressures when temperature is fixed at 180°K. Construct and plot the cubic splines using `interp1` and find the interpolated value at $x = 275$. Repeat for a third-degree polynomial using `polyfit` and compare the results.

TABLE P65

Pressure (bars) x	Comp. Factor y	Pressure (bars) x	Comp. Factor y
1	0.9967	100	0.7084
5	0.9832	150	0.7180
10	0.9660	200	0.7986
20	0.9314	250	0.9000
40	0.8625	300	1.0068
60	0.7977	400	1.2232
80	0.7432	500	1.4361



Source: *Perry's Chemical Engineers' Handbook* (6th edition), McGraw-Hill, 1984.

66.  Consider the data for $x = 0:20$ generated by the Bessel function of the first kind of order one $J_1(x)$, which in MATLAB is handled by `besselj(1, x)`. Construct

and plot the cubic splines using `interp1`, interpolate at $x = 9.5$ and compare with the actual value at that point.

Fourier Approximation and Interpolation (Section 5.7)

In Problems 67 through 74, for each given set of data,

-  Find the approximating or interpolating trigonometric polynomial of the indicated degree.
-  Confirm the results of (a) by executing the user-defined function `TrigPoly`.

67. [Table P67](#), $m = 2$

TABLE P67

t	x
1	0.9
1.3	1
1.6	-1
1.9	-0.8
2.2	0.9
2.5	1

68. [Table P68](#), $m = 2$

TABLE P68

t	x
0.3	1
0.4	0.9
0.5	0
0.6	0.1
0.7	0.8
0.8	0.9

69. [Table P69](#), $m = 2$

TABLE P69

t	x
0.6	-0.60
0.8	0.52
1.0	0.98
1.2	0.75
1.4	1.03

70. Table P70, $m = 3$

TABLE P70

t	x
2	1.40
2.3	1.06
2.6	0.77
2.9	0.15
3.2	-0.62
3.5	0.31

71. Table P71, $m = 3$

TABLE P71

t	x
1.5	1.05
1.7	1.85
1.9	1.40
2.1	0.35
2.3	1.50
2.5	0.80

72. Table P72, $m = 2$

TABLE P72

t	x
2.4	4.15
2.6	2.05
2.8	6.20
3.0	4.30
3.2	5.80


73. Table P73, $m = 3$


TABLE P73


t	x	t	x
0.7	-0.20	1.9	1.02
1.0	-0.54	2.2	0.92
1.3	-0.12	2.5	0.56
1.6	0.38	2.8	0.19

74. [Table P74](#), $m = 4$ **TABLE P74**

t	x	t	x
1.0	1.00	3.0	0.95
1.5	0.82	3.5	1.16
2.0	0.13	4.0	0.85
2.5	0.74	4.5	-0.25

75.  Write a user-defined function $x_i = \text{TrigPoly_mod}(x, m, t_1, t_N, t_i)$ that approximates or interpolates a set of equally spaced data $(t_1, x_1), \dots, (t_N, x_N)$ by a trigonometric polynomial of degree m and returns the interpolated value x_i at a given point t_i . Apply `TrigPoly_mod` to the data in Problem 67 to find the interpolated value at $t = 2$ using a trigonometric polynomial of degree 2.

76.  Apply the user-defined function `TrigPoly_mod` (Problem 75) to the data in Problem 74 to find the interpolated value at $t = 3.25$ using a trigonometric polynomial of degree 4.

 In Problems 77 through 80, for each given set of data, find the interpolating function using the MATLAB function `fft`.

77. [Table P77](#)**TABLE P77**

t	x	t	x
1	1	3	0
1.5	1	3.5	0
2	0	4	1
2.5	0	4.5	1

78. [Table P78](#)**TABLE P78**

t	x	t	x
0.4	0	1.6	0
0.7	1	1.9	-1
1.0	2	2.2	-2
1.3	1	2.5	-1

79. Table P79

TABLE P79

<i>t</i>	<i>x</i>	<i>t</i>	<i>x</i>
1.0	5.024	1.8	0.543
1.1	5.536	1.9	0.510
1.2	3.023	1.0	0.702
1.3	1.505	2.1	0.189
1.4	1.559	2.2	0.176
1.5	1.021	2.3	-0.096
1.6	0.965	2.4	-1.112
1.7	0.998	2.5	0.465

80. Table P80

TABLE P80

<i>t</i>	<i>x</i>	<i>t</i>	<i>x</i>
0.0	4.001	0.8	0.102
0.1	3.902	0.9	0.251
0.2	1.163	1.0	0.229
0.3	0.997	1.1	0.143
0.4	0.654	1.2	0.054
0.5	0.803	1.3	0.001
0.6	0.407	1.4	-0.583
0.7	0.706	1.5	-0.817

6

Numerical Differentiation and Integration

Numerical methods to find estimates for derivatives and definite integrals are presented and discussed in this chapter. Many engineering applications involve rates of change of quantities with respect to variables such as time. For example, linear damping force is directly proportional to velocity, which is the rate of change of displacement with respect to time. Other applications may involve definite integrals. For example, the voltage across a capacitor at any specified time is proportional to the integral of the current taken from an initial time to that specified time.

6.1 Numerical Differentiation

Numerical differentiation is desirable in various situations. Sometimes the analytical expression of the function to be differentiated is known but analytical differentiation proves to be either very difficult or even impossible. In that case, the function is discretized to generate several points (values), which are subsequently used by a numerical method to approximate the derivative of the function at any of the generated points. Often, however, data are available only in the form of a discrete set of points. These points may be recorded data from experimental measurements or generated as a result of some type of numerical computation. In these situations, the derivative can be numerically approximated in one of two ways. One way is to use finite differences, which utilize the data in the neighborhood of the point of interest. In [Figure 6.1a](#), for instance, the derivative at the point x_i is approximated by the slope of the line connecting x_{i-1} and x_{i+1} . The other approach is to fit a suitable, easy to differentiate function into the data ([Chapter 5](#)) and then differentiate the analytical expression of the function and evaluate at the point of interest; see [Figure 6.1b](#).

6.2 Finite-Difference Formulas for Numerical Differentiation

Finite-difference formulas are used to approximate the derivative at a point by using the values at the neighboring points. These formulas can be derived to approximate derivatives of different orders at a specified point by using the Taylor series expansion. In this section, we present the derivation for finite-difference formulas to approximate first and second derivatives at a point, but those for the third and fourth derivatives will be provided without derivation.

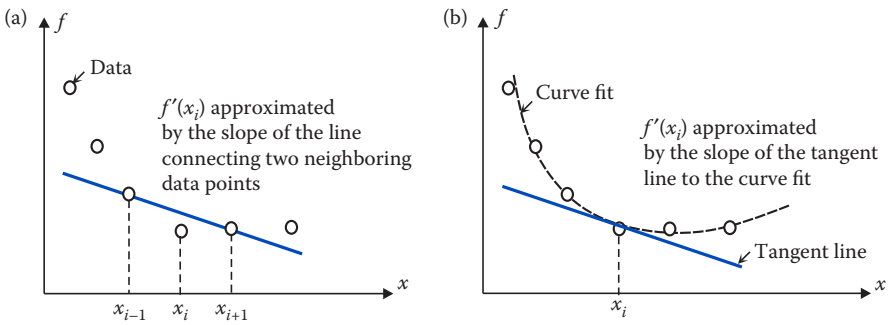


FIGURE 6.1

Approximating $f'(x_i)$ using (a) finite difference and (b) the curve fit.

6.2.1 Finite-Difference Formulas for the First Derivative

There are several methods to approximate the first derivative at a point using the values at two or more of its neighboring points. These points can be chosen to the left, to the right, or on both sides of the point at which the first derivative is to be approximated.

6.2.1.1 Two-Point Backward Difference Formula

The value of $f(x_{i-1})$ can be approximated by a Taylor series expansion at x_i . Letting $h = x_i - x_{i-1}$, we have

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

Retaining the linear terms only, yields

$$f(x_{i-1}) = f(x_i) - hf'(x_i) + \underbrace{\left| \frac{1}{2!}h^2f''(\xi) \right|}_{\text{Remainder}}$$

where $x_{i-1} \leq \xi \leq x_i$. Solving for $f'(x_i)$, we find

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h} + \underbrace{\frac{1}{2!}hf''(\xi)}_{\text{Truncation error}} \quad (6.1)$$

Approximating the first derivative can be done by neglecting the second term on the right side, which produces a truncation error. Since this is proportional to h , we say the truncation error is of the order of h and express it as $O(h)$,

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h} + O(h) \quad (6.2)$$

Note that in many applications, we only have access to a set of data (x_i, y_i) and no function $f(x)$ is available. In those cases, Equation 6.2 is simply replaced with

$$y'_i = \frac{y_i - y_{i-1}}{h} + O(h)$$

A similar tactic is employed for all formulas derived in the remainder of this section. The actual value of the truncation error is not available because the value of ξ in Equation 6.1 is not exactly known. However, $O(h)$ signifies that the error gets smaller as h gets smaller.

6.2.1.2 Two-Point Forward Difference Formula

The value of $f(x_{i+1})$ can be approximated by a Taylor series expansion at x_i . Letting $h = x_{i+1} - x_i$,

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

Retaining the linear terms only,

$$f(x_{i+1}) = f(x_i) + hf'(x_i) + \underbrace{\left| \frac{1}{2!}h^2f''(\xi) \right|}_{\text{Remainder}}$$

where $x_i \leq \xi \leq x_{i+1}$. Solving for $f'(x_i)$, we find

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{1}{2!}hf''(\xi) \quad (6.3)$$

The first term on the right side of Equation 6.3 provides an approximation for the first derivative, while the neglected second term is of the order of h so that

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} + O(h) \quad (6.4)$$

6.2.1.3 Two-Point Central Difference Formula

To derive the central difference formula, we retain up to the quadratic term in the Taylor series. Therefore,

$$f(x_{i-1}) = f(x_i) - hf'(x_i) + \frac{1}{2!}h^2f''(x_i) - \frac{1}{3!}h^3f'''(\xi) \quad , \quad x_{i-1} \leq \xi \leq x_i$$

and

$$f(x_{i+1}) = f(x_i) + hf'(x_i) + \frac{1}{2!}h^2f''(x_i) + \frac{1}{3!}h^3f'''(\eta) \quad , \quad x_i \leq \eta \leq x_{i+1}$$

Subtracting the first equation from the second, we find

$$f(x_{i+1}) - f(x_{i-1}) = 2hf'(x_i) + \frac{1}{3!}h^3[f'''(\eta) + f'''(\xi)]$$

Solving for $f'(x_i)$ and proceeding as before,

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1}))}{2h} + O(h^2) \quad (6.5)$$

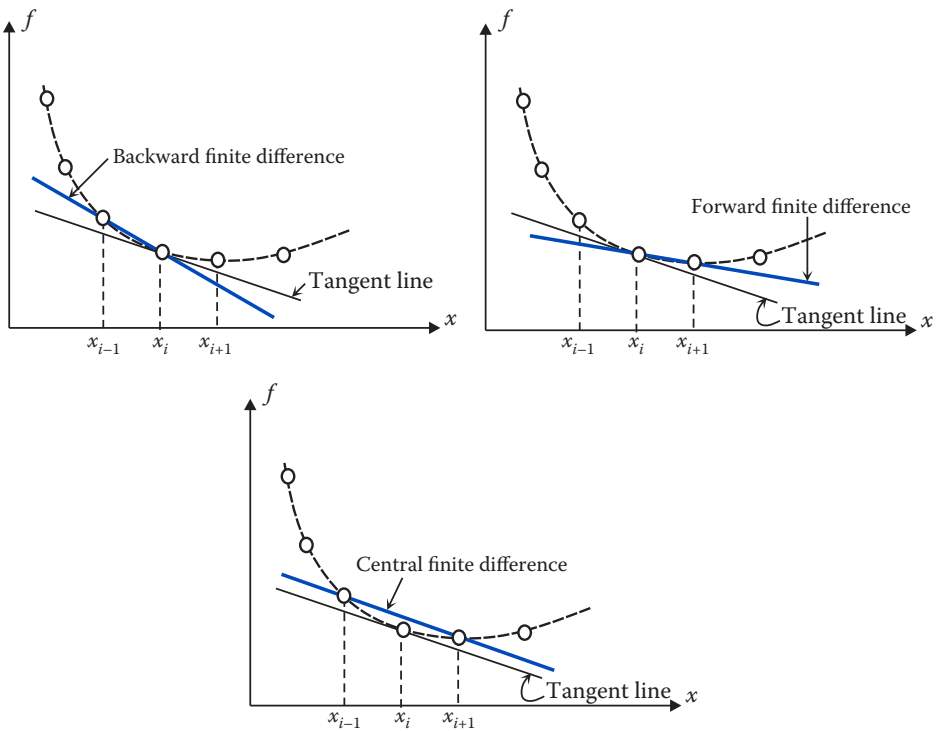


FIGURE 6.2
Two-point finite differences to approximate the first derivative.

Equation 6.5 reveals that the central difference formula provides a *better accuracy* than the backward and forward difference formulas. **Figure 6.2** supports this observation.

Consider a set of data with x_1, x_2, \dots, x_n . Since the two-point backward difference formula uses x_i and the point to its left, x_{i-1} , it cannot be applied at the first data point x_1 . But it can be used to approximate the first derivative at all interior points, as well as the last point x_n , with a truncation error of $O(h)$. The two-point forward difference formula cannot be applied at the last point x_n , but it can be used to approximate the first derivative at the first point x_1 and all the interior points with a truncation error of $O(h)$. The central difference formula approximates the first derivative at the interior points with a truncation error of $O(h^2)$ but cannot be applied at the first and last points. The central difference formula is therefore the preferred choice since it gives better accuracy, but cannot be used at the endpoints. This means the approximation of the first derivative at the interior points has an error of $O(h^2)$, while those at the endpoints come with $O(h)$. In order to have compatible accuracy, it is desired that the approximations at the endpoints also come with $O(h^2)$. These are provided by three-point difference formulas.

6.2.1.4 Three-Point Backward Difference Formula

We first approximate the value of $f(x_{i-1})$ by a Taylor series expansion at x_i ,

$$f(x_{i-1}) = f(x_i) - hf'(x_i) + \frac{1}{2!}h^2 f''(x_i) - \frac{1}{3!}h^3 f'''(\xi) \quad , \quad x_{i-1} \leq \xi \leq x_i$$

We also approximate the value of $f(x_{i-2})$ by a Taylor series expansion at x_i ,

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

Multiplying the first equation by 4 and subtracting the result from the second equation, yields

$$f(x_{i-2}) - 4f(x_{i-1}) = -3f(x_i) + 2hf'(x_i) + \frac{4}{3!}h^3 f'''(\xi) - \frac{8}{3!}h^3 f'''(\eta)$$

Solving for $f'(x_i)$, we arrive at

$$f'(x_i) = \frac{f(x_{i-2}) - 4f(x_{i-1}) + 3f(x_i)}{2h} - \frac{1}{3}h^2 f'''(\xi) + \frac{2}{3}h^2 f'''(\eta)$$

Then, $f'(x_i)$ can be approximated by neglecting the last two terms, which introduces a truncation error of the order of h^2 , that is,

$$f'(x_i) = \frac{f(x_{i-2}) - 4f(x_{i-1}) + 3f(x_i)}{2h} + O(h^2) \quad (6.6)$$

Therefore, the three-point backward difference formula approximates the first derivative at x_i by using the values at the points x_i , x_{i-1} , and x_{i-2} .

6.2.1.5 Three-Point Forward Difference Formula

The three-point forward difference formula approximates the first derivative at x_i by using the values at the points x_i , x_{i+1} , and x_{i+2} . The derivation is similar to that presented for the backward difference, except that the values of $f(x_{i+1})$ and $f(x_{i+2})$ are now considered as Taylor series expanded at x_i . This ultimately leads to

$$f'(x_i) = \frac{-3f(x_i) + 4f(x_{i+1}) - f(x_{i+2})}{2h} + O(h^2) \quad (6.7)$$

EXAMPLE 6.1: FINITE-DIFFERENCE FORMULAS FOR THE FIRST DERIVATIVE

Consider the data generated by the function $f(x) = e^{-x} \sin(x/2)$ at $x = 1.2, 1.4, 1.6, 1.8$. Approximate $f'(1.4)$ using

- Two-point backward difference formula
- Two-point forward difference formula
- Two-point central difference formula
- Three-point forward difference formula

Find the percentage relative error in each case.

Solution

Since $f'(x) = e^{-x} \left[\frac{1}{2} \cos(x/2) - \sin(x/2) \right]$, the actual value is $f'(1.4) = -0.0646$. The approximate first derivative is calculated via the four difference formulas listed above and are summarized in [Table 6.1](#). As expected, the two-point central difference and three-point forward difference formulas provide better accuracy than the other two techniques.

TABLE 6.1

Summary of Calculations in Example 6.1

Difference Formula	Approximate $f'(1.4)$	% Relative Error
Two-point backward	$\frac{f(1.4) - f(1.2)}{0.2} = -0.0560$	13.22
Two-point forward	$\frac{f(1.6) - f(1.4)}{0.2} = -0.0702$	8.66
Two-point central	$\frac{f(1.6) - f(1.2)}{2(0.2)} = -0.0631$	2.28
Three-point forward	$\frac{-3f(1.4) + 4f(1.6) - f(1.8)}{2(0.2)} = -0.0669$	3.56

6.2.2 Finite-Difference Formulas for the Second Derivative

The second derivative at x_i can also be approximated by finite difference formulas. These formulas are derived in a similar manner as those for the first derivative. Below, we present three-point backward and forward difference, as well as three-point central difference formulas for approximating the second derivative.

6.2.2.1 Three-Point Backward Difference Formula

The values of $f(x_{i-1})$ and $f(x_{i-2})$ are first approximated by Taylor series expansions about x_i

$$f(x_{i-1}) = f(x_i) - hf'(x_i) + \frac{1}{2!}h^2 f''(x_i) - \frac{1}{3!}h^3 f'''(\xi) \quad , \quad x_{i-1} \leq \xi \leq x_i$$

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

Multiplying the first equation by 2 and subtracting from the second equation results in

$$f(x_{i-2}) - 2f(x_{i-1}) = -f(x_i) + h^2 f''(x_i) - \frac{4}{3}h^3 f'''(\eta) + \frac{1}{3}h^3 f'''(\xi)$$

Proceeding as before, we find

$$f''(x_i) = \frac{f(x_{i-2}) - 2f(x_{i-1}) + f(x_i)}{h^2} + O(h) \quad (6.8)$$

6.2.2.2 Three-Point Forward Difference Formula

The values of $f(x_{i+1})$ and $f(x_{i+2})$ are first approximated by Taylor series expansions about x_i

$$f(x_{i+1}) = f(x_i) + hf'(x_i) + \frac{1}{2!}h^2 f''(x_i) + \frac{1}{3!}h^3 f'''(\xi) \quad , \quad x_i \leq \xi \leq x_{i+1}$$

$$f(x_{i+2}) = f(x_i) + (2h)f'(x_i) + \frac{1}{2!}(2h)^2 f''(x_i) + \frac{1}{3!}(2h)^3 f'''(\eta) \quad , \quad x_i \leq \eta \leq x_{i+2}$$

Multiplying the first equation by 2 and subtracting from the second equation results in

$$f(x_{i+2}) - 2f(x_{i+1}) = -f(x_i) + h^2 f''(x_i) + \frac{4}{3}h^3 f'''(\eta) - \frac{1}{3}h^3 f'''(\xi)$$

Therefore,

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + O(h) \quad (6.9)$$

6.2.2.3 Three-Point Central Difference Formula

Expanding $f(x_{i-1})$ and $f(x_{i+1})$ in Taylor series about x_i and retaining up to the third derivative terms, we find

$$f(x_{i-1}) = f(x_i) - hf'(x_i) + \frac{1}{2!}h^2 f''(x_i) - \frac{1}{3!}h^3 f'''(x_i) + \frac{1}{4!}h^4 f^{(4)}(\xi) , \quad x_{i-1} \leq \xi \leq x_i$$

$$f(x_{i+1}) = f(x_i) + hf'(x_i) + \frac{1}{2!}h^2 f''(x_i) + \frac{1}{3!}h^3 f'''(x_i) + \frac{1}{4!}h^4 f^{(4)}(\eta) , \quad x_i \leq \eta \leq x_{i+1}$$

Adding the two equations and proceeding as always, we have

$$f''(x_i) = \frac{f(x_{i-1}) - 2f(x_i) + f(x_{i+1}))}{h^2} + O(h^2) \quad (6.10)$$

Therefore, in approximating the second derivative, the three-point central difference formula has a truncation error of $O(h^2)$ compared to $O(h)$ for the three-point backward and forward difference formulas.

EXAMPLE 6.2: FINITE-DIFFERENCE FORMULAS FOR THE SECOND DERIVATIVE

Consider the data in Example 6.1. Approximate $f''(1.4)$ using

- Three-point backward difference formula ($h = 0.2$)
- Three-point forward difference formula ($h = 0.2$)
- Three-point central difference formula ($h = 0.2$)
- Three-point central difference formula ($h = 0.1$)

Find the percentage relative error in each case.

Solution

Since $f''(x) = e^{-x} \left[\frac{3}{4} \sin(x/2) - \cos(x/2) \right]$, the actual value is $f''(1.4) = -0.0695$. The numerical results are summarized in Table 6.2, where it is readily seen that the three-point central difference formula produces the most accurate estimate. It is also observed that reducing the spacing size significantly improves the accuracy.

TABLE 6.2

Summary of Calculations in Example 6.2

Difference Formula	Approximate $f''(1.4)$	% Relative Error
Three-point backward $h = 0.2$	$\frac{f(1) - 2f(1.2) + f(1.4)}{(0.2)^2} = -0.1225$	76.4
Three-point forward $h = 0.2$	$\frac{f(1.4) - 2f(1.6) + f(1.8)}{(0.2)^2} = -0.0330$	52.6
Three-point central $h = 0.2$	$\frac{f(1.2) - 2f(1.4) + f(1.6)}{(0.2)^2} = -0.0706$	1.69
Three-point central $h = 0.1$	$\frac{f(1.3) - 2f(1.4) + f(1.5)}{(0.1)^2} = -0.0698$	0.42

6.2.2.4 Summary of Finite-Difference Formulas for First to Fourth Derivatives

Table 6.3 lists the difference formulas presented earlier, as well as additional formulas for the first and second derivatives. It also includes formulas that can similarly be derived for the third and fourth derivatives at a point x_i .

6.2.3 Estimate Improvement: Richardson's Extrapolation

Derivative estimates using finite differences can clearly be improved by either reducing the spacing size or using a higher-order difference formula which involves more points. A third method is to use Richardson's extrapolation, which combines two derivative approximations to obtain a more accurate estimate. The idea is best understood through a specific example.

Consider the approximation of the first derivative using the two-point central difference formula. We will repeat some of the analysis done earlier, but show more terms in Taylor series expansions for our purpose.

$$f(x_{i-1}) = f(x_i) - hf'(x_i) + \frac{1}{2!}h^2 f''(x_i) - \frac{1}{3!}h^3 f'''(x_i) + \frac{1}{4!}h^4 f^{(4)}(x_i) - \frac{1}{5!}h^5 f^{(5)}(\xi) \quad , \quad x_{i-1} \leq \xi \leq x_i$$

and

$$f(x_{i+1}) = f(x_i) + hf'(x_i) + \frac{1}{2!}h^2 f''(x_i) + \frac{1}{3!}h^3 f'''(x_i) + \frac{1}{4!}h^4 f^{(4)}(x_i) + \frac{1}{5!}h^5 f^{(5)}(\eta) \quad , \quad x_i \leq \eta \leq x_{i+1}$$

Subtracting the first equation from the second, and solving for $f'(x_i)$, yields

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1}))}{2h} - \frac{1}{3!}h^2 f'''(x_i) + O(h^4) \tag{6.11}$$

We next repeat this process with step size $\frac{1}{2}h$. In the meantime, we introduce the notations $f(x_{i-1/2}) = f(x_i - \frac{1}{2}h)$ and $f(x_{i+1/2}) = f(x_i + \frac{1}{2}h)$. Then, it can be shown that

$$f'(x_i) = \frac{f(x_{i+1/2}) - f(x_{i-1/2}))}{2(\frac{1}{2}h)} - \frac{1}{3!}\left(\frac{1}{2}h\right)^2 f'''(x_i) + O(h^4) \tag{6.12}$$

Multiply Equation 6.12 by 4 and subtract Equation 6.11 from the result to obtain

$$f'(x_i) = \frac{4}{3} \underbrace{\frac{f(x_{i+1/2}) - f(x_{i-1/2}))}{h}}_{\substack{\text{2-pt central diff. formula} \\ \text{with } h/2 \text{ and error } O(h^2)}} - \frac{1}{3} \underbrace{\frac{f(x_{i+1}) - f(x_{i-1}))}{2h}}_{\substack{\text{2-pt central diff. formula} \\ \text{with } h \text{ and error } O(h^2)}} + O(h^4) \tag{6.13}$$

Therefore, two approximations provided by the two-point central difference formula, one with spacing h and the other $\frac{1}{2}h$, each with error $O(h^2)$, are combined to obtain a more accurate estimate of the first derivative with error $O(h^4)$.

Equation 6.13 can be expressed in a general form as

$$D = \frac{4}{3}D_{h/2} - \frac{1}{3}D_h + O(h^4) \tag{6.14}$$

TABLE 6.3

Summary of Finite Difference Formulas for First, Second, Third, and Fourth Derivatives

Difference Formula	First Derivative	Truncation error
Two-point backward	$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h}$	$O(h)$
Two-point forward	$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h}$	$O(h)$
Two-point central	$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h}$	$O(h^2)$
Three-point backward	$f'(x_i) = \frac{f(x_{i-2}) - 4f(x_{i-1}) + 3f(x_i)}{2h}$	$O(h^2)$
Three-point forward	$f'(x_i) = \frac{-3f(x_i) + 4f(x_{i+1}) - f(x_{i+2})}{2h}$	$O(h^2)$
Four-point central	$f'(x_i) = \frac{f(x_{i-2}) - 8f(x_{i-1}) + 8f(x_{i+1}) - f(x_{i+2})}{12h}$	$O(h^4)$
Difference Formula	Second Derivative	Truncation Error
Three-point backward	$f''(x_i) = \frac{f(x_{i-2}) - 2f(x_{i-1}) + f(x_i)}{h^2}$	$O(h)$
Three-point forward	$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2}$	$O(h)$
Three-point central	$f''(x_i) = \frac{f(x_{i-1}) - 2f(x_i) + f(x_{i+1})}{h^2}$	$O(h^2)$
Four-point backward	$f''(x_i) = \frac{-f(x_{i-3}) + 4f(x_{i-2}) - 5f(x_{i-1}) + 2f(x_i)}{h^2}$	$O(h^2)$
Four-point forward	$f''(x_i) = \frac{2f(x_i) - 5f(x_{i+1}) + 4f(x_{i+2}) - f(x_{i+3})}{h^2}$	$O(h^2)$
Five-point central	$f''(x_i) = \frac{-f(x_{i-2}) + 16f(x_{i-1}) - 30f(x_i) + 16f(x_{i+1}) - f(x_{i+2})}{12h^2}$	$O(h^4)$
Difference Formula	Third Derivative	Truncation Error
Four-point backward	$f'''(x_i) = \frac{-f(x_{i-3}) + 3f(x_{i-2}) - 3f(x_{i-1}) + f(x_i)}{h^3}$	$O(h)$
Four-point forward	$f'''(x_i) = \frac{-f(x_i) + 3f(x_{i+1}) - 3f(x_{i+2}) + f(x_{i+3})}{h^3}$	$O(h)$
Four-point central	$f'''(x_i) = \frac{-f(x_{i-2}) + 2f(x_{i-1}) - 2f(x_{i+1}) + f(x_{i+2})}{2h^3}$	$O(h^2)$
Five-point backward	$f'''(x_i) = \frac{3f(x_{i-4}) - 14f(x_{i-3}) + 24f(x_{i-2}) - 18f(x_{i-1}) + 5f(x_i)}{2h^3}$	$O(h^2)$
Five-point forward	$f'''(x_i) = \frac{-5f(x_i) + 18f(x_{i+1}) - 24f(x_{i+2}) + 14f(x_{i+3}) - 3f(x_{i+4})}{2h^3}$	$O(h^2)$
Six-point central	$f'''(x_i) = \frac{f(x_{i-3}) - 8f(x_{i-2}) + 13f(x_{i-1}) - 13f(x_{i+1}) + 8f(x_{i+2}) - f(x_{i+3})}{8h^3}$	$O(h^4)$

(Continued)

TABLE 6.3 (Continued)

Summary of Finite Difference Formulas for First, Second, Third, and Fourth Derivatives

Difference Formula	Fourth Derivative	Truncation error
Five-point backward	$f^{(4)}(x_i) = \frac{f(x_{i-4}) - 4f(x_{i-3}) + 6f(x_{i-2}) - 4f(x_{i-1}) + f(x_i)}{h^4}$	$O(h)$
Five-point forward	$f^{(4)}(x_i) = \frac{f(x_i) - 4f(x_{i+1}) + 6f(x_{i+2}) - 4f(x_{i+3}) + f(x_{i+4})}{h^4}$	$O(h)$
Five-point central	$f^{(4)}(x_i) = \frac{f(x_{i-2}) - 4f(x_{i-1}) + 6f(x_i) - 4f(x_{i+1}) + f(x_{i+2})}{h^4}$	$O(h^2)$
Six-point backward	$f^{(4)}(x_i) = \frac{-2f(x_{i-5}) + 11f(x_{i-4}) - 24f(x_{i-3}) + 26f(x_{i-2}) - 14f(x_{i-1}) + 3f(x_i)}{h^4}$	$O(h^2)$
Six-point forward	$f^{(4)}(x_i) = \frac{3f(x_i) - 14f(x_{i+1}) + 26f(x_{i+2}) - 24f(x_{i+3}) + 11f(x_{i+4}) - 2f(x_{i+5})}{h^4}$	$O(h^2)$
Seven-point central	$f^{(4)}(x_i) = \frac{f(x_{i-3}) + 12f(x_{i-2}) - 39f(x_{i-1}) + 56f(x_i) + 39f(x_{i+1}) + 12f(x_{i+2}) - f(x_{i+3})}{6h^4}$	$O(h^4)$

where

D = Value of the derivative

D_h = A function that approximates the derivative using h and has an error of $O(h^2)$

$D_{h/2}$ = A function that approximates the derivative using $\frac{1}{2}h$ and has an error of $O(h^2)$

Note that Equation 6.14 can be used in connection with any difference formula that has an error of $O(h^2)$. Also note that the coefficients in Equation 6.14 add up to 1, hence act as *weights* attached to each estimate. With increasing accuracy, they place greater weight on the better estimate. For instance, using spacing size $\frac{1}{2}h$ generates a better estimate than the one using h , and consequently $D_{h/2}$ has a larger weight attached to it than D_h does.

EXAMPLE 6.3: RICHARDSON’S EXTRAPOLATION

Consider the data in Example 6.2. We approximated $f''(1.4)$ using the three-point central difference formula, which has an error of $O(h^2)$. Using $h = 0.2$, we found the estimate to be -0.0706 . Using $h = 0.1$, the estimate was -0.0698 . Therefore, $D_h = -0.0706$ and $D_{h/2} = -0.0698$. By Equation 6.14,

$$D \cong \frac{4}{3}(-0.0698) - \frac{1}{3}(-0.0706) = -0.0695$$

which agrees with the actual value to four decimal places and is a superior estimate to the first two.

Richardson’s extrapolation can also be used in connection with estimates that have higher-order errors. In particular, it can combine two estimates, each with error $O(h^4)$, to compute a new, more accurate estimate with error $O(h^6)$

$$D = \frac{16}{15}D_{h/2} - \frac{1}{15}D_h + O(h^6) \tag{6.15}$$

where

D = Value of the derivative

D_h = A function that approximates the derivative using h and has an error of $O(h^4)$

$D_{h/2}$ = A function that approximates the derivative using $\frac{1}{2}h$ and has an error of $O(h^4)$.

Once again, as mentioned before, the coefficients add up to 1 and act as weights attached to the two estimates, with greater weight placed on the better estimate.

6.2.4 Richardson's Extrapolation for Discrete Sets of Data

Applications of extrapolation formulas given in Equations 6.14 and 6.15 are rather straightforward when a function $f(x)$ generates the data, as observed in Example 6.3. However, in the absence of $f(x)$, we can no longer change the value of the step size from h to $\frac{1}{2}h$ and analyze the generated data. For discrete data, D_h is calculated using a set comprised of every other data in the original set, while $D_{h/2}$ is calculated using the entire original set.

6.2.5 Derivative Estimates for Non-Evenly Spaced Data

The finite-difference formulas to approximate derivatives of various orders require that the data be equally spaced. Also, Richardson's extrapolation is applicable only to evenly spaced data and it computes better estimates by sequentially reducing the spacing by half. These techniques are appropriate if the data are equally spaced or if the data are generated by uniform discretization of a known function, such as that in Examples 6.1 and 6.2.

Empirical data—such as data resulting from experimental measurements—on the other hand, are sometimes not evenly spaced. For these situations, one possible way to approximate the derivative is as follows: (1) consider a set of three consecutive data points that contains the point at which the derivative is to be estimated, (2) fit a second-degree Lagrange or Newton interpolating polynomial (Chapter 5) to the set, and (3) differentiate the polynomial and evaluate at the point of interest. The derivative estimate obtained in this manner has the same accuracy as that offered by the central difference formula, and exactly matches it for the case of equally spaced data.

EXAMPLE 6.4: NON-EVENLY SPACED DATA

For the data in Table 6.4, approximate the first derivative at $x = 0.7$ using the data at 0.3, 0.8, and 1.1.

Solution

The data are not evenly spaced. We will consider the set of three consecutive points 0.3, 0.8, and 1.1, which includes the point of interest $x = 0.7$, and fit a second-degree Lagrange interpolating polynomial to the set. Letting $x_1 = 0.3$, $x_2 = 0.8$, and $x_3 = 1.1$, we find

$$\begin{aligned}
 p_2(x) &= \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} (0.8228) + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} (0.4670) + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} (0.2617) \\
 &= 0.0341x^2 - 0.7491x + 1.0445
 \end{aligned}$$

Differentiation yields $p_2'(x) = 0.0682x - 0.7491$ so that $p_2'(0.7) = -0.7014$.

TABLE 6.4

Data in Example 6.4

x	y
0	1
0.3	0.8228
0.8	0.4670
1.1	0.2617
1.3	0.1396

6.2.6 MATLAB Built-In Functions `diff` and `polyder`

The MATLAB built-in function `diff` can be used to estimate derivatives for both cases of equally spaced and not equally spaced data. A brief description of `diff` is given as

`diff(X)` calculates differences between adjacent elements of X .

If X is a vector, then `diff(X)` returns a vector, one element shorter than X , of differences between adjacent elements

$$[X(2)-X(1) \quad X(3)-X(2) \quad \dots \quad X(n)-X(n-1)]$$

`diff(X,n)` applies `diff` recursively n times, resulting in the n th difference. Thus, `diff(X,2)` is the same as `diff(diff(X))`.

Equally spaced data. Consider a set of equally spaced data $(x_1, y_1), \dots, (x_n, y_n)$, where $x_{i+1} - x_i = h$ ($i = 1, \dots, n - 1$). Then, by the description of `diff`, the command `diff(y) ./ h` returns the $(n - 1)$ -dimensional vector

$$\left[\frac{y_2 - y_1}{h} \quad \dots \quad \frac{y_n - y_{n-1}}{h} \right]$$

The first component is the first-derivative estimate at x_1 using the forward-difference formula; see Equation 6.4. Similarly, the second component is the derivative estimate at x_2 . The last entry is the derivative estimate at x_{n-1} . As an example consider $f(x) = e^{-x} \sin(x/2)$, $x = 1.2, 1.4, 1.6, 1.8$, of Example 6.1. We find an estimate for $f'(1.4)$ as follows:

```
>> h = 0.2; x = 1.2:h:1.8;
>> y = [0.1701 0.1589 0.1448 0.1295]; % Values of f at the discrete x values
>> y_prime = diff(y) ./ h

y_prime =

    -0.0560    -0.0702    -0.0767
```

Since 1.4 is the second point in the data, it is labeled x_2 . Which means an estimate for $f'(1.4)$ is provided by the second component of the output `y_prime`. That is, $f'(1.4) \cong -0.0702$. This agrees with the earlier numerical results in Table 6.1.

Non-equally spaced data. Consider a set of non-evenly spaced data $(x_1, y_1), \dots, (x_n, y_n)$. Then, by the description of `diff`, the command `diff(y) ./ diff(x)` returns the $(n - 1)$ -dimensional vector

$$\left[\frac{y_2 - y_1}{x_2 - x_1} \quad \dots \quad \frac{y_n - y_{n-1}}{x_n - x_{n-1}} \right]$$

The first component is the first-derivative estimate at x_1 using the forward-difference formula, the second one is the derivative estimate at x_2 , while the last entry is the derivative estimate at x_{n-1} .

As mentioned in the description of `diff` above, `diff(y,2)` is the same as `diff(diff(y))`. So, if $y = [y_1 \ \dots \ y_n]$, then `diff(y)` returns

$$\left[y_2 - y_1 \quad y_3 - y_2 \quad \dots \quad y_n - y_{n-1} \right]_{(n-1)\text{-dim}}$$

and `diff(y,2)` returns

$$\left[(y_3 - y_2) - (y_2 - y_1) \quad (y_4 - y_3) - (y_3 - y_2) \quad \dots \quad (y_n - y_{n-1}) - (y_{n-1} - y_{n-2}) \right]_{(n-2)\text{-dim}}$$

which simplifies to

$$\left[y_3 - 2y_2 + y_1 \quad y_4 - 2y_3 + y_2 \quad \dots \quad y_n - 2y_{n-1} + y_{n-2} \right]$$

The first component is the numerator in the three-point forward difference formula for estimating the second derivative at x_1 ; see Equation 6.9. Similarly, the remaining components agree with the numerator of Equation 6.9 at x_2, \dots, x_{n-2} . Therefore, for an equally spaced data $(x_1, y_1), \dots, (x_n, y_n)$, an estimate of the second derivative at x_1, x_2, \dots, x_{n-2} is provided by

$$\text{diff}(y,2) ./ h^2$$

The MATLAB built-in function `polyder` finds the derivative of a polynomial:

`polyder` Differentiate polynomial.

`polyder(P)` returns the derivative of the polynomial whose coefficients are the elements of vector `P`.

`polyder(A,B)` returns the derivative of polynomial `A*B`.

`[Q,D] = polyder(B,A)` returns the derivative of the polynomial ratio `B/A`, represented as `Q/D`.

For example, the derivative of a polynomial such as $2x^3 - x + 3$ is calculated as follows:

```
>> P = [2 0 -1 3];
>> polyder(P)
```

```
ans =
```

```
6     0    -1
```

The output corresponds to $6x^2 - 1$.

6.3 Numerical Integration: Newton–Cotes Formulas

We encounter definite integrals in a wide range of applications, generally in the form

$$\int_a^b f(x) dx$$

where $f(x)$ is the integrand and a and b are the limits of integration. The value of this definite integral is the area of the region between the graph of $f(x)$ and the x -axis, bounded by the lines $x = a$ and $x = b$. As an example of a definite integral, consider the relation between the bending moment M and shear force V along the longitudinal axis x of a beam, defined by

$$M_2 - M_1 = \int_{x_1}^{x_2} V(x) dx$$

where M_2 is the bending moment at position x_2 and M_1 is the bending moment at x_1 . In this case, the integrand is shear force $V(x)$ and the limits of integration are x_1 and x_2 .

The integrand may be given analytically or as a set of discrete points. Numerical integration is used when the integrand is given as a set of data or, the integrand is an analytical function, but the antiderivative is not easily found. In order to carry out numerical integration, discrete values of the integrand are needed. This means that even if the integrand is an analytical function, it must be discretized and the discrete values will be used in the calculations.

6.3.1 Newton–Cotes Formulas

Newton–Cotes formulas provide the most commonly used integration techniques and are divided into two categories: closed form and open form. In closed-form schemes, the data points at the endpoints of the interval are used in calculations; the trapezoidal and Simpson’s rules are closed Newton–Cotes formulas. In open-form methods, limits of integration extend beyond the range of the discrete data; the rectangular rule and the Gaussian quadrature (Section 6.4) are open Newton–Cotes formulas.

The main idea behind Newton–Cotes formulas is to replace the complicated integrand or data with an easy-to-integrate function, usually a polynomial. If the integrand is an analytical function, it is first discretized, and then the polynomial that interpolates the discretized set is found and integrated. If the integrand is a set of data, the interpolating polynomial is found and integrated.

6.3.2 Rectangular Rule

In the rectangular rule, the definite integral $\int_a^b f(x) dx$ is approximated by the area of a rectangle. This rectangle may be built using the left endpoint, the right endpoint, or the midpoint of the interval $[a, b]$ (Figure 6.3). *The one that uses the midpoint is sometimes called the midpoint method and is only applicable when the integrand is an analytical expression.* All three cases are Newton–Cotes formulas, where the integrand is replaced with a horizontal line (constant), that is, a zero-degree polynomial. But it is evident by Figure 6.3 that the error of approximation can be quite large depending on the nature of the integrand. The accuracy can be improved considerably by using the composite rectangular rule.

6.3.2.1 Composite Rectangular Rule

In applying the composite rectangular rule, the interval $[a, b]$ is divided into n subintervals defined by $n + 1$ points labeled $a = x_1, x_2, \dots, x_n, x_{n+1} = b$. The subintervals can generally have different widths so that longer intervals may be chosen for regions where the integrand exhibits slow variations and shorter intervals where the integrand experiences

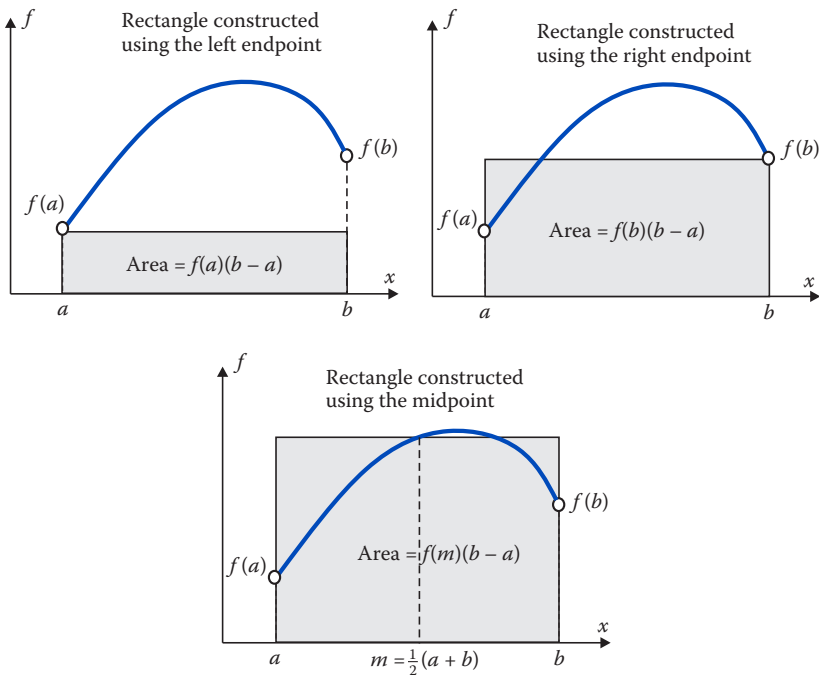


FIGURE 6.3
Rectangular rule.

rapid changes. In most of the results presented here, however, the data are assumed equally spaced. Over each subinterval $[x_i, x_{i+1}]$, the integral is approximated by the area of a rectangle. These rectangles are constructed using the left endpoint, the right endpoint, or the midpoint as described earlier (Figure 6.4). Adding the areas of rectangles yields the approximate value of the definite integral $\int_a^b f(x) dx$.

Composite Rectangular Rule (Using Left Endpoint)

$$\int_a^b f(x) dx = \sum_{i=1}^n \{f(x_i)(x_{i+1} - x_i)\} + O(h) \quad \begin{array}{l} \text{For equally spaced data} \\ = \\ h = (b-a)/n \end{array} \quad h \sum_{i=1}^n f(x_i) + O(h) \quad (6.16)$$

Composite Rectangular Rule (Using Right Endpoint)

$$\int_a^b f(x) dx = \sum_{i=2}^{n+1} \{f(x_i)(x_i - x_{i-1})\} + O(h) \quad \begin{array}{l} \text{For equally spaced data} \\ = \\ h = (b-a)/n \end{array} \quad h \sum_{i=2}^{n+1} f(x_i) + O(h) \quad (6.17)$$

Composite Rectangular Rule (Using Midpoint)

$$\int_a^b f(x) dx = \sum_{i=1}^n \{f(m_i)(x_{i+1} - x_i)\} + O(h^2) \quad \begin{array}{l} \text{For equally spaced data} \\ = \\ h = (b-a)/n \end{array} \quad h \sum_{i=1}^n f(m_i) + O(h^2), \quad m_i = \frac{1}{2}(x_{i+1} + x_i) \quad (6.18)$$

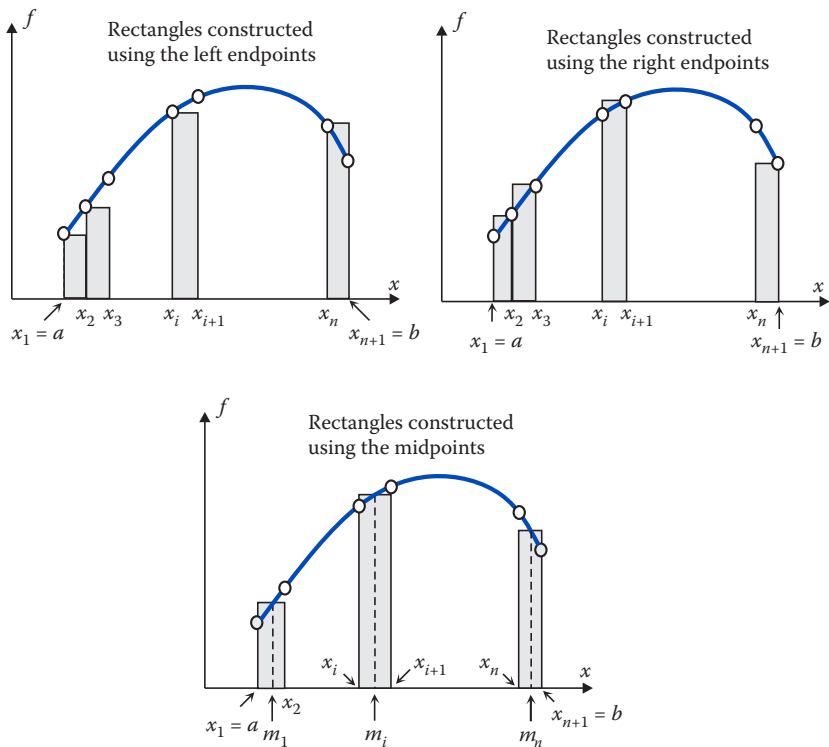


FIGURE 6.4
Composite rectangular rules.

6.3.3 Error Estimate for Composite Rectangular Rule

Equations 6.16 through 6.18 included the error estimates associated with the three composite rectangular rule options. We will elaborate on these here. Consider $\int_a^b f(x) dx$, where the points $a = x_1, x_2, \dots, x_n, x_{n+1} = b$ divide $[a, b]$ into n subintervals of equal length h . Assume that over each interval $[x_i, x_{i+1}]$, the rectangle is constructed using the left endpoint x_i so that it has an area of $hf(x_i)$. The error associated with the integral over each subinterval is

$$E_i = \int_{x_i}^{x_{i+1}} f(x) dx - \underset{\text{Estimate}}{hf(x_i)}$$

Actual value

By Taylor series expansion, we have

$$f(x) = f(x_i) + f'(\xi_i)(x - x_i), \quad x_i \leq \xi_i \leq x_{i+1}$$

Then,

$$E_i = \int_{x_i}^{x_{i+1}} [f(x_i) + f'(\xi_i)(x - x_i)] dx - hf(x_i) \stackrel{\text{Evaluate and simplify}}{=} \frac{1}{2} h^2 f'(\xi_i)$$

This indicates that each error E_i can be made very small by choosing a very small spacing size, that is, $h \ll 1$. The error associated with the entire interval $[a, b]$ is given by

$$E = \sum_{i=1}^n E_i = \sum_{i=1}^n \frac{1}{2} h^2 f''(\xi_i) = \frac{1}{2} h^2 \sum_{i=1}^n f''(\xi_i)$$

An average value for f'' over $[a, b]$ may be estimated by

$$\bar{f}'' \cong \frac{1}{n} \sum_{i=1}^n f''(\xi_i)$$

Consequently,

$$E = \frac{1}{2} h^2 n \bar{f}'' = \frac{1}{2} \left(\frac{b-a}{n} \right) h n \bar{f}'' = \left[\frac{1}{2} (b-a) \bar{f}'' \right] h$$

Since $\frac{1}{2}(b-a)\bar{f}'' = \text{const}$, the error E is of the order of h , written $O(h)$. In summary,

Composite rectangular rule (left endpoint)

$$E = \left[\frac{1}{2} (b-a) \bar{f}'' \right] h = O(h) \quad (6.19)$$

Similarly, for the composite rectangular rule (using right endpoint), $E = O(h)$. Finally, we present without proof:

Composite rectangular rule (midpoint)

$$E = \left[\frac{1}{24} (b-a) \bar{f}''' \right] h^2 = O(h^2) \quad (6.20)$$

where \bar{f}''' is the estimated average value of f''' over $[a, b]$.

EXAMPLE 6.5: COMPOSITE RECTANGULAR RULE

Evaluate the following definite integral using all three composite rectangular rule strategies with $n = 8$:

$$\int_{-1}^1 \frac{1}{x+2} dx$$

Solution

With the limits of integration at $b=1$, $a=-1$, we find the spacing size as $h = (b-a)/n = 2/8 = 0.25$. The nine nodes are thus defined as $x_1 = -1, -0.75, -0.5, \dots, 0.75, 1 = x_9$. Letting $f(x) = 1/(x+2)$, the three integral estimates are found as follows:

Using left endpoint,

$$\int_{-1}^1 f(x) dx \cong h \sum_{i=1}^8 f(x_i) = 0.25[f(-1) + f(-0.75) + \dots + f(0.75)] = 1.1865$$

Using right endpoint,

$$\int_{-1}^1 f(x) dx \cong h \sum_{i=2}^9 f(x_i) = 0.25[f(-0.75) + f(-0.5) + \dots + f(0.75) + f(1)] = 1.0199$$

Using midpoint,

$$\int_{-1}^1 f(x) dx \cong h \sum_{i=1}^8 f(m_i) = 0.25[f(-0.8750) + f(-0.6250) + \dots + f(0.8750)] = 1.0963$$

Noting that the actual value of the integral is $\ln 3 = 1.0986$, the above estimates come with relative errors of 8%, 7.17%, and 0.21%, respectively. As suggested by Equations 6.19 and 6.20, the midpoint method yields the best accuracy.

6.3.4 Trapezoidal Rule

The trapezoidal rule is a Newton–Cotes formula, where the integrand is replaced with a straight line (a first-degree polynomial) connecting the points $(a, f(a))$ and $(b, f(b))$ so that the definite integral $\int_a^b f(x) dx$ is approximated by the area of a trapezoid (Figure 6.5a). The equation of this connecting line is

$$p_1(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a)$$

Therefore,

$$\int_a^b f(x) dx \cong \int_a^b p_1(x) dx = \int_a^b \left\{ f(a) + \frac{f(b) - f(a)}{b - a}(x - a) \right\} dx = \left[f(a)x + \frac{f(b) - f(a)}{b - a} \frac{(x - a)^2}{2} \right]_{x=a}^b$$

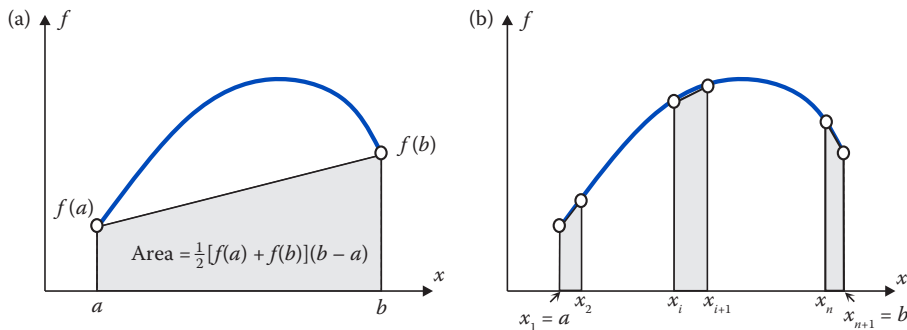


FIGURE 6.5

(a) Trapezoidal rule and (b) composite trapezoidal rule.

Evaluation of this last expression yields

$$\int_a^b f(x) dx \cong \frac{f(a) + f(b)}{2}(b - a) \quad (6.21)$$

The right side is indeed the area of the trapezoid as shown in [Figure 6.5a](#). It is also evident by [Figure 6.5a](#) that the error of approximation can be quite large depending on the nature of the integrand. The accuracy of estimation can be improved significantly by using the composite trapezoidal rule ([Figure 6.5b](#)).

6.3.4.1 Composite Trapezoidal Rule

In the composite rectangular rule, the interval $[a, b]$ is divided into n subintervals defined by $n + 1$ points labeled as $a = x_1, x_2, \dots, x_n, x_{n+1} = b$. As in the case of rectangular rule, the subintervals can have different widths so that longer intervals can be used for regions where the integrand shows slow variations and shorter intervals where the integrand shows rapid changes. In most of the results presented here, however, the data are assumed equally spaced. Over each subinterval $[x_i, x_{i+1}]$, the integral is approximated by the area of a trapezoid. Adding the areas of trapezoids yields the approximate value of the definite integral:

$$\begin{aligned} \int_a^b f(x) dx &\cong \frac{f(x_1) + f(x_2)}{2}(x_2 - x_1) + \frac{f(x_2) + f(x_3)}{2}(x_3 - x_2) + \dots + \frac{f(x_n) + f(x_{n+1})}{2}(x_{n+1} - x_n) \\ &= \sum_{i=1}^n \left\{ \frac{f(x_i) + f(x_{i+1})}{2}(x_{i+1} - x_i) \right\} + O(h^2) \end{aligned} \quad (6.22)$$

For the case of equally spaced data, $x_{i+1} - x_i = h$ ($i = 1, 2, \dots, n$), Equation 6.22 simplifies to

$$\begin{aligned} \int_a^b f(x) dx &= \frac{h}{2} \sum_{i=1}^n \{f(x_i) + f(x_{i+1})\} + O(h^2) \\ &= \frac{h}{2} [f(a) + 2f(x_2) + 2f(x_3) + \dots + 2f(x_n) + f(b)] + O(h^2) \end{aligned} \quad (6.23)$$

6.3.4.2 Error Estimate for Composite Trapezoidal Rule

The error for the composite trapezoidal rule can be shown to be

$$E = \left[-\frac{1}{12}(b - a)\bar{f}'' \right] h^2 = O(h^2) \quad (6.24)$$

where \bar{f}'' is the estimated average value of f'' over $[a, b]$. Therefore, the error $O(h^2)$ is compatible with the midpoint method and superior to the rectangular rule using the endpoints whose error is $O(h)$.

The user-defined function `TrapComp` uses the composite trapezoidal rule to estimate the value of a definite integral.

```
function I = TrapComp(f,a,b,n)
%
% TrapComp estimates the value of the integral of f(x) from a to b
% by using the composite trapezoidal rule applied to n equal-length
% subintervals.
%
% I = TrapComp(f,a,b,n), where
%
% f is an anonymous function representing the integrand,
% a and b are the limits of integration,
% n is the number of equal-length subintervals in [a,b],
%
% I is the integral estimate.
%
h = (b-a)/n; x = a:h:b;
y = f(x);
I = (y(1) + 2*sum(y(2:end-1)) + y(end))*h/2;
```

EXAMPLE 6.6: COMPOSITE TRAPEZOIDAL RULE

1. Evaluate the definite integral in Example 6.5 using the composite trapezoidal rule with $n = 8$:

$$\int_{-1}^1 \frac{1}{x+2} dx$$

2. Confirm the result by executing the user-defined function `TrapComp`.

Solution

1. The spacing size is $h = (b - a)/n = 0.25$ and the nine nodes are $x_1 = -1, -0.75, -0.5, \dots, 0.75, 1 = x_9$. Letting $f(x) = 1/(x + 2)$, the integral estimate is found by Equation 6.23 as follows:

$$\int_{-1}^1 \frac{1}{x+2} dx = \frac{0.25}{2} [f(-1) + 2f(-0.75) + 2f(-0.5) + \dots + 2f(0.75) + f(1)] = 1.1032$$

Recalling the actual value 1.0986, the relative error is calculated as 0.42%. As expected, and stated earlier, the accuracy of composite trapezoidal rule is compatible with the midpoint rectangular method and better than the composite rectangular rule using either endpoint.

2.

```
>> f = @(x)(1./(x+2));
>> I = TrapComp(f,-1,1,8)
```

```
I =
1.1032
```


6.3.5 Simpson's Rules

The trapezoidal rule estimates the value of a definite integral by approximating the integrand with a first-degree polynomial, the line connecting the points $(a, f(a))$ and $(b, f(b))$. Any method that uses a higher-degree polynomial to connect these points will provide a more accurate estimate. Simpson's 1/3 and 3/8 rules, respectively, use second and third-degree polynomials to approximate the integrand.

6.3.5.1 Simpson's 1/3 Rule

In evaluating $\int_a^b f(x) dx$, the Simpson's 1/3 rule uses a second-degree polynomial to approximate the integrand $f(x)$. The three points that are needed to determine this polynomial are picked as $x_1 = a$, $x_2 = (a + b)/2$, and $x_3 = b$ as shown in Figure 6.6a. Consequently, the second-degree Lagrange interpolating polynomial (Section 5.5) is constructed as

$$p_2(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} f(x_1) + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} f(x_2) + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} f(x_3)$$

The definite integral will then be evaluated with this polynomial replacing the integrand

$$\int_a^b f(x) dx \cong \int_a^b p_2(x) dx$$

Substituting for $p_2(x)$, integrating from a to b , and simplifying, yields

$$\int_a^b f(x) dx \cong \frac{h}{3} [f(x_1) + 4f(x_2) + f(x_3)], \quad h = \frac{b-a}{2} \tag{6.25}$$

The method is known as the 1/3 rule because h is multiplied by 1/3. The estimation error, which can be large depending on the nature of the integrand, can be improved significantly by repeated applications of the Simpson's 1/3 rule.

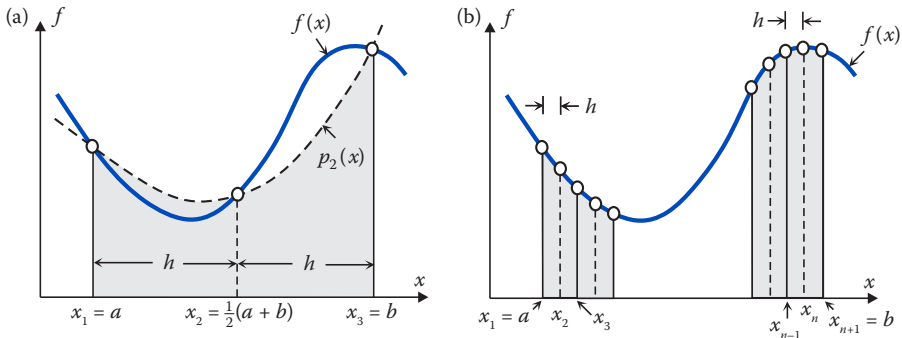


FIGURE 6.6

(a) Simpson's 1/3 rule and (b) composite Simpson's 1/3 rule.

6.3.5.2 Composite Simpson's 1/3 Rule

In the composite Simpson's 1/3 rule, the interval $[a, b]$ is divided into n subintervals defined by $n + 1$ points labeled $a = x_1, x_2, \dots, x_n, x_{n+1} = b$. Although the subintervals can have different widths, the results that follow are based on the assumption that the points are equally spaced with spacing size $h = (b - a)/n$. Since three points are needed to construct a second-degree interpolating polynomial, the Simpson's 1/3 rule must be applied to two adjacent subintervals at a time. For example, the first application will be to the first two subintervals $[x_1, x_2]$ and $[x_2, x_3]$ so that the three points at x_1, x_2 , and x_3 are used for polynomial construction. The next application will be to $[x_3, x_4]$ and $[x_4, x_5]$ so that x_3, x_4 , and x_5 are used for construction. Continuing this pattern, the very last interval is comprised of $[x_{n-1}, x_n]$ and $[x_n, x_{n+1}]$; see Figure 6.6b. Therefore, $[a, b]$ must be divided into an even number of subintervals for the composite 1/3 rule to be implemented. As a result,

$$\int_a^b f(x) dx \cong \frac{h}{3}[f(x_1) + 4f(x_2) + f(x_3)] + \frac{h}{3}[f(x_3) + 4f(x_4) + f(x_5)] \\ + \dots + \frac{h}{3}[f(x_{n-1}) + 4f(x_n) + f(x_{n+1})]$$

The even-indexed points (x_2, x_4, \dots, x_n) are the middle terms in each application of 1/3 rule, and therefore by Equation 6.25 have a coefficient of 4. The odd-indexed terms $(x_3, x_5, \dots, x_{n-1})$ are the common points to adjacent intervals and thus count twice and have a coefficient of 2. The two terms $f(x_1)$ and $f(x_{n+1})$ on the far left and far right each has a coefficient of 1. In summary,

$$\int_a^b f(x) dx = \frac{h}{3} \left\{ f(x_1) + 4 \sum_{i=2,4,6,\dots}^n f(x_i) + 2 \sum_{j=3,5,7,\dots}^{n-1} f(x_j) + f(x_{n+1}) \right\} + O(h^4) \quad (6.26)$$

6.3.5.3 Error Estimate for Composite Simpson's 1/3 Rule

The error for the composite Simpson's 1/3 rule can be shown to be

$$E = \left[-\frac{1}{180}(b-a)\overline{f^{(4)}} \right] h^4 = O(h^4) \quad (6.27)$$

where $\overline{f^{(4)}}$ is the estimated average value of $f^{(4)}$ over $[a, b]$. Therefore, the error $O(h^4)$ is superior to the composite trapezoidal rule which has an error of $O(h^2)$.

The user-defined function `Simpson` uses the composite Simpson's 1/3 rule to estimate the value of a definite integral.

```
function I = Simpson(f,a,b,n)
%
% Simpson estimates the value of the integral of f(x) from a to b
% by using the composite Simpson's 1/3 rule applied to n equal-length
% subintervals.
%
% I = Simpson(f,a,b,n), where
```

```

%
% f is an anonymous function representing the integrand,
% a, b are the limits of integration,
% n is the (even) number of subintervals,
%
% I is the integral estimate.
%
h = (b-a)/n; x = a:h:b; I = 0;
for i = 1:2:n,
    I = I + f(x(i)) + 4*f(x(i+1)) + f(x(i+2));
end
I = (h/3)*I;

```

EXAMPLE 6.7: COMPOSITE SIMPSON'S 1/3 RULE

1. Evaluate the definite integral in Examples 6.5 and 6.6 using the composite Simpson's 1/3 rule with $n = 8$:

$$\int_{-1}^1 \frac{1}{x+2} dx$$

2. Confirm the result by executing the user-defined function `Simpson`.

Solution

1. The spacing size is $h = (b - a)/n = 0.25$ and the nine nodes are defined as $x_1 = -1, -0.75, -0.5, \dots, 0.75, 1 = x_9$. Letting $f(x) = 1/(x + 2)$, the integral estimate is found by Equation 6.26 as follows:

$$\int_{-1}^1 \frac{1}{x+2} dx = \frac{0.25}{3} \left[f(-1) + 4f(-0.75) + 2f(-0.5) + 4f(-0.25) + 2f(0) + 4f(0.25) \right. \\ \left. + 2f(0.5) + 4f(0.75) + f(1) \right] \\ = 1.0987$$

Knowing the actual value is 1.0986, the relative error is calculated as 0.01%. As expected, the accuracy of the composite Simpson's 1/3 rule is superior to the composite trapezoidal rule. Recall that the relative error associated with the composite trapezoidal rule was calculated in Example 6.6 as 0.42%.

- 2.

```

>> f = @(x)(1/(x+2));
>> I = Simpson(f,-1,1,8)

```

```

I =
    1.0987

```

6.3.5.4 Simpson's 3/8 Rule

The Simpson's 3/8 rule uses a third-degree polynomial to approximate the integrand $f(x)$. The four points that are needed to form this polynomial are picked as the four equally spaced points $x_1 = a$, $x_2 = (2a + b)/3$, $x_3 = (a + 2b)/3$, and $x_4 = b$ with spacing size $h = (b - a)/3$

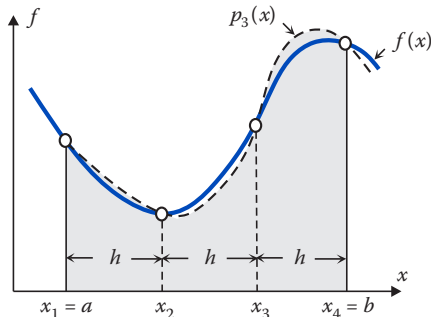


FIGURE 6.7
Simpson's 3/8 rule.

as shown in [Figure 6.7](#). The third-degree Lagrange interpolating polynomial (Section 5.5) is then constructed as

$$\begin{aligned}
 p_3(x) = & \frac{(x-x_2)(x-x_3)(x-x_4)}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} f(x_1) + \frac{(x-x_1)(x-x_3)(x-x_4)}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} f(x_2) \\
 & + \frac{(x-x_1)(x-x_2)(x-x_4)}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} f(x_3) + \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} f(x_4)
 \end{aligned}$$

The definite integral will be evaluated with this polynomial replacing the integrand

$$\int_a^b f(x) dx \cong \int_a^b p_3(x) dx$$

Substituting for $p_3(x)$, integrating from a to b , and simplifying, yields

$$\int_a^b f(x) dx \cong \frac{3h}{8} [f(x_1) + 3f(x_2) + 3f(x_3) + f(x_4)], \quad h = \frac{b-a}{3} \tag{6.28}$$

The method is known as the 3/8 rule because h is multiplied by 3/8. As before, the estimation error can be improved significantly by repeated applications of the Simpson's 3/8 rule.

6.3.5.5 Composite Simpson's 3/8 Rule

In the composite Simpson's 3/8 rule, the interval $[a, b]$ is divided into n subintervals defined by $n + 1$ points labeled $a = x_1, x_2, \dots, x_n, x_{n+1} = b$. The subintervals can have different widths, but the results presented here are based on the assumption that they are equally spaced with spacing size $h = (b - a)/n$. Since four points are needed to construct a third-degree polynomial, the Simpson's 3/8 rule is applied to three adjacent subintervals at a time. For example, the first application will be to the first three subintervals $[x_1, x_2]$, $[x_2, x_3]$, and $[x_3, x_4]$ so that the four points at x_1, x_2, x_3 , and x_4 are used for polynomial construction. The next application will be to $[x_4, x_5]$, $[x_5, x_6]$, and $[x_6, x_7]$ so that x_4, x_5, x_6 , and x_7 are used for construction. Continuing this pattern, the very last interval comprises of $[x_{n-2}, x_{n-1}]$, $[x_{n-1}, x_n]$, and

$[x_n, x_{n+1}]$. Therefore, $[a, b]$ must be divided into a number of subintervals that is a multiple of 3 for the composite 3/8 rule to be implemented. As a result,

$$\int_a^b f(x) dx \cong \frac{3h}{8}[f(x_1) + 3f(x_2) + 3f(x_3) + f(x_4)] + \frac{3h}{8}[f(x_4) + 3f(x_5) + 3f(x_6) + f(x_7)] \\ + \dots + \frac{3h}{8}[f(x_{n-2}) + 3f(x_{n-1}) + 3f(x_n) + f(x_{n+1})], \quad h = \frac{b-a}{n}$$

The middle terms in each application of 3/8 rule have a coefficient of 3 by Equation 6.28, while the common points to adjacent intervals are counted twice and have a coefficient of 2. The two terms $f(x_1)$ and $f(x_{n+1})$ on the far left and far right each has a coefficient of 1. In summary,

$$\int_a^b f(x) dx = \frac{3h}{8} \left\{ f(x_1) + 3 \sum_{i=2,5,8,\dots}^{n-1} [f(x_i) + f(x_{i+1})] + 2 \sum_{j=4,7,10,\dots}^{n-2} f(x_j) + f(x_{n+1}) \right\} + O(h^4) \quad (6.29)$$

We summarize the implementation of the composite Simpson rules as follows: if the number of subintervals is even, then Simpson's 1/3 rule is applied. If the number of subintervals is odd, then Simpson's 3/8 rule is applied to the last three subintervals and the 1/3 rule is applied to all previous ones; see Problem Set 6.3.

6.3.5.6 Error Estimate for Composite Simpson's 3/8 Rule

The error for the composite Simpson's 3/8 rule can be shown to be

$$E = \left[-\frac{1}{80}(b-a)\overline{f^{(4)}} \right] h^4 = O(h^4) \quad (6.30)$$

where $\overline{f^{(4)}}$ is the estimated average value of $f^{(4)}$ over the interval $[a, b]$. Therefore, the error $O(h^4)$ is compatible with that of the composite 1/3 rule.

The rectangular rule, trapezoidal rule, and the Simpsons' 1/3 and 3/8 rules all belong to a class of integration techniques known as Newton–Cotes formulas. Although there are higher-order formulas, which need more than four points to form the interpolating polynomial and naturally offer better accuracy, Simpson's rules are adequate for most applications in engineering. To improve estimation accuracy, the composite Simpson's rules are preferred to higher-order formulas. In the event that the integrand is given analytically, other methods such as Romberg integration and Gaussian quadrature (Section 6.4) are practical alternatives.

6.3.6 MATLAB Built-In Functions `quad` and `trapz`

MATLAB has two built-in functions to compute definite integrals: `quad` and `trapz`. The `quad` function handles cases where the integrand is given analytically, while `trapz` is used when the integrand is given as a discrete set of data.

QUAD Numerically evaluate integral, adaptive Simpson quadrature.

`Q = quad(FUN,A,B)` tries to approximate the integral of scalar-valued function `FUN` from `A` to `B` to within an error of `1.e-6` using recursive adaptive Simpson quadrature. `FUN` is a function handle. The function `Y = FUN(X)` should accept a vector argument `X` and return a vector result `Y`, the integrand evaluated at each element of `X`.

Note that `quad` uses adaptive Simpson quadrature. Adaptive integration methods adjust the number of subintervals needed to meet a desired accuracy by using more function evaluations in regions where the integrand shows rapid changes and less in areas where the integrand is well approximated by a quadratic function. In particular, adaptive Simpson quadrature uses an error estimate associated with the Simpson's rule, and if the error exceeds the desired tolerance, it divides the interval in two and applies Simpson's rule to each subinterval recursively.

The integral $\int_{-1}^1 [1/(x+2)] dx$, considered throughout this section, can be evaluated as follows:

```
function y = integrand(x)
y = 1./(x +2);
end
```

```
>> Q = quad(@integrand,-1,1)
```

```
Q =
```

```
1.0986
```

For situations where the integrand is defined as a set of discrete data, the built-in function `trapz` is used.

TRAPZ Trapezoidal numerical integration.

`Z = trapz(X,Y)` computes the integral of `Y` with respect to `X` using the trapezoidal method. `X` and `Y` must be vectors of the same length, or `X` must be a column vector and `Y` an array whose first non-singleton dimension is `length(X)`. `trapz` operates along this dimension.

In Example 6.6, we used the composite trapezoidal rule with $n=8$ to evaluate $\int_{-1}^1 [1/(x+2)] dx$. To confirm the result of that example using `trapz`, we must first generate a discrete set of data (x, y) equally spaced on $[-1, 1]$ with spacing size of $h = 0.25$.

```
>> f = @(x)(1./(x+2));
>> x = -1:0.25:1;
>> y = f(x);      % Generate 9 discrete values for integrand
>> I = trapz(x,y)
```

```
I =
```

```
1.1032      % Result agrees with that in Example 6.6
```

6.4 Numerical Integration of Analytical Functions: Romberg Integration, Gaussian Quadrature

Throughout Section 6.3 we presented numerical methods to evaluate integrals of analytical functions, as well as tabulated data. When the function is given analytically, it can be discretized at as many points as desired and these points are subsequently used to estimate the value of the integral. When the integrand is in tabulated form, only the given points in the data can be used for integral estimation and the number of points cannot be increased.

In this section, we introduce two methods that are exclusively developed to estimate the value of $\int_a^b f(x) dx$, where $f(x)$ is an analytical function. The first method is based on Richardson's extrapolation, which combines two numerical estimates of an integral to find a third, more accurate estimate. Richardson's extrapolation can be efficiently implemented using Romberg integration. The second method is the Gaussian quadrature, which approximates the value of the integral by using a weighted sum of values of $f(x)$ at several *nodes* in $[a, b]$. These nodes and the weights are determined such that the error is minimized.

6.4.1 Romberg Integration

The errors associated with the composite trapezoidal and Simpson's rules were shown in Equations 6.24 and 6.27 to be

$$E_{\text{trapezoid}} = \left[-\frac{1}{12} (b-a) \bar{f}'' \right] h^2 \stackrel{h=(b-a)/n}{=} -\frac{(b-a)^3}{12n^2} \bar{f}''$$

and

$$E_{\text{Simpson}} = \left[-\frac{1}{180} (b-a) \bar{f}^{(4)} \right] h^4 \stackrel{h=(b-a)/n}{=} -\frac{(b-a)^5}{180n^4} \bar{f}^{(4)}$$

This means in both cases, the error is reduced as n increases. Therefore, to achieve high levels of precision, a large number n of subintervals of $[a, b]$ are needed, requiring greater computational effort as n gets larger. Consequently, as an alternative to composite trapezoidal and Simpson's rules with large n , Romberg integration can be used to attain more accurate estimates more efficiently.

6.4.1.1 Richardson's Extrapolation

Richardson's extrapolation combines two numerical estimates of an integral to find a third, more accurate estimate. For example, two estimates each with error $O(h^2)$ can be combined to obtain an estimate with error $O(h^4)$. Similarly, two estimates each with error $O(h^4)$ can be combined to obtain an estimate with error $O(h^6)$. In general, Richardson's extrapolation combines two integral estimates each with order $O(h^{\text{even}})$ to obtain a third, more accurate estimate with error $O(h^{\text{even}+2})$.

As an estimate with error $O(h^2)$, consider the composite trapezoidal rule applied to n subintervals with spacing $h = (b - a)/n$, and let the corresponding integral estimate be I_h . Noting the error as given in Equation 6.24, the true value of the integral is expressed as

$$I \cong I_h - \left(\frac{b-a}{12} \overline{f''} \right) h^2$$

But since $\overline{f''}$ is the estimated average value of f'' over $[a, b]$, it is independent of h and we can rewrite the above as

$$I \cong I_h + Ch^2, \quad C = \text{const}$$

Suppose the composite trapezoidal rule is used with two different spacing sizes h_1 and h_2 to find two estimates I_{h_1} and I_{h_2} of the same integral. Then,

$$I \cong I_{h_1} + Ch_1^2$$

$$I \cong I_{h_2} + Ch_2^2$$

Eliminating C between the two equations, we find

$$I \cong \frac{\left(\frac{h_1}{h_2} \right)^2 I_{h_2} - I_{h_1}}{\left(\frac{h_1}{h_2} \right)^2 - 1}$$

It can be shown* that this new estimate has an error of $O(h^4)$. In particular, two estimates given by the composite trapezoidal rule applied with $h_1 = h$ and $h_2 = \frac{1}{2}h$ can be combined to obtain

$$I \cong \frac{2^2 I_{h/2} - I_h}{2^2 - 1} \tag{6.31}$$

Simplifying the above, and realizing the error of the estimate is $O(h^4)$, we have

$$I = \frac{4}{3} I_{h/2} - \frac{1}{3} I_h + O(h^4) \tag{6.32}$$

Note that Equation 6.32 can be used in connection with any integration formula that has an error of $O(h^2)$. Also note that the coefficients in Equation 6.32 add up to 1, hence act as weights attached to each estimate. With increasing accuracy, they place greater weight on the better estimate. For instance, using spacing size $\frac{1}{2}h$ generates a better estimate than the one using h , and consequently $I_{h/2}$ has a larger weight attached to it than I_h does.

* Refer to Ralston, A. and P. Rabinowitz, *A First Course in Numerical Analysis*, 2nd ed., McGraw-Hill, New York, 1978.

Similarly, it can readily be shown that two integral estimates with spacing sizes h_1 and h_2 , each with error $O(h^4)$, can be combined to obtain a more accurate estimate

$$I \cong \frac{\left(\frac{h_1}{h_2}\right)^4 I_{h_2} - I_{h_1}}{\left(\frac{h_1}{h_2}\right)^4 - 1}$$

with an error of $O(h^6)$. In particular, two estimates corresponding to $h_1 = h$ and $h_2 = \frac{1}{2}h$ can be combined to obtain

$$I \cong \frac{2^4 I_{h/2} - I_h}{2^4 - 1} \quad (6.33)$$

Simplifying, and realizing the error of the estimate is $O(h^6)$, we find

$$I = \frac{16}{15} I_{h/2} - \frac{1}{15} I_h + O(h^6) \quad (6.34)$$

Continuing in this fashion, two integral estimates corresponding to $h_1 = h$ and $h_2 = \frac{1}{2}h$, each with error $O(h^6)$, can be combined to obtain

$$I \cong \frac{2^6 I_{h/2} - I_h}{2^6 - 1} \quad (6.35)$$

so that

$$I = \frac{64}{63} I_{h/2} - \frac{1}{63} I_h + O(h^8) \quad (6.36)$$

EXAMPLE 6.8: RICHARDSON'S EXTRAPOLATION

Consider

$$\int_1^3 x \ln x \, dx$$

Application of the trapezoidal rule with $n = 2$, $n = 4$, and $n = 8$ yields three estimates with error $O(h^2)$, as listed in the column labeled "Level 1" in [Table 6.5](#), together with their respective percentage relative errors. Combining the first two estimates in Level 1 via Equation 6.32, we find a new estimate with error $O(h^4)$:

$$I \cong \frac{4}{3}(2.966568642984845) - \frac{1}{3}(3.034212794122055) = 2.944020592605774$$

Combining the second and third estimates in Level 1 also yields a new estimate with $O(h^4)$:

$$I \cong \frac{4}{3}(2.949472461900501) - \frac{1}{3}(2.966568642984845) = 2.943773734872386$$

TABLE 6.5

Integral Estimates at Three Levels of Accuracy; Example 6.8

n	Estimate $O(h^2)$ Level 1	Estimate $O(h^4)$ Level 2	Estimate $O(h^6)$ Level 3
2	3.034212794122055 (3.0729%)		
		2.944020592605774 (0.0090%)	
4	2.966568642984845 (0.7750%)		2.943757277690160 (0.000067%)
		2.943773734872386 (0.0006%)	
8	2.949472461900501 (0.1942%)		

These two better estimates are listed in Level 2 of Table 6.5. Combining these two via Equation 6.34 gives a new estimate with error $O(h^6)$, in Level 3:

$$I \cong \frac{16}{15}(2.943773734872386) - \frac{1}{15}(2.944020592605774) = 2.943757277690160$$

Noting the exact value of the integral is $\frac{9}{2}\ln 3 - 2 \cong 2.943755299006494$, the last estimate shows a five decimal accuracy.

6.4.1.2 Romberg Integration

In the foregoing analysis, Richardson’s extrapolation was employed to combine two estimates corresponding to spacing sizes h and $\frac{1}{2}h$, each with error $O(h^{\text{even}})$, to obtain a third, more accurate estimate with error $O(h^{\text{even}+2})$. The first three such results were shown in Equations 6.31, 6.33, and 6.35. In all three, the coefficients add up to 1, hence act as weights attached to each estimate, and with increasing accuracy, they place greater emphasis on the better estimate. These equations also follow a definite pattern that allows us to create a general formula, as

$$I_{i,j} = \frac{4^{j-1}I_{i+1,j-1} - I_{i,j-1}}{4^{j-1} - 1} \tag{6.37}$$

The entries $I_{1,1}, I_{2,1}, \dots, I_{m,1}$ are placed in the first column and represent the estimates by the composite trapezoidal rule with the number of subintervals $n, 2n, \dots, 2^{m-1}n$. For example, $I_{4,1}$ is the trapezoidal estimate applied to $2^{4-1}n = 8n$ subintervals. The second column has one element fewer than the first column, with entries $I_{1,2}, I_{2,2}, \dots, I_{m-1,2}$, which are obtained by combining every two successive entries of the first column and represent more accurate estimates. This continues until the very last column, whose only entry is $I_{1,m}$. This scheme is depicted in [Figure 6.8](#).

The user-defined function `Romberg` uses the scheme described in [Figure 6.8](#) to find integral estimates at various levels of accuracy.

```
function I = Romberg(f,a,b,n,n_levels)
%
% Romberg uses the Romberg integration scheme to find integral estimates
% at different levels of accuracy.
%
% I = Romberg(f,a,b,n,n_levels), where
```

```

%
% f is an anonymous function representing the integrand,
% a and b are the limits of integration,
% n is the initial number of equal-length subintervals in [a,b],
% n_levels is the number of accuracy levels,
%
% I is the matrix of integral estimates.
%
I = zeros(n_levels,n_levels); % Pre-allocate

% Calculate the first-column entries by using the composite
% trapezoidal rule, where the number of subintervals is doubled
% going from one element to the next.

for i = 1:n_levels,
    n_intervals = 2^(i-1)*n;
    I(i,1) = TrapComp(f,a,b,n_intervals);
end

% Starting with the second level, use Romberg scheme to generate
% the remaining entries of the table.

for j = 2:n_levels,
    for i = 1:n_levels-j+1,
        I(i,j) = (4^(j-1)*I(i+1,j-1)-I(i,j-1))/(4^(j-1)-1);
    end
end

```

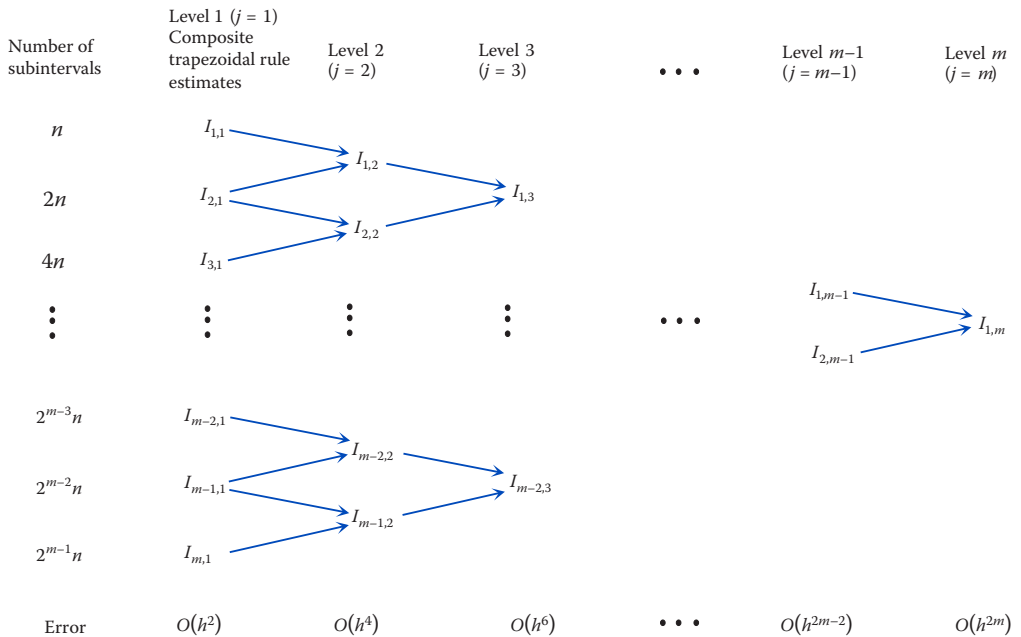


FIGURE 6.8

Romberg integration scheme.

The results of Example 6.8 can be verified by executing Romberg. Note that the initial number of subintervals for the application of composite trapezoidal rule was $n = 2$ and three levels of accuracy are desired.

```
>> format long
>> f = @(x)(x*log(x));
>> I = Romberg(f,1,3,2,3)
```

I =

3.034212794122055	2.944020592605774	2.943757277690160
2.966568642984845	2.943773734872386	0
2.949472461900501	0	0

As mentioned earlier, the Romberg integration scheme is more efficient than the trapezoidal and Simpson rules. Referring to the above example, if only Simpson's 1/3 rule were to be used, it would have to be applied with 14 subintervals to achieve 5-decimal accuracy.

6.4.2 Gaussian Quadrature

In estimating the value of $\int_a^b f(x) dx$ all the numerical integration methods presented up to now have been based on approximating $f(x)$ with a polynomial, followed by function evaluations at fixed, equally spaced points. But if these points were not fixed, we could pick them in such a way that the estimation error is minimized. Consider, for instance, the trapezoidal rule, [Figure 6.9a](#), where the (fixed) points on the curve must correspond to a and b . Without this limitation, we could select two points on the curve so that the area of the resulting trapezoid is a much better estimate of the area under the curve ([Figure 6.9b](#)).

The Gaussian quadrature is based on this general idea, and estimates the integral value by using a weighted sum of values of $f(x)$ at several points in $[a, b]$ that are not fixed, nor equally spaced. These points and the weights are determined such that the error is minimized.

The Gaussian quadrature is presented as applied to an integral in the explicit form

$$\int_{-1}^1 f(x) dx$$

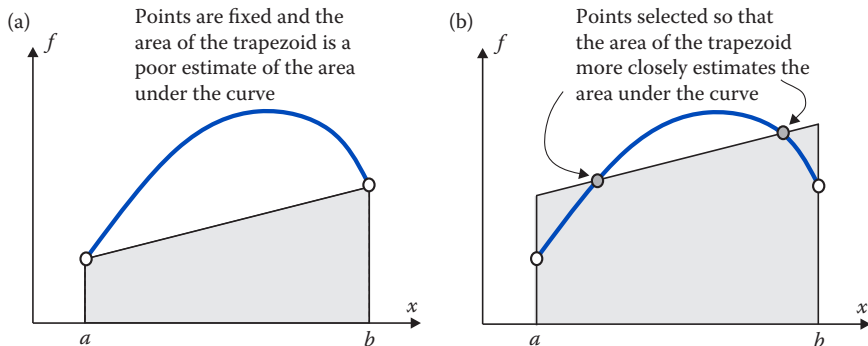


FIGURE 6.9

(a) Integral estimate by trapezoidal rule and (b) improved integral estimate.

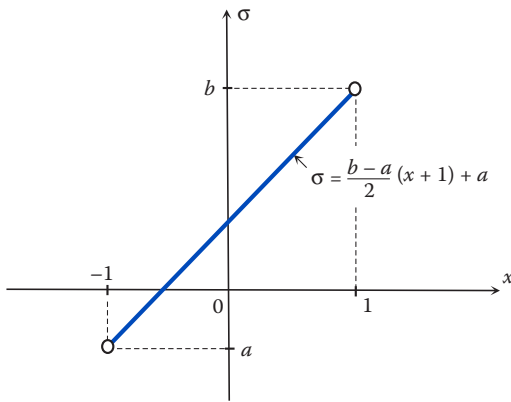


FIGURE 6.10

Linear transformation of data.

Note that any integral in the general form $\int_a^b f(\sigma) d\sigma$ can be converted to $\int_{-1}^1 f(x) dx$ via a linear transformation (see [Figure 6.10](#))

$$\sigma = \frac{b-a}{2}(x+1) + a \quad \text{so that} \quad d\sigma = \frac{b-a}{2} dx$$

and the new limits of integration are -1 and 1 . Upon substitution, the original integral is transformed into

$$\int_{-1}^1 f\left(\frac{b-a}{2}(x+1) + a\right) \frac{b-a}{2} dx \tag{6.38}$$

Gaussian quadrature estimates the integral as

$$\int_{-1}^1 f(x) dx \cong \sum_{i=1}^n c_i f(x_i) \tag{6.39}$$

where the weights c_i and the Gauss nodes x_i ($i = 1, 2, \dots, n$) are determined by assuming that Equation 6.39 fits exactly the above integral for functions $f(x) = 1, x, x^2, \dots$. How many of these functions need to be used depends on the value of n . For the simple case of $n = 2$, for example, we have

$$\int_{-1}^1 f(x) dx \cong c_1 f(x_1) + c_2 f(x_2) \tag{6.40}$$

so that there are four unknowns: weights c_1 and c_2 , and nodes x_1 and x_2 . The required four equations will be provided by fitting the integral for functions $f(x) = 1, x, x^2, x^3$:

$$f(x) = 1 \Rightarrow \int_{-1}^1 1 \cdot dx = c_1 + c_2 \Rightarrow \boxed{2 = c_1 + c_2}$$

$$f(x) = x \Rightarrow \int_{-1}^1 x \cdot dx = c_1 x_1 + c_2 x_2 \Rightarrow \boxed{0 = c_1 x_1 + c_2 x_2}$$

$$f(x) = x^2 \Rightarrow \int_{-1}^1 x^2 \cdot dx = c_1 x_1^2 + c_2 x_2^2 \Rightarrow \boxed{\frac{2}{3} = c_1 x_1^2 + c_2 x_2^2}$$

$$f(x) = x^3 \Rightarrow \int_{-1}^1 x^3 \cdot dx = c_1 x_1^3 + c_2 x_2^3 \Rightarrow \boxed{0 = c_1 x_1^3 + c_2 x_2^3}$$

Solving this system of four equations in four unknowns yields

$$c_1 = 1 = c_2$$

$$x_1 = -\frac{1}{\sqrt{3}} = -0.5773502692, \quad x_2 = \frac{1}{\sqrt{3}} = 0.5773502692$$

As a result, by Equation 6.40,

$$\int_{-1}^1 f(x) dx \cong f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \quad (6.41)$$

This provides the exact value of the integral as long as the integrand is any of the functions $f(x) = 1, x, x^2, x^3$ or their linear combination. Otherwise, it yields an approximate value of the integral.

The accuracy of approximation can be improved by increasing the value of n . For example, for the case of $n = 3$,

$$\int_{-1}^1 f(x) dx \cong c_1 f(x_1) + c_2 f(x_2) + c_3 f(x_3)$$

and there are now six unknowns: three weights c_1, c_2, c_3 , and three nodes x_1, x_2, x_3 . The six equations needed to solve for the unknowns are generated by fitting the integral for functions $f(x) = 1, x, x^2, x^3, x^4, x^5$. Proceeding as before, and solving the system of six equations, we arrive at

$$c_1 = 0.5555555556 = c_3, \quad c_2 = 0.8888888889$$

$$x_1 = -0.7745966692, \quad x_2 = 0, \quad x_3 = 0.7745966692$$

In general,

$$\int_{-1}^1 f(x) dx \cong \sum_{i=1}^n c_i f(x_i) = c_1 f(x_1) + c_2 f(x_2) + \cdots + c_n f(x_n)$$

TABLE 6.6

Weights and Nodes Used in the Gaussian Quadrature

n	Weights c_i	Gauss Nodes x_i
2	$c_1 = 1.000000000$	$x_1 = -0.577350269$
	$c_2 = 1.000000000$	$x_2 = 0.577350269$
3	$c_1 = 0.555555556$	$x_1 = -0.774596669$
	$c_2 = 0.888888889$	$x_2 = 0$
	$c_3 = 0.555555556$	$x_3 = 0.774596669$
4	$c_1 = 0.347854845$	$x_1 = -0.861136312$
	$c_2 = 0.652145155$	$x_2 = -0.339981044$
	$c_3 = 0.652145155$	$x_3 = 0.339981044$
	$c_4 = 0.347854845$	$x_4 = 0.861136312$
5	$c_1 = 0.236926885$	$x_1 = -0.906179846$
	$c_2 = 0.478628670$	$x_2 = -0.538469310$
	$c_3 = 0.568888889$	$x_3 = 0$
	$c_4 = 0.478628670$	$x_4 = 0.538469310$
	$c_5 = 0.236926885$	$x_5 = 0.906179846$
6	$c_1 = 0.171324492$	$x_1 = -0.932469514$
	$c_2 = 0.360761573$	$x_2 = -0.661209386$
	$c_3 = 0.467913935$	$x_3 = -0.238619186$
	$c_4 = 0.467913935$	$x_4 = 0.238619186$
	$c_5 = 0.360761573$	$x_5 = 0.661209386$
	$c_6 = 0.171324492$	$x_6 = 0.932469514$

which contains $2n$ unknowns: n weights and n nodes. The needed equations are generated by fitting the integral for functions $f(x) = 1, x, x^2, \dots, x^{2n-1}$. The resulting values of c_i and x_i are tabulated in Table 6.6 for $n = 2, \dots, 6$. It turns out that the weights c_1, c_2, \dots, c_n can be calculated via

$$c_i = \int_{-1}^1 \prod_{\substack{j=1 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j} dx \tag{6.42}$$

and the Gauss nodes x_1, x_2, \dots, x_n are the zeros of the n th-degree Legendre polynomial*. For example, for $n = 3$, the nodes are the zeros of $P_3(x) = \frac{1}{2}(5x^3 - 3x)$, that is, $0, \pm\sqrt{\frac{3}{5}}$, which agree with the values obtained earlier. The weights are computed via Equation 6.42 and will agree with those given above, as well as in Table 6.6.

EXAMPLE 6.9: GAUSSIAN QUADRATURE

Consider

$$\int_{0.1}^{0.5} e^{-x^3} dx$$

* The first five Legendre polynomials are

$P_0(x) = 1, P_1(x) = x, P_2(x) = \frac{1}{2}(3x^2 - 1), P_3(x) = \frac{1}{2}(5x^3 - 3x), P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3)$

1. Find an approximate value for the integral using the Gaussian quadrature with $n = 3$, $n = 4$, and $n = 5$.
2. How many subintervals must Simpson's 1/3 rule be applied to so that the accuracy is at the same level as that offered by the quadrature with $n = 5$? Use the user-defined function `Simpson` (Section 6.3) for this purpose. Use the `int` command (Chapter 2) to compute the actual value of the integral.

Solution

1. First rewrite the integral as $\int_{0.1}^{0.5} e^{-\sigma^3} d\sigma$ and then convert it into the standard form using the linear transformation

$$\sigma = \frac{b-a}{2}(x+1) + a = \frac{0.4}{2}(x+1) + 0.1 = 0.2x + 0.3 \quad \text{so that} \quad d\sigma = 0.2dx$$

Consequently, the integral in the desired form is

$$\int_{-1}^1 0.2e^{-(0.2x+0.3)^3} dx \quad \text{so that} \quad f(x) = 0.2e^{-(0.2x+0.3)^3}$$

For the case of $n = 3$,

$$\begin{aligned} \int_{-1}^1 0.2e^{-(0.2x+0.3)^3} dx &\cong c_1 f(x_1) + c_2 f(x_2) + c_3 f(x_3) \\ &= 0.5555555556 \cdot f(-0.774596669) + 0.8888888889 \cdot f(0) \\ &\quad + 0.5555555556 \cdot f(0.774596669) \\ &= 0.384942060052956 \end{aligned}$$

For the case of $n = 4$,

$$\begin{aligned} \int_{-1}^1 0.2e^{-(0.2x+0.3)^3} dx &\cong c_1 f(x_1) + c_2 f(x_2) + c_3 f(x_3) + c_4 f(x_4) \\ &= 0.347854845 \cdot f(-0.861136312) + 0.652145155 \cdot f(-0.339981044) \\ &\quad + 0.652145155 \cdot f(0.339981044) + 0.347854845 \cdot f(0.861136312) \\ &= 0.384942137622670 \end{aligned}$$

Similarly, for $n = 5$,

$$\begin{aligned} \int_{-1}^1 0.2e^{-(0.2x+0.3)^3} dx &\cong c_1 f(x_1) + c_2 f(x_2) + c_3 f(x_3) + c_4 f(x_4) + c_5 f(x_5) \\ &= 0.384942135961292 \end{aligned}$$

2.

```
>> format long
>> syms x
>> f = @(x)(exp(-x.^3));
>> I_exact = double(int(f,x,0.1,0.5))
```

```
I_exact =
    0.384942135972449
```


The percentage relative error for the estimate corresponding to $n = 5$ is

```
>> abs((0.384942135961292-I_exact)/I_exact*100)
```

```
ans =
```

```
2.898333979996639e-09
```

To achieve similar accuracy, Simpson's 1/3 rule must be applied to roughly 96 subintervals:

```
>> I = Simpson(f,0.1,0.5,96);
```

```
>> (I-I_exact)/I_exact*100
```

```
ans =
```

```
2.709827266627403e-09
```

6.5 Improper Integrals

All numerical integration techniques introduced in Sections 6.3 and 6.4 were designed to estimate integrals in the form $\int_a^b f(x) dx$ where the limits a and b are finite. While it is quite common to see these types of integrals in engineering applications, there are situations where improper integrals are encountered and must be approximated numerically. Some of these integrals appear in the following forms:

$$\int_a^{\infty} f(x) dx \quad (a > 0), \quad \int_{-\infty}^{-b} f(x) dx \quad (b > 0), \quad \int_{-\infty}^{\infty} f(x) dx \quad (6.43)$$

Consider $\int_a^{\infty} f(x) dx, a > 0$. If the integrand $f(x)$ reduces to zero at least as fast as x^{-2} does as $x \rightarrow \infty$, then the integral is handled by a simple change of variable

$$x = \frac{1}{\sigma} \quad \text{so that} \quad dx = -\frac{1}{\sigma^2} d\sigma$$

Then,

$$\int_a^{\infty} f(x) dx = \int_{1/a}^0 f\left(\frac{1}{\sigma}\right) \cdot \left(\frac{-1}{\sigma^2}\right) d\sigma = \int_0^{1/a} \frac{1}{\sigma^2} f\left(\frac{1}{\sigma}\right) d\sigma \quad (6.44)$$

The only concern is that the integrand is singular at the lower limit. Because of this, an open Newton–Cotes formula such as the composite midpoint rule (Section 6.3) can be utilized so that the integral is estimated without using the data at the endpoint(s).

The integral $\int_{-\infty}^{-b} f(x) dx, b > 0$, can be dealt with in a similar manner, including the condition on the rate of reduction of $f(x)$ to zero. The last form $\int_{-\infty}^{\infty} f(x) dx$ is treated as follows: we first decompose the integral as

$$\int_{-\infty}^{\infty} f(x) dx = \int_{-\infty}^{-b} f(x) dx + \int_{-b}^a f(x) dx + \int_a^{\infty} f(x) dx \quad (6.45)$$

In the first integral, we choose $-b$ so that $f(x)$ has started to asymptotically converge to zero at least as fast as x^{-2} . In the last integral, a is chosen such that the condition on the rate of reduction of $f(x)$ to zero is met as well. The integral in the middle can be approximated using a closed Newton–Cotes formula such as Simpson’s 1/3 rule.

EXAMPLE 6.10: IMPROPER INTEGRAL

Consider

$$\int_2^{\infty} \frac{\sin x}{x^2} dx$$

This is in the form of the first integral in Equation 6.43 hence we use the change of variable $x = 1/\sigma$ leading to Equation 6.44:

$$\int_2^{\infty} \frac{\sin x}{x^2} dx = \int_0^{1/2} \frac{1}{\sigma^2} \frac{\sin(1/\sigma)}{(1/\sigma)^2} d\sigma = \int_0^{1/2} \sin\left(\frac{1}{\sigma}\right) d\sigma$$

Noting that the integrand is singular at the lower limit, we will use the composite midpoint method with $h = 0.0125$ to estimate this last integral.

```
% Use s to represent sigma
f = @(s)(sin(1/s)); h = 0.0125; s = 0:h:0.5;
n = length(s)-1; m = zeros(n,1); I = 0;

for i = 1:n,
m(i) = (s(i+1)+s(i))/2;
I = I + f(m(i));
end

I = I*h;

I =

    0.0277
```

Accuracy can be improved by reducing the spacing size h .

PROBLEM SET (CHAPTER 6)

Finite-Difference Formulas for Numerical Differentiation (Section 6.2)

- Consider $f(t) = e^{-t/3} \sin t$, $t = 1.3, 1.6, 1.9, 2.2$. Approximate $\dot{f}(1.9)$ using
 - Two-point backward difference formula.
 - Two-point forward difference formula.
 - Two-point central difference formula.
 - Three-point backward difference formula.

Find the percentage relative error in each case.

- Consider $g(t) = t^2 e^{-t/2}$, $t = 0, 0.4, 0.8, 1.2$. Approximate $\dot{g}(0.4)$ using
 - Two-point backward difference formula.
 - Two-point forward difference formula.

- Two-point central difference formula.
- Three-point forward difference formula.

Find the percentage relative error in each case.

3. ✎ Consider $f(x) = x^{-2}2^x$, $x = 1.7, 2.0, 2.3, 2.6, 2.9$. Approximate $f'(2.3)$ using
- Three-point backward difference formula.
 - Three-point forward difference formula.
 - Four-point central difference formula.

Find the percentage relative error in each case.

4. ✎ Consider $f(x) = 3^x \log x$, $x = 2.0, 2.4, 2.8, 3.2, 3.6$. Find an estimate for $f'(2.8)$ using
- Three-point backward difference formula.
 - Three-point forward difference formula.
 - Four-point central difference formula.

Find the percentage relative error in each case.

5. ✎ The position of a moving object has been recorded as shown in [Table P5](#).
- Find the velocity of the object at $t = 3$ s. using the three-point backward difference formula.
 - Using the result of (a), and applying the two-point central difference formula, predict the position at $t = 3.5$ s.

TABLE P5

Time, t (seconds)	Position, x (meters)
1	0.75
1.5	1.35
2	2.50
2.5	3.25
3	4.55

6. ✎ The data in [Table P6](#) show the population of Canada recorded every 10 years between 1960 and 2010.
- Find the rate of population growth in 2010 using the three-point backward difference formula.

TABLE P6

Year, t	Population, p (millions)
1960	17.9
1970	21.3
1980	24.6
1990	27.8
2000	30.8
2010	34.1

- b. Using the result of (a), and applying the two-point central difference formula, predict the population in 2020.
7. ✍ Consider $g(x) = (2x + 1)e^{-3x/4}$, $x = 1, 1.5, 2, 2.5, 3$. Approximate $g''(2)$ using
- Three-point central difference formula.
 - Five-point central difference formula.
- Find the percentage relative error in each case.
8. ✍ The position of a moving object has been recorded as shown in [Table P8](#).
- a. Find the acceleration of the object at $t = 1.9$ s. using the four-point backward difference formula.
- b. Using the result of (a), and the three-point central difference formula, predict the position at $t = 2.2$ s.

TABLE P8

Time, t (seconds)	Position, x (meters)
0.7	0.62
1	0.76
1.3	1.02
1.6	1.18
1.9	1.49

9. ✍ The deflection u of a beam along its longitudinal (x) axis has been recorded as shown in [Table P9](#). The bending moment at any point along this beam is modeled as $M(x) = 1.05u''(x)$. All parameters are in consistent physical units. Find an estimate for the bending moment at $x = 0.6$ using
- a. The three-point central difference formula.
- b. The three-point backward difference formula.

TABLE P9

Position x	Deflection u
0.2	-0.13
0.4	-0.21
0.6	-0.22
0.8	-0.14

10. ✍ Let $f(x) = 3^{x/2} - 2$.
- a. Approximate $f'(2.4)$ using the two-point central difference formula with $h = 0.2$.
- b. Approximate $f'(2.4)$ using the two-point central difference formula with $h = 0.1$.
- c. Apply an appropriate form of Richardson's extrapolation to the results of (a) and (b) to obtain a superior estimate.
- d. Calculate the percentage relative errors in (a) through (c), compare, and discuss.
11. ✍ Repeat Problem 10 for $g(x) = 2^{-x/3} \ln x$.

12. ✎ Let $f(x) = x^{1/2} \sin x$.
- Approximate $f'(3.4)$ using the four-point central difference formula with $h = 0.2$.
 - Approximate $f'(3.4)$ using the four-point central difference formula with $h = 0.1$.
 - Apply an appropriate form of Richardson's extrapolation to the results of (a) and (b) to obtain a superior estimate.
 - Calculate the percentage relative errors in (a) through (c), compare, and discuss.
13. ✎ Let $g(x) = e^{-x} + 2^x$.
- Approximate $g''(1.5)$ using the three-point central difference formula with $h = 0.3$.
 - Approximate $g''(1.5)$ using the three-point central difference formula with $h = 0.15$.
 - Apply an appropriate form of Richardson's extrapolation to the results of (a) and (b) to obtain a superior estimate.
 - Calculate the percentage relative errors in (a) through (c), compare, and discuss.
14. ✎ For the unevenly spaced data in [Table P14](#) estimate the first derivative at $x = 1.75$ by
- Fitting a second-degree Lagrange interpolating polynomial to the set of the first three data points.
 - Fitting a third-degree Lagrange interpolating polynomial to the entire set.

TABLE P14

x	y
1.3	1.27
1.5	1.37
2	1.72
2.4	2.12

15. ✎ Given the unequally spaced data in [Table P15](#) estimate the first derivative at $x = 2.5$ by
- Fitting a second-degree Lagrange interpolating polynomial to the set of the first three data points.
 - Fitting a third-degree Lagrange interpolating polynomial to the entire set.

TABLE P15

x	y
2	5.8432
2.4	7.5668
2.6	8.5643
3	10.8731

16. Given the unequally spaced data in [Table P16](#) estimate the first derivative at $x = 2.3$ by
- ✎ Fitting a second-degree Lagrange interpolating polynomial to the set of the last three data points.


- b.  Applying the MATLAB built-in function `diff` to the entire set.

TABLE P16

x	y
2	1.0827
2.3	1.2198
2.7	1.3228
3	1.3443



17. Given the equally spaced data in [Table P17](#) estimate the second derivative at $x = 3.6$ by
-  Fitting a second-degree Lagrange interpolating polynomial to the set of the last three data points.
 -  Applying the MATLAB built-in function `diff` to the entire set. Compare with (a) and discuss.

TABLE P17

x	y
3	0.4817
3.3	0.9070
3.6	1.4496
3.9	2.1287



18.  Consider the equally spaced data in [Table P18](#). Using `polyfit` ([Chapter 5](#)), find the interpolating polynomial for the entire set. Differentiate this polynomial using `polyder`. Use this result to find the estimate of the first derivative at $x = 1.75$.

TABLE P18

x	y
0.5	11.3137
1.0	16.0000
1.5	22.6274
2.0	32.0000
2.5	45.2548
3.0	64.0000

Numerical Integration: Newton–Cotes Formulas (Section 6.3)

Composite Rectangular Rule

 In Problems 19 through 22 evaluate the definite integral using all three composite rectangular rule strategies with the indicated number of *equally spaced* data, calculate the percentage relative errors for all cases, and discuss.

19.
$$\int_1^3 e^{-3x/5} dx, n = 8$$

$$20. \int_1^5 \sqrt{2+xdx}, n = 10$$

$$21. \int_2^3 x \sin x dx, n = 10$$

$$22. \int_{0.4}^1 (x+1)e^x dx, n = 8$$

23. Write a user-defined function `I = Midpoint_Comp(f, a, b, n)` that uses the midpoint strategy of the rectangular rule to estimate the value of the integral of `f` from `a` to `b` using `n` subintervals, and `f` is an anonymous function representing the integrand. Then, apply `Midpoint_Comp` to estimate the value of

$$\int_1^3 e^{-x^2} \sin x dx, n = 10$$

Find the actual value of the integral using the `int` command and calculate the percentage relative error for the estimate.

24. Apply the user-defined function `Midpoint_Comp` (Problem 23) to estimate the value of

$$\int_1^4 \frac{\sin x}{x} dx, n = 20$$

Find the actual value of the integral using the `int` command and calculate the percentage relative error for the estimate.

25. In Problems 25 through 28, find the integral estimate using the left- and right-end composite rectangular rule strategies with the indicated *non-equally spaced* nodes, and calculate the percentage relative errors.

$$25. \int_1^3 x^{2/3} dx, \quad 1, 1.2, 1.5, 1.7, 1.8, 2.2, 2.5, 3$$

$$26. \int_0^{2.5} x^2 \sin x dx, \quad 0, 0.5, 0.8, 1.1, 1.6, 2, 2.3, 2.5$$

$$27. \int_{0.3}^{1.3} \cos^2(x+1) dx, \quad 0.3, 0.4, 0.6, 0.9, 1, 1.1, 1.2, 1.3$$

$$28. \int_0^3 \frac{5x}{2x^2+1} dx, \quad 0, 0.4, 0.9, 1.2, 1.8, 2.3, 2.6, 3$$

Composite Trapezoidal Rule

In Problems 29 through 32,

- ✎ Evaluate the integral using the composite trapezoidal rule with the given number of *equally spaced* subintervals.
- 🦋 Confirm the results by executing the user-defined function `TrapComp`.

29. $\int_1^4 \frac{1}{\ln(x+1)} dx, \quad n = 5$

30. $\int_0^{2.1} 2^x e^{-2x} dx, \quad n = 7$

31. $\int_{0.2}^{1.4} 2^{x+3} dx, \quad n = 6$

32. $\int_{0.3}^3 \sqrt{1+x^3} dx, \quad n = 9$

33. 🦋 For implementation of the composite trapezoidal rule when the nodes are generally not equally spaced, write a user-defined function with function call `I = TrapComp_Gen(f, x)` to estimate the value of the integral of `f` from `a` to `b`, where `f` is an anonymous function representing the integrand and `a` and `b` are the first and last entries of the vector `x`, whose components are not necessarily equally spaced. Apply `TrapComp_Gen` to estimate the value of

$$\int_1^3 e^{-x/2} \cos^2 2x dx$$

where

- $x = 1, 1.2, 1.6, 1.8, 1.9, 2, 2.1, 2.3, 2.6, 2.8, 3$.
 - $x = 1 : 0.2 : 3$ (equally spaced with increments of 0.2).
34. 🦋 Find the estimate for

$$\int_{-2}^0 e^{-x} \cos x dx$$

by executing the user-defined function `TrapComp_Gen` (Problem 33) with $x = -2, -1.8, -1.3, -0.9, -0.3, 0$. Find the actual value using the `int` command and calculate the percentage relative error associated with the integral estimate.

35. 🦋 Consider $\int_0^2 [x / (x^2 + 1)] dx$.

- Evaluate by executing `TrapComp_Gen` (Problem 33) with $x = 0, 0.2, 0.35, 0.6, 0.7, 0.9, 1.2, 1.5, 1.7, 1.8, 2$.


- b. Evaluate by executing `TrapComp` with $x = 0 : 0.2 : 2$ (equally spaced with increments of 0.2).
- c. Find the actual integral value using the `int` command and calculate the percentage relative errors associated with the integral estimates in (a) and (b).
36.  Write a user-defined function with function call `I = TrapComp_Data(x, y)` that estimates the value of the integral of a tabular data (x, y) , which is not necessarily equally spaced, using the composite trapezoidal rule. Apply `TrapComp_Data` to the data in [Table P36](#).

TABLE P36

x	0	0.15	0.25	0.40	0.50	0.55	0.65	0.8	0.90	1
y	0	0.29	0.23	0.33	0.38	0.39	0.42	0.38	0.34	0.31



37.  Write a user-defined function with function call `I = TrapComp_ESData(x, y)` that estimates the value of the integral of a tabular, equally spaced data (x, y) using the composite trapezoidal rule. Apply `TrapComp_ESData` to the data in [Table P37](#).

TABLE P37

x	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2	2.1
y	0.41	0.37	0.26	0.19	0.24	0.13	0.11	0.09	0.07	0.05	0.01

38.  A fragile instrument is placed inside a package to be protected during shipping and handling. The characteristics of the packing material are available experimentally, as shown in [Table P38](#). Assume that the force $F(x)$ exerted on the instrument is not to exceed 14 lbs. In order to determine the maximum safe drop height for the package, we first need to compute

$$\int_0^3 F(x) dx$$

- a. Evaluate this integral by executing the user-defined function `TrapComp_ESData` (Problem 37).
- b. Determine the sixth-degree interpolating polynomial for the data in [Table P38](#) using `polyfit`, and then integrate this polynomial from 0 to 3 to approximate the above integral.



TABLE P38

x (inches)	0	0.5	1	1.5	2	2.5	3
F (lbs)	0	0.5	1.1	1.75	3.75	7.25	14

Composite Simpson's 1/3 Rule

In Problems 39 through 43,

- a.  Evaluate the integral using composite Simpson's 1/3 rule with the given number of equally-spaced subintervals.

- b.  Confirm the results by executing the user-defined function `Simpson`.
- c.  Find the actual integral value and calculate the percentage relative error for the estimate in (b).


39.
$$\int_{2.05}^{4.15} 4^{3-2x} dx, \quad n = 6$$

40.
$$\int_{0.2}^{1.4} x^3 e^{-x} dx, \quad n = 6$$

41.
$$\int_3^7 \frac{x+3}{2x^2+x} dx, \quad n = 8$$



42.
$$\int_{3.1}^{4.3} 2^{1-x} \sin x dx, \quad n = 8$$

43.
$$\int_{-1}^3 \left(\frac{1}{3}x^2 + \frac{1}{2}\right)^2 dx, \quad n = 10$$

44.  For implementation of the composite Simpson's 1/3 rule when the nodes are generally not equally spaced, write a user-defined function with function call `I = Simpson_Gen(f, x)` to estimate the value of the integral of `f` from `a` to `b`, where `f` is an anonymous function representing the integrand and `a` and `b` are the first and last entries of the vector `x`, whose components are not necessarily equally spaced. Apply `Simpson_Gen` to estimate the value of

$$\int_0^1 \left(2 + \frac{1}{3} \sin 2x\right)^3 dx$$

where

- a. $x = 0, 0.1, 0.25, 0.3, 0.4, 0.55, 0.6, 0.7, 0.85, 0.9, 1$.
- b. $x = 0 : 0.1 : 1$ (equally spaced with increments of 0.1).
45.  Consider $\int_{-1}^1 \left[\frac{(x^3 - 1)}{(x^4 + 1)} \right] dx$.
- a. Evaluate by executing `Simpson_Gen` (Problem 44) with $x = -1, -0.85, -0.6, -0.4, -0.25, -0.1, 0, 0.25, 0.6, 0.8, 1$.
- b. Evaluate by executing `Simpson` with $x = -1 : 0.2 : 1$ (equally spaced with increments of 0.2).
46.  Consider $\int_1^4 x^2 \ln x dx$.
- a. Evaluate by executing `Simpson_Gen` (Problem 44) with $x = 1, 1.4, 1.7, 1.9, 2.3, 2.5, 2.6, 2.8, 3.3, 3.8, 4$.
- b. Evaluate by executing `Simpson` with $x = 1 : 0.3 : 4$ (equally spaced with increments of 0.3).
- c. Calculate the percentage relative errors associated with the results of (a) and (b).

47. ✎ Write a user-defined function with function call $I = \text{Simpson_Data}(x, Y)$ that estimates the value of the integral of a tabular data (x, Y) , not necessarily equally spaced, using the composite Simpson's 1/3 rule. Apply Simpson_Data to the data in Table P47.

TABLE P47

x	0	0.20	0.35	0.50	0.60	0.70	0.85	0.90	1
y	0	0.24	0.32	0.38	0.41	0.44	0.35	0.33	0.31

48. ✎ Write a user-defined function with function call $I = \text{Simpson_ESData}(x, Y)$ that estimates the value of the integral of a tabular, equally spaced data (x, Y) using the composite Simpson's 1/3 rule. Apply Simpson_ESData to the data in Table P48.

TABLE P48

x	1	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2
y	0.39	0.32	0.27	0.21	0.19	0.13	0.10	0.07	0.04	0.03	0.01

Composite Simpson's 3/8 Rule

- ✎ In Problems 49 through 52, evaluate each integral using the composite Simpson's 3/8 rule.

49. $\int_{1.2}^{4.8} x^{1/3} \cos(2x+1) dx, \quad n = 9$

50. $\int_0^{1.8} x^{2.5x} dx, \quad n = 9$

51. $\int_{0.2}^2 x^2 e^{-x/2} dx, \quad n = 6$

52. $\int_2^5 \frac{\ln(\frac{1}{2}x)}{2x+1} dx, \quad n = 6$

53. ✎ Write a user-defined function with function call $I = \text{Simpson_38}(f, a, b, n)$ that estimates the value of the integral of f from a to b using the composite Simpson's 3/8 rule applied to n subintervals of equal length, where f is an anonymous function. Apply Simpson_38 to estimate the value of

$$\int_1^3 \cos^3(2x-1) dx, \quad n = 12$$

Calculate the percentage relative error of the estimate.

54. ✎ Evaluate $\int_{-1}^{3.5} e^{-x} \sin^2 x dx$ by executing
- a. Simpson_38 (Problem 53) with $n = 6$.

- b. Simpson with $n = 6$.
- c. quad.
55. Evaluate $\int_{0.2}^{0.65} x^{-2} e^{2x} dx$ using the two functions listed below, and calculate the percentage relative error for each.
- Simpson_38 (Problem 53) with $n = 9$.
 - quad.
56. Write a user-defined function with function call `I = Simpson_38_ESData(x, y)` that estimates the integral of a tabular, equally spaced data (x, y) using the composite Simpson's 3/8 rule. Apply `Simpson_38_ESData` to the set of data generated by $y = x/(1 + \ln x)$ for $x = \text{linspace}(1, 2, 13)$.
57. The user-defined function `Simpson_38_ESData` (Problem 56) is to be applied to the set of data generated by $y = 2^{-x} \cos x$ for $x = \text{linspace}(-2, 1, n)$, where n is the number of subintervals. Write a MATLAB script to determine the smallest n such that the integral estimate by `Simpson_38_ESData` has a percentage relative error of at most 10^{-4} . Find the actual value of the integral using the `int` command where the integrand is $2^{-x} \cos x$.
58. Write a user-defined function with function call `I = Simpson_13_38(f, a, b, n)` that estimates the integral of f , an anonymous function, from a to b as follows: If $n = \text{even}$, it applies composite Simpson's 1/3 rule throughout, and if $n = \text{odd}$, it applies the 3/8 rule to the last three subintervals and the 1/3 rule to all the previous ones. Execute this function to evaluate

$$\int_2^5 \frac{\sin^2 x}{2x+3} dx$$

using $n = 20$ and $n = 25$.

59. In estimating $\int_a^b f(x) dx$, Boole's rule uses five equally spaced points to form a fourth-degree polynomial that approximates the integrand $f(x)$. The formula for Boole's rule is derived as

$$\int_a^b f(x) dx \cong \frac{2h}{45} [7f(x_1) + 32f(x_2) + 12f(x_3) + 32f(x_4) + 7f(x_5)], \quad h = \frac{b-a}{4}$$

The composite Boole's rule applies the above formula to four adjacent intervals at a time. Therefore, interval $[a, b]$ must be divided into a number of subintervals that is a multiple of 4. Write a user-defined function with function call `I = Boole_Comp(f, a, b, n)` that estimates the integral of f from a to b (with n subintervals) using the composite Boole's rule. Execute this function to evaluate

$$\int_1^3 \frac{x+1}{1-\cos x} dx, \quad n = 20$$

Also evaluate using the user-defined function `Simpson` (with the same n) and compare percentage relative errors. The actual integral value is computed via the `int` command.

60. Evaluate $\int_0^1 (x^2 + 1)^{-1} dx$ with $n = 20$ (subintervals) using

`TrapComp`

`Simpson`

`Boole_Comp` (Problem 59)

Find the actual integral value using the `int` command. Calculate the percentage relative errors corresponding to all three strategies used above. Discuss the results.

Numerical Integration of Analytical Expressions: Romberg integration, Gaussian quadrature (Section 6.4)

Romberg Integration

In Problems 61 through 64,

- Apply the trapezoidal rule with the indicated values of n to yield estimates with error $O(h^2)$. Then, calculate all subsequent higher-order estimates using Richardson's extrapolation, and tabulate the results as in Table 6.5. Also compute the relative error associated with each integral estimate.
- Confirm the results by executing `Romberg`. Use `format long`.

61. $\int_{-1}^1 \sqrt{1-x^2} dx, \quad n = 2, 4, 8$

62. $\int_0^{\pi/3} e^{-x/3} \sin 3x dx, \quad n = 2, 4, 8$

63. $\int_0^4 \frac{\cos x}{x^2 + 2} dx, \quad n = 2, 4, 8$

64. $\int_0^1 e^{x^2+1} dx, \quad n = 2, 4, 8, 16$

Gaussian Quadrature

In Problems 65 through 70, evaluate each integral using the Gaussian quadrature with the indicated number(s) of nodes.

65. $\int_{-4}^4 \frac{\sin x}{x+1} dx, \quad n = 4$

66. $\int_0^1 2^x x^3 dx, \quad n = 3, n = 4$

$$67. \int_0^5 \left(\sin 2x + \frac{1}{3} \cos x \right)^2 dx, \quad n = 3$$

$$68. \int_{-1}^3 e^{\cos x} \sin x dx, \quad n = 3, n = 5$$

$$69. \int_0^4 \frac{x}{x^3 + 2} dx, \quad n = 2, \quad n = 3, n = 4$$

$$70. \int_0^{3/2} \left(\frac{1}{2} x^2 + x + 3 \right)^2 dx, \quad n = 4$$

71. Write a user-defined function with syntax `I = Gauss_Quad_4(f, a, b)` that evaluates the integral of `f` (an anonymous function) from `a` to `b` using the Gaussian quadrature with `n=4`. Use the values for weights and Gauss nodes offered in Table 6.6. Execute `Gauss_Quad_4` to evaluate

$$\int_{1.5}^5 \frac{2x}{x^3 + x + 1} dx$$

72. Write a user-defined function with syntax `I = Gauss_Quad_5(f, a, b)` that evaluates the integral of `f` (an anonymous function) from `a` to `b` using the Gaussian quadrature with `n=5`. Use the values for weights and Gauss nodes offered in Table 6.6. Execute `Gauss_Quad_5` to evaluate

$$\int_1^3 \cos(x^2 + 2) dx$$


73. Use `format long` throughout. Consider

$$\int_2^4 x^2 \ln x dx$$

- Evaluate by executing the user-defined function `Simpson` using `n = 4`.
 - Evaluate by executing the user-defined function `Gauss_Quad_4` (Problem 71).
 - Calculate the percentage relative errors in (a) and (b), and discuss. Use the `int` command to find the actual integral value.
74. Use `format long` throughout. Consider

$$\int_1^2 \frac{e^{-x} \sin x}{x^2 + 1} dx$$

- Evaluate by executing the user-defined function `Gauss_Quad_5` (Problem 72).

- b. Evaluate by executing the user-defined function Romberg with $n = 2$ and $n_levels = 3$.
- c. Calculate the percentage relative errors in (a) and (b), and discuss. Use the `int` command to find the actual integral value.
75.  Write a user-defined function with syntax `I = Gauss_Quad(f, a, b, n)` that evaluates the integral of f (an anonymous function) from a to b using the Gaussian quadrature with n nodes. Use the fact that the Legendre polynomial of degree n is generated by (Rodrigues' formula)

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} [(x^2 - 1)^n]$$

Evaluate the following definite integral by using `Gauss_Quad`:

$$\int_1^6 e^{-x} x^{1/3} dx, \quad n = 8$$

76.  Evaluate the following definite integral by using `Gauss_Quad` (Problem 75):

$$\int_2^5 \cos(x+1) \ln x dx, \quad n = 10$$

Improper Integrals (Section 6.5)

77.  Estimate


$$\int_0^{\infty} e^{-x} x^{1/2} dx$$

as follows: decompose it as $\int_0^{\infty} \dots = \int_0^1 \dots + \int_1^{\infty} \dots$. Evaluate the first integral using composite Simpson's 1/3 rule with $n = 6$, and evaluate the second integral by first changing the variable and subsequently using the composite midpoint method with $h = 0.05$.

78.  Estimate

$$\int_{-\infty}^{\infty} \frac{1}{2x^2 + 1} dx$$

as follows: decompose it as $\int_{-\infty}^{\infty} \dots = \int_{-\infty}^{-1} \dots + \int_{-1}^1 \dots + \int_1^{\infty} \dots$. Evaluate the middle integral using composite Simpson's 1/3 rule with $n = 6$. Evaluate the first and third integrals by first changing the variable and subsequently using the composite midpoint method with $h = 0.05$.

79.  The cumulative distribution function (CDF) for the standard normal variable z (zero mean and standard deviation of 1) is defined as

$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^z e^{-z^2/2} dz$$

and gives the probability that an event is less than z . Approximate $\Phi(0.5)$.

80.  Evaluate

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-z^2/2} dz$$

Numerical Solution of Initial-Value Problems

7.1 Introduction

An n th-order differential equation is accompanied by n auxiliary conditions, which are required in order to determine the n constants of integration that arise in the general solution. When these conditions are provided at the same initial value of the independent variable, we have an initial-value problem (IVP). In other situations, these auxiliary conditions are specified at different values of the independent variable. And since these values are usually stated at the extremities of the system, these types of problems are referred to as boundary-value problems (BVPs).

In this chapter, we will present various methods to numerically solve initial-value problems. Stability and stiffness of differential equations will also be discussed. Treatment of BVPs is presented in [Chapter 8](#). Numerical methods for a single, first-order initial-value problem will be studied first ([Figure 7.1](#)). Some of these methods will be extended and used to solve higher-order and systems of differential equations.

A single, first-order initial-value problem is formulated as

$$y' = f(x, y), \quad y(a) = y_0, \quad \boxed{x_0 = a} \leq x \leq \boxed{b = x_n} \quad (7.1)$$

where y_0 is a prescribed initial condition, the independent variable x assumes values in $[a, b]$, and it is assumed that a unique solution $y(x)$ exists in the interval $[a, b]$. The interval is divided into n segments of equal length h so that

$$x_1 = x_0 + h, \quad x_2 = x_0 + 2h, \dots, \quad x_n = x_0 + nh$$

The solution at the point x_0 is available by the initial condition. The objective is to find estimates of the solution at the subsequent mesh points x_1, x_2, \dots, x_n .

7.2 One-Step Methods

One-step methods find the solution estimate y_{i+1} at the location x_{i+1} by extrapolating from the solution estimate y_i at the previous location x_i . Exactly how this new estimate is extrapolated from the old estimate depends on the specific numerical method used. [Figure 7.2](#)

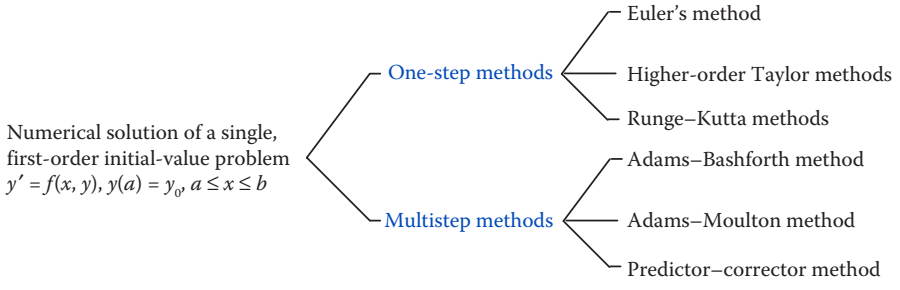


FIGURE 7.1
 Classification of numerical methods to solve an initial-value problem.

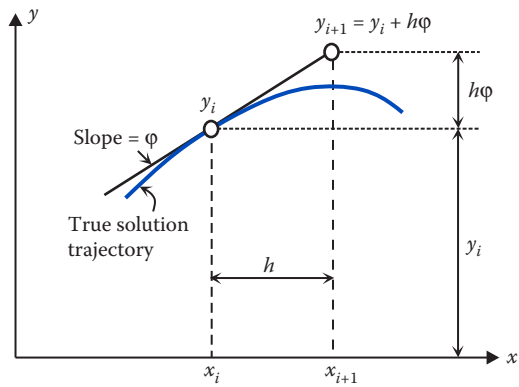


FIGURE 7.2
 A simple one-step method.

describes a very simple one-step method, where the slope ϕ is used to extrapolate from y_i to the new estimate y_{i+1} ,

$$y_{i+1} = y_i + h\phi, \quad i = 0, 1, 2, \dots, n-1 \tag{7.2}$$

Starting with the prescribed initial condition y_0 , Equation 7.2 is applied in every subinterval $[x_i, x_{i+1}]$ to find solution estimates at x_1, x_2, \dots, x_n . The general form in Equation 7.2 describes all one-step methods, with each method using a particular approach to estimate the slope ϕ . The simplest of all one-step methods is Euler's method, explained below.

7.3 Euler's Method

Expansion of $y(x_1)$ in a Taylor series about x_0 yields

$$y(x_1) = y(x_0 + h) = y(x_0) + hy'(x_0) + \frac{1}{2!}h^2y''(x_0) + \dots$$

Retaining the linear terms only, the above is rewritten as

$$y(x_1) = y(x_0) + hy'(x_0) + \frac{1}{2!}h^2y''(\xi_0)$$

for some ξ_0 between x_0 and x_1 . In general, expanding $y(x_{i+1})$ about x_i yields

$$y(x_{i+1}) = y(x_i) + hy'(x_i) + \frac{1}{2!}h^2y''(\xi_i)$$

for some ξ_i between x_i and x_{i+1} . Note that by Equation 7.1, we have $y'(x_i) = f(x_i, y_i)$. Introducing notations $y_i = y(x_i)$ and $y_{i+1} = y(x_{i+1})$, the estimated solution y_{i+1} can be found via

$$y_{i+1} = y_i + hf(x_i, y_i), \quad i = 0, 1, 2, \dots, n-1 \quad (7.3)$$

known as Euler's method. Comparing with the description of the general one-step method, Equation 7.2, we see that the slope ϕ at x_i is simply estimated by $f(x_i, y_i)$, which is the first derivative at x_i , namely, $y'(x_i)$. Equation 7.3 is called the difference equation for Euler's method.

The user-defined function `EulerODE` uses Euler's method to estimate the solution of an initial-value problem.

```
function y = EulerODE(f,x,y0)
%
% EulerODE uses Euler's method to solve a first-order initial-value
% problem in the form y' = f(x,y), y(x0) = y0.
%
% y = EulerODE(f,x,y0), where
%
%     f is an anonymous function representing f(x,y),
%     x is a vector representing the mesh points,
%     y0 is a scalar representing the initial value of y,
%
%     y is the vector of solution estimates at the mesh points.
%
y = 0*x;    % Pre-allocate
y(1) = y0; h = x(2)-x(1); n = length(x);
for i = 1:n-1,
    y(i+1) = y(i)+h*f(x(i),y(i));
end
```

EXAMPLE 7.1: EULER'S METHOD

Consider the initial-value problem

$$2y' + y = e^{-x}, \quad y(0) = \frac{1}{2}, \quad 0 \leq x \leq 1$$

The exact solution is derived as $y_{\text{exact}}(x) = \frac{3}{2}e^{-x/2} - e^{-x}$. We will solve the IVP numerically using Euler's method with step size $h = 0.1$. Comparing with Equation 7.1, we find

$f(x, y) = \frac{1}{2}(e^{-x} - y)$. Starting with $y_0 = \frac{1}{2}$, we use Equation 7.3 to find the estimate at the next mesh point ($x = 0.1$), as

$$y_1 = y_0 + hf(x_0, y_0) = \frac{1}{2} + 0.1f\left(0, \frac{1}{2}\right) = \frac{1}{2} + 0.1\left(\frac{1}{4}\right) = 0.5250$$

The exact solution at $x = 0.1$ is calculated as

$$y_{\text{exact}}(0.1) = 0.5220$$

Therefore, the relative error is 0.57%. Similar computations may be performed at the subsequent points 0.2, 0.3, ..., 1. The following MATLAB script uses the user-defined function EulerODE to find the numerical solution of the IVP and returns the results, including the exact values, in tabulated form. Note that the exact solution of the IVP is determined via dsolve in MATLAB.

```

disp('    x            yEuler        yExact')
h = 0.1; x = 0:0.1:1; y0 = 1/2;
f = @(x,y)((exp(-x)-y)/2);
yEuler = EulerODE(f,x,y0);
% Solve the IVP
y_exact = dsolve('2*Dy + y = exp(-x)', 'y(0)=1/2', 'x');
% Convert for evaluation purposes
y_exact = matlabFunction(y_exact);

for k = 1:length(x),
    x_coord = x(k);
    yE = yEuler(k);
    yEx = y_exact(x(k));

    fprintf('%6.2f    %11.6f    %11.6f\n', x_coord, yE, yEx)

end

```

x	yEuler	yExact	
0.00	0.500000	0.500000	
0.10	0.525000	0.522007	Hand calculations
0.20	0.543992	0.538525	
0.30	0.557729	0.550244	
0.40	0.566883	0.557776	
0.50	0.572055	0.561671	
0.60	0.573779	0.562416	
0.70	0.572531	0.560447	
0.80	0.568733	0.556151	
0.90	0.562763	0.549873	
1.00	0.554953	0.541917	

Figure 7.3 shows that Euler estimates capture the general trend of the actual solution. However, the percentage relative error gets larger as x increases, growing to 2.41% at the end of the interval at $x = 1$. Using a smaller step size h will reduce the errors but requires more computations. For example, executing the above script with $h = 0.05$ results in a maximum relative error of 1.18% at $x = 1$.

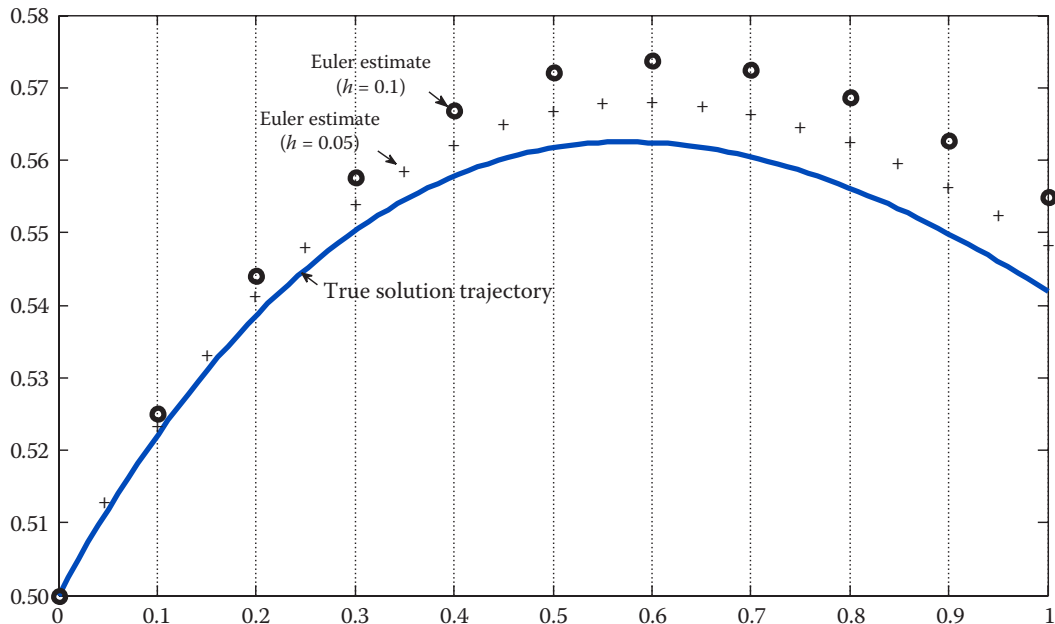


FIGURE 7.3
Comparison of Euler's and exact solutions in Example 7.1.

7.3.1 Error Analysis for Euler's Method

Two sources of error are involved in the numerical solution of ordinary differential equations: round-off and truncation. Round-off errors are caused by the number of significant digits retained and used for calculations by the computer. Truncation errors are caused by the way a numerical method approximates the solution, and comprise two parts. The first part is a local truncation error resulting from the application of the numerical method in each step. The second part is a propagated truncation error caused by the approximations made in the previous steps. Adding the local and propagated truncation errors yields the global truncation error. It can be shown that the local truncation error is $O(h^2)$, while the global truncation error is $O(h)$.

7.3.2 Calculation of Local and Global Truncation Errors

The global truncation error at the point x_{i+1} is simply the difference between the actual solution y_{i+1}^a and the computed solution y_{i+1}^c at that point. This contains the local truncation error, as well as the effects of all the errors accumulated in the steps prior to the current location x_{i+1} :

$$\text{Global truncation error at } x_{i+1} = \underbrace{y_{i+1}^a}_{\text{Actual solution at } x_{i+1}} - \underbrace{y_i^c + hf(x_i, y_i^c)}_{\text{Euler's estimate at } x_{i+1} \text{ using computed solution at } x_i} \quad (7.4)$$

The local truncation error at x_{i+1} is the difference between the actual solution y_{i+1}^a at that point and the solution that would have been computed had the actual solution been used by Euler's method going from x_i to x_{i+1} :

$$\text{Local truncation error at } x_{i+1} = \boxed{y_{i+1}^a} - \boxed{y_i^a + hf(x_i, y_i^a)} \quad (7.5)$$

Actual solution at x_{i+1}
Euler's estimate at x_{i+1} using actual solution at x_i

EXAMPLE 7.2: LOCAL AND GLOBAL TRUNCATION ERRORS

In Example 7.1, calculate the local and global truncation errors at each point and tabulate the results.

Solution

Starting with the initial condition $y_0 = \frac{1}{2}$, the Euler's computed value at $x_1 = 0.1$ is $y_1^c = 0.5250$ while the actual value is $y_1^a = 0.522007$. At this stage, the global and local truncation errors are the same because Euler's method used the initial condition, which is exact, to find the estimate. At $x_2 = 0.2$, the computed value is $y_2^c = 0.543992$, which was calculated by Euler's method using the estimated value $y_1^c = 0.5250$ from the previous step. If instead of y_1^c we use the actual value $y_1^a = 0.522007$, the computed value at x_2 is

$$\tilde{y}_2 = y_1^a + hf(x_1, y_1^a) = 0.522007 + 0.1f(0.1, 0.522007) = 0.541148$$

Therefore, local truncation error at x_2 is

$$y_2^a - \tilde{y}_2 = 0.538525 - 0.541148 = -0.002623$$

The global truncation error at x_2 is simply calculated as

$$y_2^a - y_2^c = 0.538525 - 0.543992 = -0.005467$$

It is common to express these errors in the form of percent relative errors, hence at each point, we evaluate

$$\frac{(\text{local or global}) \text{ truncation error}}{\text{actual value}} \times 100$$

With this, the (local) percentage relative error at x_2 is

$$\frac{y_2^a - \tilde{y}_2}{y_2^a} \times 100 = \frac{-0.002623}{0.538525} \times 100 \cong -0.49\%$$

The (global) percentage relative error at x_2 is

$$\frac{y_2^a - y_2^c}{y_2^a} \times 100 = \frac{-0.005467}{0.538525} \times 100 \cong -1.02\%$$

The following MATLAB script uses this approach to find the percentage relative errors at all x_i , and completes the table presented earlier in Example 7.1.

```
disp(' x          yEuler          yExact          e_local   e_global')
h = 0.1; x = 0:h:1; y0 = 1/2;
f = @(x,y)((exp(-x)-y)/2);
yEuler = EulerODE(f,x,y0);
y_exact = matlabFunction(dsolve('2*Dy + y = exp(-x)','y(0)=1/2','x'));

ytilda = 0*x; ytilda(1) = y0;
for n = 1:length(x)-1,
    ytilda(n+1) = y_exact(x(n)) + h*f(x(n),y_exact(x(n)));
end

for k = 1:length(x),
    x_coord = x(k);
    yE = yEuler(k);
    yEx = y_exact(x(k));
    e_local = (yEx-ytilda(k))/yEx*100;
    e_global = (yEx-yE)/yEx*100;

fprintf('%6.2f %11.6f %11.6f %6.2f %6.2f\n',x_coord,yE,yEx,e_local,e_global)

end
```

x	yEuler	yExact	e_local	e_global	
0.00	0.500000	0.500000	0.00	0.00	
0.10	0.525000	0.522007	-0.57	-0.57	
0.20	0.543992	0.538525	-0.49	-1.02	Hand calculations (see above)
0.30	0.557729	0.550244	-0.42	-1.36	
0.40	0.566883	0.557776	-0.36	-1.63	
0.50	0.572055	0.561671	-0.31	-1.85	
0.60	0.573779	0.562416	-0.27	-2.02	
0.70	0.572531	0.560447	-0.23	-2.16	
0.80	0.568733	0.556151	-0.20	-2.26	
0.90	0.562763	0.549873	-0.17	-2.34	
1.00	0.554953	0.541917	-0.15	-2.41	Max. % rel. err. reported earlier

7.3.3 Higher-Order Taylor Methods

Euler’s method was developed by retaining only the linear terms in a Taylor series. Retaining more terms in the series is the premise of higher-order Taylor methods. Expanding $y(x_{i+1})$ in a Taylor series about x_i yields

$$y(x_{i+1}) = y(x_i) + hy'(x_i) + \frac{1}{2!}h^2y''(x_i) + \dots + \frac{1}{k!}h^k y^{(k)}(x_i) + \frac{1}{(k+1)!}h^{k+1}y^{(k+1)}(\xi_i)$$

where ξ_i is between x_i and x_{i+1} . The k th-order Taylor method is defined as

$$y_{i+1} = y_i + hp_k(x_i, y_i), \quad i = 0, 1, 2, \dots, n-1 \quad (7.6)$$

where

$$p_k(x_i, y_i) = f(x_i, y_i) + \frac{1}{2!}hf'(x_i, y_i) + \dots + \frac{1}{k!}h^{k-1}f^{(k-1)}(x_i, y_i)$$

It is clear that *Euler's method is a first-order Taylor method*. Recall that Euler's method has a local truncation error $O(h^2)$ and a global truncation error $O(h)$. The k th-order Taylor method has a local truncation error $O(h^{k+1})$ and a global truncation error $O(h^k)$. Therefore, the higher the order of the Taylor method, the more accurately it estimates the solution of the initial-value problem. However, this reduction in error demands the calculation of the derivatives of $f(x, y)$, which is an obvious drawback.

EXAMPLE 7.3: SECOND-ORDER TAYLOR METHOD

Solve the initial-value problem in Examples 7.1 and 7.2 using the second-order Taylor method with the same step size $h = 0.1$ as before, and compare the numerical results with those produced by Euler's method.

Solution

The problem is

$$2y' + y = e^{-x}, \quad y(0) = \frac{1}{2}, \quad 0 \leq x \leq 1$$

so that $f(x, y) = \frac{1}{2}(e^{-x} - y)$. Implicit differentiation with respect to x yields

$$f'(x, y) = \frac{1}{2}(-e^{-x} - y') \stackrel{y'=f(x,y)}{=} \frac{1}{2}\left[-e^{-x} - \frac{1}{2}(e^{-x} - y)\right] = \frac{1}{4}(-3e^{-x} + y)$$

By Equation 7.6, the second-order Taylor method is defined as

$$y_{i+1} = y_i + hp_2(x_i, y_i), \quad i = 0, 1, 2, \dots, n-1$$

where

$$p_2(x_i, y_i) = f(x_i, y_i) + \frac{1}{2!}hf'(x_i, y_i)$$

Therefore,

$$y_{i+1} = y_i + h\left[f(x_i, y_i) + \frac{1}{2!}hf'(x_i, y_i)\right]$$

Starting with $y_0 = \frac{1}{2}$, the solution estimate at the next location $x_1 = 0.1$ is calculated as

$$y_1 = y_0 + h\left[f(x_0, y_0) + \frac{1}{2}hf'(x_0, y_0)\right] = 0.521875$$

Noting the actual value at $x_1 = 0.1$ is 0.522007, this estimate has a (global) % relative error of 0.03%, which is a significant improvement over the -0.57% offered by Euler's method at the same location. This upgrading of accuracy was expected because the second-order Taylor method has a (global) truncation error $O(h^2)$ compared with $O(h)$ for Euler's method. As mentioned before, this came at the expense of the evaluation of the first derivative $f'(x,y)$. The following MATLAB script tabulates the solution estimates generated by the second-order Taylor method.

```
disp(' x          yEuler      yTaylor2      e_Euler    e_Taylor2')
h = 0.1; x = 0:h:1; y0 = 1/2;
f = @(x,y)((exp(-x)-y)/2); fp = @(x,y)((-3*exp(-x)+y)/4);
yEuler = EulerODE(f,x,y0);
y_exact = matlabFunction(dsolve('2*Dy + y = exp(-x)','y(0)=1/2','x'));

yTaylor2 = 0*x; yTaylor2(1) = y0;
for i = 1:length(x)-1,
    yTaylor2(i+1) = yTaylor2(i)+h*(f(x(i),yTaylor2(i)))+(1/2)*h*fp(x(i),yTaylor2(i));
end

for k = 1:length(x),
    x_coord = x(k); yE = yEuler(k); yEx = y_exact(x(k)); yT = yTaylor2(k);
    e_Euler = (yEx-yE)/yEx*100;
    e_Taylor2 = (yEx-yT)/yEx*100;
    fprintf('%6.2f %11.6f %11.6f %6.2f %6.2f\n',x_coord,yE,yT,e_Euler,e_Taylor2)
end
```

x	yEuler	yTaylor2	e_Euler	e_Taylor2	
0.00	0.500000	0.500000	0.00	0.00	
0.10	0.525000	0.521875	-0.57	0.03	Hand calculations (see above)
0.20	0.543992	0.538282	-1.02	0.05	
0.30	0.557729	0.549907	-1.36	0.06	
0.40	0.566883	0.557362	-1.63	0.07	
0.50	0.572055	0.561193	-1.85	0.08	
0.60	0.573779	0.561887	-2.02	0.09	
0.70	0.572531	0.559878	-2.16	0.10	
0.80	0.568733	0.555551	-2.26	0.11	
0.90	0.562763	0.549249	-2.34	0.11	
1.00	0.554953	0.541277	-2.41	0.12	Max. % rel. err.

7.4 Runge–Kutta Methods

In the last section, we learned that a k th-order Taylor method has a global truncation error $O(h^k)$ but requires the calculation of derivatives of $f(x,y)$. *Runge–Kutta methods generate solution estimates with the accuracy of Taylor methods without having to calculate these derivatives.* Recall from Equation 7.2 that all one-step methods to solve the initial-value problem

$$y' = f(x, y), \quad y(a) = y_0, \quad a = x_0 \leq x \leq x_n = b$$

are expressed as

$$y_{i+1} = y_i + h\varphi(x_i, y_i)$$

where $\varphi(x_i, y_i)$ is an increment function and is essentially a suitable slope over the interval $[x_i, x_{i+1}]$ that is used for extrapolating y_{i+1} from y_i . The number of points that are used in $[x_i, x_{i+1}]$ to determine this suitable slope defines the order of the Runge–Kutta method. For example, second-order Runge–Kutta methods use two points in each subinterval to find the representative slope, and so on.

7.4.1 Second-Order Runge–Kutta (RK2) Methods

For the second-order Runge–Kutta methods, the increment function is expressed as $\varphi(x_i, y_i) = a_1k_1 + a_2k_2$ so that

$$y_{i+1} = y_i + h(a_1k_1 + a_2k_2) \tag{7.7}$$

with

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + b_1h, y_i + c_{11}k_1h) \end{aligned} \tag{7.8}$$

where a_1, a_2, b_1 , and c_{11} are constants, each set determined separately for each specific RK2 method. These constants are evaluated by setting Equation 7.7 equal to the first three terms in a Taylor series, neglecting terms with h^3 and higher:

$$y_{i+1} = y_i + h y' \Big|_{x_i} + \frac{1}{2} h^2 y'' \Big|_{x_i} + O(h^3) \tag{7.9}$$

The term $y' \Big|_{x_i}$ is simply $f(x_i, y_i)$, while

$$y'' \Big|_{x_i} = f'(x_i, y_i) \stackrel{\text{Chain rule}}{=} \frac{\partial f}{\partial x} \Big|_{(x_i, y_i)} + \frac{\partial f}{\partial y} \Big|_{(x_i, y_i)} \frac{dy}{dx} \Big|_{x_i} \stackrel{y'=f(x,y)}{=} \frac{\partial f}{\partial x} \Big|_{(x_i, y_i)} + \frac{\partial f}{\partial y} \Big|_{(x_i, y_i)} f(x_i, y_i)$$

Substituting for $y' \Big|_{x_i}$ and $y'' \Big|_{x_i}$ in Equation 7.9, we have

$$y_{i+1} = y_i + hf(x_i, y_i) + \frac{1}{2} h^2 \frac{\partial f}{\partial x} \Big|_{(x_i, y_i)} + \frac{1}{2} h^2 \frac{\partial f}{\partial y} \Big|_{(x_i, y_i)} f(x_i, y_i) + O(h^3) \tag{7.10}$$

Next, we will calculate y_{i+1} using a different approach as follows. In Equation 7.7, the term $k_2 = f(x_i + b_1h, y_i + c_{11}k_1h)$ is a function of two variables, and can be expanded about (x_i, y_i) as

$$k_2 = f(x_i + b_1h, y_i + c_{11}k_1h) = f(x_i, y_i) + b_1h \frac{\partial f}{\partial x} \Big|_{(x_i, y_i)} + c_{11}k_1h \frac{\partial f}{\partial y} \Big|_{(x_i, y_i)} + O(h^2) \tag{7.11}$$

Substituting Equation 7.11 and $k_1 = f(x_i, y_i)$ in Equation 7.7, we find

$$\begin{aligned}
 y_{i+1} &= y_i + h \left\{ a_1 f(x_i, y_i) + a_2 \left[f(x_i, y_i) + b_1 h \left. \frac{\partial f}{\partial x} \right|_{(x_i, y_i)} + c_{11} k_1 h \left. \frac{\partial f}{\partial y} \right|_{(x_i, y_i)} + O(h^2) \right] \right\} \\
 &\stackrel{k_1 = f(x_i, y_i)}{=} y_i + (a_1 + a_2) h f(x_i, y_i) + a_2 b_1 h^2 \left. \frac{\partial f}{\partial x} \right|_{(x_i, y_i)} + a_2 c_{11} h^2 \left. \frac{\partial f}{\partial y} \right|_{(x_i, y_i)} f(x_i, y_i) + O(h^3) \quad (7.12)
 \end{aligned}$$

The right-hand sides of Equations 7.10 and 7.12 represent the same quantity, y_{i+1} , hence must be equal. That yields

$$a_1 + a_2 = 1, \quad a_2 b_1 = \frac{1}{2}, \quad a_2 c_{11} = \frac{1}{2} \quad (7.13)$$

Since there are four unknowns and only three equations, a unique set of solutions is not available. But if a value is assigned to one of the constants, the other three can be calculated. This is why there are several versions of RK2 methods, three of which are presented below. Second-order Runge–Kutta methods have local truncation error $O(h^3)$ and global truncation error $O(h^2)$, as did the second-order Taylor methods.

7.4.1.1 Improved Euler's Method

Assuming $a_2 = 1$, the other three constants in Equation 7.13 are determined as $a_1 = 0$, $b_1 = \frac{1}{2}$, and $c_{11} = \frac{1}{2}$. Inserting into Equations 7.7 and 7.8, improved Euler's method is described by

$$y_{i+1} = y_i + h k_2 \quad (7.14)$$

where

$$\begin{aligned}
 k_1 &= f(x_i, y_i) \\
 k_2 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h\right) \quad (7.15)
 \end{aligned}$$

7.4.1.2 Heun's Method

Assuming $a_2 = \frac{1}{2}$, the other three constants in Equation 7.13 are determined as $a_1 = \frac{1}{2}$, $b_1 = 1$, and $c_{11} = 1$. Inserting into Equations 7.7 and 7.8, Heun's method is described by

$$y_{i+1} = y_i + \frac{1}{2} h (k_1 + k_2) \quad (7.16)$$

where

$$\begin{aligned}
 k_1 &= f(x_i, y_i) \\
 k_2 &= f(x_i + h, y_i + k_1 h) \quad (7.17)
 \end{aligned}$$

7.4.1.3 Ralston's Method

Assuming $a_2 = \frac{2}{3}$, the other three constants in Equation 7.13 are determined as $a_1 = \frac{1}{3}$, $b_1 = \frac{3}{4}$, and $c_{11} = \frac{3}{4}$. Inserting into Equations 7.7 and 7.8, Ralston's method is described by

$$y_{i+1} = y_i + \frac{1}{3}h(k_1 + 2k_2) \quad (7.18)$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{3}{4}h, y_i + \frac{3}{4}k_1h\right) \end{aligned} \quad (7.19)$$

Note that each of these RK2 methods produces estimates with the accuracy of a second-order Taylor method without calculating the derivative of $f(x,y)$. Instead, *each method requires two function evaluations per step*.

7.4.1.4 Graphical Representation of Heun's Method

Equations 7.16 and 7.17 can be combined as

$$y_{i+1} = y_i + h \frac{f(x_i, y_i) + f(x_i + h, y_i + k_1h)}{2} \quad (7.20)$$

Since $k_1 = f(x_i, y_i)$, we have $y_i + k_1h = y_i + hf(x_i, y_i)$. But this is the estimate y_{i+1} given by Euler's method at x_{i+1} , which we denote by y_{i+1}^{Euler} to avoid confusion with y_{i+1} in Equation 7.20. With this, and the fact that $x_i + h = x_{i+1}$, Equation 7.20 is rewritten as

$$y_{i+1} = y_i + h \frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^{\text{Euler}})}{2} \quad (7.21)$$

The fraction multiplying h is the average of two quantities: the first one is the slope at the left end x_i of the interval and the second one is the estimated slope at the right end x_{i+1} of the interval. This is illustrated in [Figure 7.4](#).

In [Figure 7.4a](#), the slope at the left end of the interval is shown as $f(x_i, y_i)$. [Figure 7.4b](#) shows the estimated slope at the right end of the interval to be $f(x_{i+1}, y_{i+1}^{\text{Euler}})$. The line whose slope is the average of these two slopes, [Figure 7.4c](#), yields an estimate that is superior to y_{i+1}^{Euler} . In Heun's method, y_{i+1} is extrapolated from y_i using this line.

The user-defined function HeunODE uses Heun's method to estimate the solution of an initial-value problem.

```
function y = HeunODE(f,x,y0)
%
% HeunODE uses Heun's method to solve a first-order initial-value
```

```

% problem in the form  $y' = f(x,y)$ ,  $y(x_0) = y_0$ .
%
%  $y = \text{HeunODE}(f,x,y_0)$ , where
%
%  $f$  is an anonymous function representing  $f(x,y)$ ,
%  $x$  is a vector representing the mesh points,
%  $y_0$  is a scalar representing the initial value of  $y$ ,
%
%  $y$  is the vector of solution estimates at the mesh points.
%
%
y = 0*x; % Pre-allocate
y(1) = y0; h = x(2)-x(1); n = length(x);
for i = 1:n-1,
    k1 = f(x(i),y(i));
    k2 = f(x(i)+h,y(i)+h*k1);
    y(i+1) = y(i)+h*(k1+k2)/2;
end

```

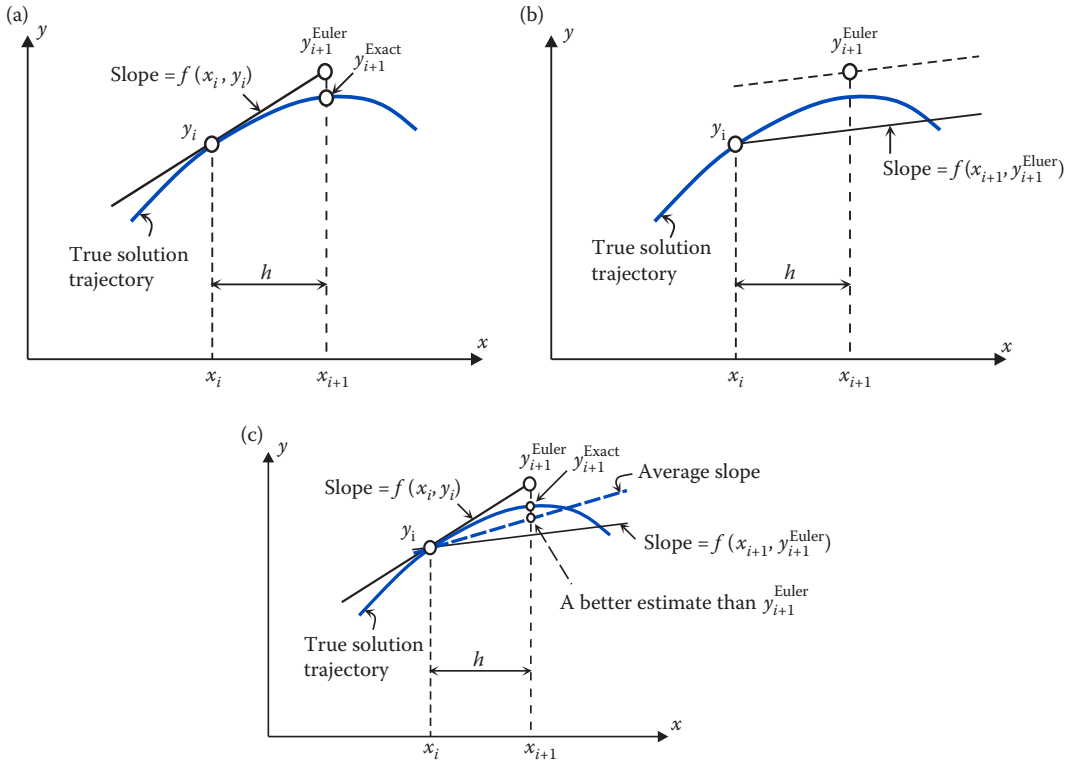


FIGURE 7.4
Graphical representation of Heun's method.

EXAMPLE 7.4: RK2 METHODS

Consider the initial-value problem

$$y' - x^2y = 2x^2, \quad y(0) = 1, \quad 0 \leq x \leq 1, \quad h = 0.1$$

Compute the estimated value of $y_1 = y(0.1)$ using each of the three RK2 methods discussed earlier.

Solution

Noting that $f(x, y) = x^2(2 + y)$, the value of $y_1 = y(0.1)$ estimated by each RK2 method is calculated as follows.

Improved Euler's method

$$\begin{aligned} k_1 &= f(x_0, y_0) = f(0, 1) = 0 \\ k_2 &= f\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_1h\right) = f(0.05, 1) = 0.0075 \end{aligned} \quad \Rightarrow \quad y_1 = y_0 + hk_2 = 1 + 0.1(0.0075) = 1.0008$$

Heun's method

$$\begin{aligned} k_1 &= f(x_0, y_0) = f(0, 1) = 0 \\ k_2 &= f(x_0 + h, y_0 + k_1h) = f(0.1, 1) = 0.0300 \end{aligned} \quad \Rightarrow \quad y_1 = y_0 + \frac{1}{2}h(k_1 + k_2) = 1 + 0.05(0.0300) = 1.0015$$

Ralston's method

$$\begin{aligned} k_1 &= f(x_0, y_0) = f(0, 1) = 0 \\ k_2 &= f\left(x_0 + \frac{3}{4}h, y_0 + \frac{3}{4}k_1h\right) = f(0.075, 1) = 0.0169 \end{aligned} \quad \Rightarrow \quad y_1 = y_0 + \frac{1}{3}h(k_1 + 2k_2) = 1 + \frac{1}{3}(0.1)(0.0338) = 1.0011$$

Continuing this process, the estimates given by the three methods at the remaining points will be obtained and tabulated as in [Table 7.1](#). The exact solution is $y = 3e^{x^3/3} - 2$. The global % relative errors for all three methods are also listed, where it is readily observed that all RK2 methods perform better than Euler.

TABLE 7.1

Summary of Calculations in Example 7.4

x	y_{Euler}	y_{Heun}	$y_{\text{Imp_Euler}}$	y_{Ralston}	e_{Euler}	e_{Heun}	$e_{\text{Imp_Euler}}$	e_{Ralston}
0.0	1.0000	1.0000	1.0000	1.0000	0.00	0.00	0.00	0.00
0.1	1.0000	<u>1.0015</u>	<u>1.0008</u>	<u>1.0011</u>	0.10	-0.05	0.02	-0.01
0.2	1.0030	1.0090	1.0075	1.0083	0.50	-0.10	0.05	-0.02
0.3	1.0150	1.0286	1.0263	1.0275	1.18	-0.15	0.08	-0.03
0.4	1.0421	1.0667	1.0636	1.0651	2.12	-0.19	0.10	-0.04
0.5	1.0908	1.1302	1.1261	1.1281	3.27	-0.23	0.14	-0.04
0.6	1.1681	1.2271	1.2219	1.2245	4.56	-0.25	0.17	-0.04
0.7	1.2821	1.3671	1.3604	1.3637	5.96	-0.27	0.22	-0.03
0.8	1.4430	1.5626	1.5541	1.5583	7.40	-0.28	0.27	-0.00
0.9	1.6633	1.8301	1.8191	1.8246	8.87	-0.27	0.34	0.04
1.0	1.9600	2.1922	2.1777	2.1849	10.37	-0.25	0.42	0.09

7.4.2 Third-Order Runge–Kutta (RK3) Methods

For the third-order Runge–Kutta methods, the increment function is expressed as $\varphi(x_i, y_i) = a_1k_1 + a_2k_2 + a_3k_3$ so that

$$y_{i+1} = y_i + h(a_1k_1 + a_2k_2 + a_3k_3) \quad (7.22)$$

with

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + b_1h, y_i + c_{11}k_1h) \\ k_3 &= f(x_i + b_2h, y_i + c_{21}k_1h + c_{22}k_2h) \end{aligned}$$

where $a_1, a_2, a_3, b_1, b_2, c_{11}, c_{21}$, and c_{22} are constants, each set determined separately for each specific RK3 method. These constants are found by setting Equation 7.22 equal to the first four terms in a Taylor series, neglecting terms with h^4 and higher. Proceeding as with RK2 methods, we will end up with six equations and eight unknowns. By assigning values to two of the constants, the other six can be determined. Because of this, there are several RK3 methods, two of which are presented here. Third-order Runge–Kutta methods have local truncation error $O(h^4)$ and global truncation error $O(h^3)$, as did the third-order Taylor methods.

7.4.2.1 The Classical RK3 Method

The classical third-order Runge–Kutta method is described by

$$y_{i+1} = y_i + \frac{1}{6}h(k_1 + 4k_2 + k_3) \quad (7.23)$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h\right) \\ k_3 &= f(x_i + h, y_i - k_1h + 2k_2h) \end{aligned}$$

7.4.2.2 Heun's RK3 Method

Heun's third-order Runge–Kutta method is described by

$$y_{i+1} = y_i + \frac{1}{4}h(k_1 + 3k_3) \quad (7.24)$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{1}{3}h, y_i + \frac{1}{3}k_1h\right) \\ k_3 &= f\left(x_i + \frac{2}{3}h, y_i + \frac{2}{3}k_2h\right) \end{aligned}$$

Each of these RK3 methods produces estimates with the accuracy of a third-order Taylor method without calculating the derivatives of $f(x, y)$. Instead, each method requires three function evaluations per step.

EXAMPLE 7.5: RK3 METHODS

Consider the initial-value problem in Example 7.4:

$$y' - x^2y = 2x^2, \quad y(0) = 1, \quad 0 \leq x \leq 1, \quad h = 0.1$$

Compute the estimated value of $y_1 = y(0.1)$ using the two RK3 methods presented above.

Solution

Noting that $f(x, y) = x^2(2 + y)$, the calculations are carried out as follows.

The classical RK3 method

$$k_1 = f(x_0, y_0) = f(0, 1) = 0$$

$$k_2 = f\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_1h\right) = f(0.5, 1) = 0.0075$$

$$k_3 = f(x_0 + h, y_0 - k_1h + 2k_2h) = f(0.1, 1.0015) = 0.0300$$

$$y_1 = y_0 + \frac{1}{6}h(k_1 + 4k_2 + k_3) = 1 + \frac{1}{6}(0.1)(4 \times 0.0075 + 0.0300) = 1.0010$$

Heun's RK3 method

$$k_1 = f(x_0, y_0) = f(0, 1) = 0$$

$$k_2 = f\left(x_0 + \frac{1}{3}h, y_0 + \frac{1}{3}k_1h\right) = f(0.333, 1) = 0.0033$$

$$k_3 = f\left(x_0 + \frac{2}{3}h, y_0 + \frac{2}{3}k_2h\right) = f(0.0667, 1.0002) = 0.0133$$

$$y_1 = y_0 + \frac{1}{4}h(k_1 + 3k_3) = 1 + 0.05(0.0300) = 1.0010$$

A summary of calculations is given in [Table 7.2](#) where it is readily seen that the global % relative errors for the two RK3 methods are considerably lower than those for Euler. And, as expected, the errors are also lower than those generated by the three RK2 methods used previously (see [Table 7.1](#)). This, of course, is because the global truncation error is $O(h^2)$ for RK2 methods and $O(h^3)$ for RK3 methods.

7.4.3 Fourth-Order Runge–Kutta (RK4) Methods

For the fourth-order Runge–Kutta methods the increment function is expressed as

$$\varphi(x_i, y_i) = a_1k_1 + a_2k_2 + a_3k_3 + a_4k_4$$

TABLE 7.2

Summary of Calculations in Example 7.5

x	y_{Euler}	y_{RK3}	$y_{\text{Heun_RK3}}$	e_{Euler}	e_{RK3}	$e_{\text{Heun_RK3}}$
0.0	1.0000	1.0000	1.0000	0.00	0.0000	0.0000
0.1	1.0000	1.0010	1.0010	0.10	-0.0000	0.0000
0.2	1.0030	1.0080	1.0080	0.50	-0.0001	0.0001
0.3	1.0150	1.0271	1.0271	1.18	-0.0004	0.0003
0.4	1.0421	1.0647	1.0647	2.12	-0.0010	0.0007
0.5	1.0908	1.1277	1.1276	3.27	-0.0018	0.0014
0.6	1.1681	1.2240	1.2239	4.56	-0.0030	0.0024
0.7	1.2821	1.3634	1.3633	5.96	-0.0044	0.0038
0.8	1.4430	1.5584	1.5582	7.40	-0.0059	0.0059
0.9	1.6633	1.8253	1.8250	8.87	-0.0074	0.0086
1.0	1.9600	2.1870	2.1866	10.37	-0.0087	0.0124

so that

$$y_{i+1} = y_i + h(a_1k_1 + a_2k_2 + a_3k_3 + a_4k_4) \tag{7.25}$$

with

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + b_1h, y_i + c_{11}k_1h\right) \\ k_3 &= f\left(x_i + b_2h, y_i + c_{21}k_1h + c_{22}k_2h\right) \\ k_4 &= f\left(x_i + b_3h, y_i + c_{31}k_1h + c_{32}k_2h + c_{33}k_3h\right) \end{aligned}$$

where a_j , b_j , and c_{ij} are constants, each set determined separately for each specific RK4 method. These constants are found by setting Equation 7.25 equal to the first five terms in a Taylor series, neglecting terms with h^5 and higher. Proceeding as before, leads to 10 equations and 13 unknowns. By assigning values to three of the constants, the other 10 can be determined. This is why there are many RK4 methods, but only the classical RK4 is presented here. Fourth-order Runge–Kutta methods have local truncation error $O(h^5)$ and global truncation error $O(h^4)$.

7.4.3.1 The Classical RK4 Method

The classical fourth-order Runge–Kutta method is described by

$$y_{i+1} = y_i + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) \tag{7.26}$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h\right) \\ k_3 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2h\right) \\ k_4 &= f(x_i + h, y_i + k_3h) \end{aligned} \tag{7.27}$$

RK4 methods produce estimates with the accuracy of a fourth-order Taylor method without calculating the derivatives of $f(x, y)$. Instead, four function evaluations per step are performed. The classical RK4 method is the most commonly used technique for numerical solution of first-order initial-value problems, as it offers the most acceptable balance of accuracy and computational effort.

The user-defined function RK4 uses the classical fourth-order Runge–Kutta method to estimate the solution of an initial-value problem.

```
function y = RK4(f,x,y0)
%
% RK4 uses the classical RK4 method to solve a first-order initial-value
% problem in the form  $y' = f(x,y)$ ,  $y(x_0) = y_0$ .
%
%  $y = \text{RK4}(f,x,y_0)$ , where
%
%  $f$  is an anonymous function representing  $f(x,y)$ ,
%  $x$  is a vector representing the mesh points,
%  $y_0$  is a scalar representing the initial value of  $y$ ,
%
%  $y$  is the vector of solution estimates at the mesh points.
%
y = 0*x; % Pre-allocate
y(1) = y0; h = x(2)-x(1); n = length(x);
for i = 1:n-1,
    k1 = f(x(i),y(i));
    k2 = f(x(i)+h/2,y(i)+h*k1/2);
    k3 = f(x(i)+h/2,y(i)+h*k2/2);
    k4 = f(x(i)+h,y(i)+h*k3);
    y(i+1) = y(i)+h*(k1+2*k2+2*k3+k4)/6;
end
```

EXAMPLE 7.6 : CLASSICAL RK4 METHOD

Consider the initial-value problem in Examples 7.4 and 7.5:

$$y' - x^2y = 2x^2, \quad y(0) = 1, \quad 0 \leq x \leq 1, \quad h = 0.1$$

1. Using hand calculations, compute the estimated value of $y_1 = y(0.1)$ by the classical RK4 method.
2. Solve the initial-value problem by executing the user-defined function RK4.

Solution

1. Noting that $f(x, y) = x^2(2 + y)$, the calculations are carried out as follows:

$$k_1 = f(x_0, y_0) = f(0, 1) = 0$$

$$k_2 = f\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_1h\right) = f(0.05, 1) = 0.0075$$

$$k_3 = f\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_2h\right) = f(0.05, 1.0004) = 0.0075$$

$$k_4 = f(x_0 + h, y_0 + k_3h) = f(0.1, 1.0008) = 0.0300$$

$$y_1 = y_0 + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) = \boxed{1.0010}$$

2.

```
>> f = @(x,y)((x^2)*(2+y));
>> x = 0:0.1:1;
>> y0 = 1;
>> y = RK4(f,x,y0); y = y'
```

y =

```
1.0000
1.0010
1.0080
1.0271
1.0647
1.1276
1.2240
1.3634
1.5583
1.8252
2.1868
```

A summary of all calculations is provided in [Table 7.3](#) where it is easily seen that the global % relative error for the classical RK4 method is significantly lower than all previous methods used up to this point. As expected, starting with Euler's method, which is indeed a first-order Runge–Kutta method, the accuracy improves with the order of the RK method.

7.4.4 Higher-Order Runge–Kutta Methods

The classical RK4 is the most commonly used numerical method for solving first-order initial-value problems. If higher levels of accuracy are desired, the recommended technique is Butcher's fifth-order Runge–Kutta method (RK5), which is defined as

TABLE 7.3

Summary of Calculations in Example 7.6

x	y_{RK4}	RK4 e_{RK4}	RK3 e_{RK3}	RK2 e_{Heun}	RK1 e_{Euler}
0.0	1.000000	0.000000	0.0000	0.00	0.00
0.1	1.001000	0.000001	-0.0000	-0.05	0.10
0.2	1.008011	0.000002	-0.0001	-0.10	0.50
0.3	1.027122	0.000003	-0.0004	-0.15	1.18
0.4	1.064688	0.000004	-0.0010	-0.19	2.12
0.5	1.127641	0.000005	-0.0018	-0.23	3.27
0.6	1.223966	0.000006	-0.0030	-0.25	4.56
0.7	1.363377	0.000007	-0.0044	-0.27	5.96
0.8	1.558286	0.000010	-0.0059	-0.28	7.40
0.9	1.825206	0.000016	-0.0074	-0.27	8.87
1.0	2.186837	0.000028	-0.0087	-0.25	10.37

$$y_{i+1} = y_i + \frac{1}{90}h(7k_1 + 32k_3 + 12k_4 + 32k_5 + 7k_6) \quad (7.28)$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{1}{4}h, y_i + \frac{1}{4}k_1h\right) \\ k_3 &= f\left(x_i + \frac{1}{4}h, y_i + \frac{1}{8}k_1h + \frac{1}{8}k_2h\right) \\ k_4 &= f\left(x_i + \frac{1}{2}h, y_i - \frac{1}{2}k_2h + k_3h\right) \\ k_5 &= f\left(x_i + \frac{3}{4}h, y_i + \frac{3}{16}k_1h + \frac{9}{16}k_4h\right) \\ k_6 &= f\left(x_i + h, y_i - \frac{3}{7}k_1h + \frac{2}{7}k_2h + \frac{12}{7}k_3h - \frac{12}{7}k_4h + \frac{8}{7}k_5h\right) \end{aligned}$$

Therefore, Butcher's RK5 method requires six function evaluations per step.

7.4.5 Selection of Optimal Step Size: Runge–Kutta Fehlberg (RKF) Method

Up to this point, we have discussed methods for solving the initial-value problem, Equation 7.1, that use a constant step size h . And in all those cases, the accuracy of the solution can be improved by reducing the step size. But a smaller step size requires a significant amount of additional computation. Another way to improve accuracy is to find the largest step size so that the global error is within a specified tolerance. The global error, however, is not generally available since the exact solution is not known. Therefore, the optimal step size must be determined based on the local truncation error which we have some knowledge of. One way to estimate the local truncation error for Runge–Kutta methods is to use two RK methods of different order and subtract the results. Naturally, a drawback of this approach is the number of function evaluations required per step. For example, we consider a common approach that uses a fourth-order and a fifth-order RK. This requires a total of 10 (four for RK4 and six for RK5) function evaluations per step. To get around the computational burden, the Runge–Kutta Fehlberg (RKF) method utilizes an RK5 method that uses the function evaluations provided by its accompanying RK4 method*. For this reason, the method is also commonly known as the RKF45. This will reduce the number of function evaluations per step from 10 to six. More specifically, the fourth-order accurate solution estimate is given by

$$y_{i+1} = y_i + h\left(\frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5\right) \quad (7.29)$$

* Refer to K.E. Atkinson, *An Introduction to Numerical Analysis*, 2nd ed., John Wiley, New York, 1989.

while the fifth-order accurate estimate is given by

$$y_{i+1} = y_i + h \left(\frac{16}{135} k_1 + \frac{6656}{12825} k_3 + \frac{28561}{56430} k_4 - \frac{9}{50} k_5 + \frac{2}{55} k_6 \right) \quad (7.30)$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{1}{4}h, y_i + \frac{1}{4}k_1h\right) \\ k_3 &= f\left(x_i + \frac{3}{8}h, y_i + \frac{3}{32}k_1h + \frac{9}{32}k_2h\right) \\ k_4 &= f\left(x_i + \frac{12}{13}h, y_i + \frac{1932}{2197}k_1h - \frac{7200}{2197}k_2h + \frac{7296}{2197}k_3h\right) \\ k_5 &= f\left(x_i + h, y_i + \frac{439}{216}k_1h - 8k_2h + \frac{3680}{513}k_3h - \frac{845}{4104}k_4h\right) \\ k_6 &= f\left(x_i + \frac{1}{2}h, y_i - \frac{8}{27}k_1h + 2k_2h - \frac{3544}{2565}k_3h + \frac{1859}{4104}k_4h - \frac{11}{40}k_5h\right) \end{aligned}$$

Note that the same values of k_1, k_2, \dots, k_6 are used for both estimates. Subtracting Equation 7.29 from 7.30 yields the estimate of the local truncation error:

$$\text{Error} = h \left(\frac{1}{360} k_1 - \frac{128}{4275} k_3 - \frac{2197}{75240} k_4 + \frac{1}{50} k_5 + \frac{2}{55} k_6 \right) \quad (7.31)$$

Therefore, in each step, the fourth-order accurate estimate is given by Equation 7.29 with the local truncation error provided by Equation 7.31. If the error is within an acceptable tolerance, the fourth-order estimate is accepted as a solution in that step and used in the next step. Otherwise, the step size gets adjusted until the tolerance condition is met.

7.4.5.1 Adjustment of Step Size

If an RK method of order $p + 1$ (with solution estimates given by \tilde{y}_{n+1}) is used to approximate the error in an RK method of order p (with solution estimates given by y_{n+1}), then in the n th step, the step size h is adjusted via

$$h_{\text{adj}} = qh = \alpha \left[\frac{\epsilon h}{|\tilde{y}_{n+1} - y_{n+1}|} \right]^{1/p} h$$

where $\alpha \leq 1$ is a suitable *adjustment factor* and ϵ is the tolerance. The value of α (hence q) is picked in a rather conservative manner. In particular, the Runge–Kutta Fehlberg method (RK45) mentioned above uses a fifth-order method to estimate the error of a fourth-order method so that $p = 4$. For this case, the recommended value for α , based on extensive experimentation by investigators, is $\alpha \cong 0.84$ so that

$$h_{\text{adj}} = 0.84 \left[\frac{\epsilon h}{|\tilde{y}_{n+1} - y_{n+1}|} \right]^{1/4} h$$

Note that the quantity $\tilde{y}_{n+1} - y_{n+1}$ is the local truncation error given by Equation 7.31. In order to ensure reasonable computational accuracy without too many evaluations per step, a minimum (h_{\min}) and a maximum (h_{\max}) are imposed on the step size. If $h_{\min} \leq h_{\text{adj}} \leq h_{\max}$, the value of h_{adj} is accepted. If h_{adj} is smaller than h_{\min} , we choose $h_{\text{adj}} = h_{\min}$. If it is greater than h_{\max} , we pick $h_{\text{adj}} = h_{\max}$. The procedure is then repeated in the subsequent intervals.

EXAMPLE 7.7: RKF METHOD

Consider the initial-value problem in Examples 7.4 through 7.6:

$$y' - x^2 y = 2x^2, \quad y(0) = 1, \quad 0 \leq x \leq 1$$

For the upper and lower bounds of step size, we choose $h_{\max} = 0.1$ and $h_{\min} = 0.01$. The error tolerance is selected as $\epsilon = 10^{-6}$. To begin the procedure, we set $h = h_{\max} = 0.1$. In the first step of the RKF45 (hand calculations), we find

$$\begin{aligned} k_1 &= f(0, 1) = 0 \\ k_2 &= f\left(x_0 + \frac{1}{4}h, y_0 + \frac{1}{4}k_1h\right) = 0.0001875 \\ k_3 &= f\left(x_0 + \frac{3}{8}h, y_0 + \frac{3}{32}k_1h + \frac{9}{32}k_2h\right) = 0.000422 \\ k_4 &= f\left(x_0 + \frac{12}{13}h, y_0 + \frac{1932}{2197}k_1h - \frac{7200}{2197}k_2h + \frac{7296}{2197}k_3h\right) = 0.002557 \\ k_5 &= f\left(x_0 + h, y_0 + \frac{439}{216}k_1h - 8k_2h + \frac{3680}{513}k_3h - \frac{845}{4104}k_4h\right) = 0.003001 \\ k_6 &= f\left(x_0 + \frac{1}{2}h, y_0 - \frac{8}{27}k_1h + 2k_2h - \frac{3544}{2565}k_3h + \frac{1859}{4104}k_4h - \frac{1}{40}k_5h\right) = 0.000750 \end{aligned}$$

Subsequently, the fourth-order accurate estimate is provided by Equation 7.29, as

$$y_1 = y_0 + h\left(\frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5\right) = 1.001000162865655$$

while the local truncation error is estimated by Equation 7.31, as

$$\text{Error} = h\left(\frac{1}{360}k_1 - \frac{128}{4275}k_3 - \frac{2197}{75240}k_4 + \frac{1}{50}k_5 + \frac{2}{55}k_6\right) = 1.343087565187744 \times 10^{-9}$$

We next examine the step size via

$$h_{\text{adj}} = 0.84 \left[\frac{\epsilon h}{|\tilde{y}_{n+1} - y_{n+1}|} \right]^{1/4} \quad h = 0.84 \left[\frac{10^{-6}(0.1)}{1.343087565187744 \times 10^{-9}} \right]^{1/4} = 2.4675$$

Since $h_{\text{adj}} > h_{\max}$, we set $h_{\text{adj}} = h_{\max} = 0.1$ and use this in the next step. Note that since $h = 0.1$ was accepted in this step, y_1 calculated above represents $y(0.1)$. Recall the exact solution given in Example 7.4 was $y_{\text{exact}} = 3e^{x^3/3} - 2$ so that $y_{\text{exact}}(0.1) = 1.001000166685187$. Running the classical RK4 method with a step size of 0.1 reveals that $y_{\text{RK4}}(0.1) = 1.001000156265625$. Comparison with y_1 shows that the solution estimate by RKF45 has a relative error of 0.00000038%, while that by RK4 is 0.000001%.

7.5 Multistep Methods

In single-step methods, the solution estimate y_{i+1} at x_{i+1} is obtained by using information at a single previous point x_i . Multistep methods are based on the idea that a more accurate

estimate for y_{i+1} at x_{i+1} can be attained by utilizing information on two or more previous points rather than one.

Consider $y' = f(x, y)$ subject to initial condition $y(x_0) = y_0$. To use a multistep method to find an estimate for y_i , information on at least two previous points are needed. However, the only available information is y_0 . This means that such methods cannot self start and the estimates at the first few points—depending on the order of the method—must be found using either a single-step method such as the classical RK4 or another multistep method that uses fewer previous points.

Multistep methods can be explicit or implicit. Explicit methods employ an explicit formula to calculate the estimate. For example, if an explicit method uses two previous points, the estimate y_{i+1} at x_{i+1} is in the form

$$y_{i+1} = F(x_{i+1}, x_i, y_i, x_{i-1}, y_{i-1})$$

This way, only known values appear on the right-hand side. In implicit methods, the unknown estimate y_{i+1} is involved on both sides of the equation

$$y_{i+1} = \tilde{F}(x_{i+1}, y_{i+1}, x_i, y_i, x_{i-1}, y_{i-1})$$

and must be determined iteratively using the methods described in [Chapter 3](#).

7.5.1 Adams–Bashforth Method

Adams–Bashforth method is an explicit multistep method to estimate the solution y_{i+1} of an IVP at x_{i+1} by using the solution estimates at two or more previous points. Several formulas can be derived depending on the number of previous points used. The order of each formula is the number of previous points it uses. For example, a second-order formula finds y_{i+1} by utilizing the estimates y_i and y_{i-1} at the two prior points x_i and x_{i-1} .

To derive the Adams–Bashforth formulas, we integrate $y' = f(x, y)$ over an arbitrary interval $[x_i, x_{i+1}]$,

$$\int_{x_i}^{x_{i+1}} y' dx = \int_{x_i}^{x_{i+1}} f(x, y) dx$$

Because $\int_{x_i}^{x_{i+1}} y' dx = y(x_{i+1}) - y(x_i)$, the above can be rewritten as

$$y(x_{i+1}) = y(x_i) + \int_{x_i}^{x_{i+1}} f(x, y) dx$$

or

$$y_{i+1} = y_i + \int_{x_i}^{x_{i+1}} f(x, y) dx \tag{7.32}$$

But since $y(x)$ is unknown, $f(x, y)$ cannot be integrated. To remedy this, $f(x, y)$ is approximated by a polynomial that interpolates the data at (x_i, y_i) and a few previous points. The number of the previous points that end up being used depends on the order of the formula to be derived. For example, for a second-order Adams–Bashforth formula, we use the polynomial that interpolates the data at (x_i, y_i) and one previous point, (x_{i-1}, y_{i-1}) , and so on.

7.5.1.1 Second-Order Adams–Bashforth Formula

The polynomial that interpolates the data at (x_i, y_i) and (x_{i-1}, y_{i-1}) is linear and in the form

$$p_1(x) = f(x_i, y_i) + \frac{f(x_i, y_i) - f(x_{i-1}, y_{i-1})}{x_i - x_{i-1}}(x - x_{i-1})$$

Letting $f_i = f(x_i, y_i)$ and $f_{i-1} = f(x_{i-1}, y_{i-1})$ for brevity, using $p_1(x)$ in Equation 7.32,

$$y_{i+1} = y_i + \int_{x_i}^{x_{i+1}} p_1(x) dx$$

and assuming equally spaced data with spacing h , we arrive at

$$y_{i+1} = y_i + \frac{1}{2}h(3f_i - f_{i-1}) \quad (7.33)$$

As mentioned earlier, this formula cannot self start because finding y_1 requires y_0 and y_{-1} , the latter not known. First, a single-step method such as RK4 is used to find y_1 from the initial condition y_0 . The first application of Equation 7.33 is when $i = 1$ so that y_2 can be obtained using the information on y_0 and y_1 .

7.5.1.2 Third-Order Adams–Bashforth Formula

Approximating the integrand $f(x, y)$ in Equation 7.32 by the second-degree polynomial $p_2(x)$ that interpolates the data at (x_i, y_i) , (x_{i-1}, y_{i-1}) , and (x_{i-2}, y_{i-2}) , and carrying out the integration, yields

$$y_{i+1} = y_i + \frac{1}{12}h(23f_i - 16f_{i-1} + 5f_{i-2}) \quad (7.34)$$

Since only y_0 is known, we first apply a method such as RK4 to find y_1 and y_2 . The first application of Equation 7.34 is when $i = 2$ to obtain y_3 by using the information on y_0 , y_1 , and y_2 .

7.5.1.3 Fourth-Order Adams–Bashforth Formula

Approximating the integrand $f(x, y)$ in Equation 7.32 by the third-degree polynomial $p_3(x)$ that interpolates the data at (x_i, y_i) , (x_{i-1}, y_{i-1}) , (x_{i-2}, y_{i-2}) , and (x_{i-3}, y_{i-3}) , and carrying out the integration, yields

$$y_{i+1} = y_i + \frac{1}{24}h(55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}) \quad (7.35)$$

Since only y_0 is known, we first apply a method such as RK4 to find y_1 , y_2 , and y_3 . The first application of Equation 7.35 is when $i = 3$ to obtain y_4 by using the information on y_0 , y_1 , y_2 , and y_3 .

Adams–Bashforth formulas are primarily used in conjunction with the Adams–Moulton formulas, which are also multistep but implicit, to be presented next. A weakness of higher-order Adams–Bashforth formulas is that stability requirements place limitations on the step size that is necessary for desired accuracy. The user-defined function `AdamsBashforth4` uses the fourth-order Adams–Bashforth formula to estimate the solution of an initial-value problem.

```
function y = AdamsBashforth4(f,x,y0)
%
% AdamsBashforth4 uses the fourth-order Adams-Bashforth formula to solve
% a first-order initial-value problem in the form y' = f(x,y), y(x0) = y0.
%
% y = AdamsBashforth4(f,x,y0), where
%
% f is an anonymous function representing f(x,y),
% x is a vector representing the mesh points,
% y0 is a scalar representing the initial value of y,
%
% y is the vector of solution estimates at the mesh points.
%
y(1:4) = RK4(f,x(1:4),y0); n = length(x);
for i = 4:n-1,
    h = x(i+1)-x(i);
    y(i+1) = y(i)+h*(55*f(x(i),y(i))-59*f(x(i-1),y(i-1))+37*f(x(i-2),
y(i-2))-9*f(x(i-3),y(i-3)))/24;
end
```

7.5.2 Adams–Moulton Method

Adams–Moulton method is an implicit multistep method to estimate the solution y_{i+1} of an IVP at x_{i+1} by using the solution estimates at two or more previous points, as well as (x_{i+1}, y_{i+1}) , where the solution is to be determined. Several formulas can be derived depending on the number of points used. The order of each formula is the total number of points it uses. For example, a second-order formula finds y_{i+1} by utilizing the estimates y_i and y_{i+1} at the points x_i and x_{i+1} . This makes the formula implicit because the unknown y_{i+1} will appear on both sides of the ensuing equation.

Derivation of Adams–Moulton formulas is similar to Adams–Bashforth where the integrand in Equation 7.32 is approximated by a polynomial that interpolates the data at prior points, as well as the point where the solution is being determined.

7.5.2.1 Second-Order Adams–Moulton Formula

The polynomial that interpolates the data at (x_i, y_i) and (x_{i+1}, y_{i+1}) is linear and in the form

$$p_1(x) = f_i + \frac{f_{i+1} - f_i}{x_{i+1} - x_i}(x - x_i)$$

where $f_i = f(x_i, y_i)$ and $f_{i+1} = f(x_{i+1}, y_{i+1})$. Replacing $f(x, y)$ in Equation 7.32 with $p_1(x)$ and performing the integration, yields

$$y_{i+1} = y_i + \frac{1}{2}h(f_i + f_{i+1}) \quad (7.36)$$

This formula is implicit because $f_{i+1} = f(x_{i+1}, y_{i+1})$ contains y_{i+1} which is the solution being determined. In this type of a situation, y_{i+1} must be found iteratively using the techniques listed in Chapter 3. This formula has a global truncation error $O(h^2)$.

7.5.2.2 Third-Order Adams–Moulton Formula

Approximating the integrand $f(x, y)$ in Equation 7.32 by the second-degree polynomial that interpolates the data at (x_{i+1}, y_{i+1}) , (x_i, y_i) , and (x_{i-1}, y_{i-1}) , and carrying out the integration, yields

$$y_{i+1} = y_i + \frac{1}{12}h(5f_{i+1} + 8f_i - f_{i-1}) \quad (7.37)$$

Since only y_0 is initially known, a method such as RK4 is first applied to find y_1 . The first application of Equation 7.37 is when $i = 1$ to obtain y_2 implicitly. This formula has a global truncation error $O(h^3)$.

7.5.2.3 Fourth-Order Adams–Moulton Formula

Approximating the integrand $f(x, y)$ in Equation 7.32 by the third-degree polynomial that interpolates the data at (x_{i+1}, y_{i+1}) , (x_i, y_i) , (x_{i-1}, y_{i-1}) , and (x_{i-2}, y_{i-2}) , and carrying out the integration, we find

$$y_{i+1} = y_i + \frac{1}{24}h(9f_{i+1} + 19f_i - 5f_{i-1} + f_{i-2}) \quad (7.38)$$

Since only y_0 is initially known, a method such as RK4 is first applied to find y_1 and y_2 . The first application of Equation 7.38 is when $i = 2$ to obtain y_3 implicitly. This formula has a global truncation error $O(h^4)$.

7.5.3 Predictor–Corrector Methods

Predictor–corrector methods are a class of techniques that employ a combination of an explicit formula and an implicit formula to solve an initial-value problem. First, the explicit formula is used to predict the value of y_{i+1} . This predicted value is denoted by \tilde{y}_{i+1} . The predicted \tilde{y}_{i+1} is then used on the right-hand side of an implicit formula to obtain a new, more accurate value for y_{i+1} on the left-hand side.

The simplest predictor–corrector method is Heun’s method, presented in Section 7.4. Heun’s method first uses Euler’s method—an explicit formula—as the predictor to obtain y_{i+1}^{Euler} . This predicted value is then used in Equation 7.21, which is the corrector,

$$y_{i+1} = y_i + h \frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^{\text{Euler}})}{2}$$

to find a more accurate value for y_{i+1} . A modified version of this approach is derived next so that a desired accuracy may be achieved through repeated applications of the corrector formula.

7.5.3.1 Heun’s Predictor–Corrector Method

The objective is to find an estimate for y_{i+1} . The method is implemented as follows:

1. Find a first estimate for y_{i+1} , denoted by $y_{i+1}^{(1)}$, using Euler’s method, which is an explicit formula,

$$\text{Predictor } y_{i+1}^{(1)} = y_i + hf(x_i, y_i) \quad (7.39)$$

2. Improve the predicted estimate by solving using Equation 7.21 iteratively,

$$\text{Corrector } y_{i+1}^{(k+1)} = y_i + h \frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^{(k)})}{2}, \quad k = 1, 2, 3, \dots \quad (7.40)$$

Therefore, $y_{i+1}^{(1)}$ is used in Equation 7.40 to obtain $y_{i+1}^{(2)}$, and so on.

3. The iterations in Step 2 are terminated when the following criterion is satisfied:

$$\text{Tolerance } \left| \frac{y_{i+1}^{(k+1)} - y_{i+1}^{(k)}}{y_{i+1}^{(k+1)}} \right| < \epsilon \quad (7.41)$$

where ϵ is a prescribed tolerance.

4. If the tolerance criterion is met, increment i by 1 and set y_i equal to this last $y_{i+1}^{(k+1)}$ and go to Step 1.

7.5.3.2 Adams–Bashforth–Moulton (ABM) Predictor–Corrector Method

Several predictor–corrector formulas can be created by combining one of the (explicit) Adams–Bashforth formulas of a particular order as the predictor with the (implicit) Adams–Moulton formula of the same order as the corrector. The fourth-order formulas of these two methods, for example, can be combined to create the fourth-order Adams–Bashforth–Moulton (ABM4) predictor–corrector:

$$\text{Predictor } y_{i+1}^{(1)} = y_i + \frac{1}{24} h(55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}), \quad i = 3, 4, \dots, n \quad (7.42)$$

$$\text{Corrector } y_{i+1}^{(k+1)} = y_i + \frac{1}{24} h(9f_{i+1}^{(k)} + 19f_i - 5f_{i-1} + f_{i-2}), \quad k = 1, 2, 3, \dots \quad (7.43)$$

where $f_{i+1}^{(k)} = f(x_{i+1}, y_{i+1}^{(k)})$. This method cannot self start and is implemented as follows: starting with the initial condition y_0 , apply a method such as RK4 to find estimates for y_1, y_2 , and y_3 and calculate their respective $f(x, y)$ values. At this stage, the predictor (Equation 7.42) is applied to find $y_4^{(1)}$, which is then used to calculate $f_4^{(1)}$. The corrector (Equation 7.43) is next applied to obtain $y_4^{(2)}$. The estimate can be substituted back into Equation 7.43 for iterative correction. The process is repeated for the remaining values of the index i .

The user-defined function `ABM4PredCorr` uses the fourth-order Adams–Bashforth–Moulton predictor–corrector method to estimate the solution of an initial-value problem. Note that *this function does not perform the iterative correction mentioned above*.

```
function y = ABM4PredCorr(f,x,y0)
%
% ABM4PredCorr uses the fourth-order Adams-Bashforth-Moulton predictor-
% corrector formula to solve a first-order initial-value problem in the
% form y' = f(x,y), y(x0) = y0.
%
%   y = ABM4PredCorr(f,x,y0), where
%
%       f is an anonymous function representing f(x,y),
%       x is a vector representing the mesh points,
%       y0 is a scalar representing the initial value of y,
%
%       y is the vector of solution estimates at the mesh points.
%
py = zeros(4,1); % Pre-allocate
y(1:4) = RK4(f,x(1:4),y0); % Find the first 4 elements by RK4
h = x(2) - x(1); n = length(x);

% Start ABM4
for i = 4:n-1,
    py(i+1) = y(i) + (h/24)*(55*f(x(i),y(i))-59*f(x(i-1),y(i-1))+
37*f(x(i-2),y(i-2))-9*f(x(i-3),y(i-3)));
    y(i+1) = y(i) + (h/24)*(9*f(x(i+1),py(i+1))+19*f(x(i),y(i))-
5*f(x(i-1),y(i-1))+f(x(i-2),y(i-2)));
end
```

EXAMPLE 7.8: ABM4 PREDICTOR–CORRECTOR METHOD

Consider the initial-value problem in Examples 7.4 through 7.7

$$y' - x^2y = 2x^2, \quad y(0) = 1, \quad 0 \leq x \leq 1, \quad h = 0.1$$

Compute the estimated value of $y_4 = y(0.4)$ using the ABM4 predictor–corrector method.

Solution

$f(x, y) = x^2(2 + y)$. The first element y_0 is given by the initial condition. The next three are obtained by RK4 as

$$y_1 = 1.001000, \quad y_2 = 1.008011, \quad y_3 = 1.027122$$

The respective $f(x,y)$ values are calculated next

$$f_0 = f(x_0, y_0) = f(0, 1) = 0, \quad f_1 = f(x_1, y_1) = f(0.1, 1.001000) = 0.030010$$

$$f_2 = f(x_2, y_2) = f(0.2, 1.008011) = 0.120320, \quad f_3 = f(x_3, y_3) = f(0.3, 1.027122) = 0.272441$$

Prediction

Equation 7.42 yields

$$y_4^{(1)} = y_3 + \frac{1}{24} h(55f_3 - 59f_2 + 37f_1 - 9f_0) = 1.064604$$

$$f_4^{(1)} = f(x_4, y_4^{(1)}) = f(0.4, 1.064604) = 0.490337$$

Correction

Equation 7.43 yields

$$y_4^{(2)} = y_3 + \frac{1}{24} h(9f_4^{(1)} + 19f_3 - 5f_2 + f_1) = 1.064696, \quad \text{Rel error} = 0.0008\%$$

This corrected value may be improved by substituting $y_4^{(2)}$ and the corresponding $f_4^{(2)} = f(x_4, y_4^{(2)})$ into Equation 7.43,

$$y_4^{(3)} = y_3 + \frac{1}{24} h(9f_4^{(2)} + 19f_3 - 5f_2 + f_1)$$

and inspecting the accuracy. In the present analysis, we perform only one correction so that $y_4^{(2)}$ is regarded as the value that will be used for y_4 . This estimate is then used in Equation 7.42 with the index i incremented from 3 to 4. Continuing this process, we generate the numerical results in [Table 7.4](#).

TABLE 7.4

Summary of Calculations in Example 7.8

x	y_{RK4}	\tilde{y}_i Predicted	y_i Corrected
0.0	1.000000		
0.1	1.001000		
0.2	1.008011		
0.3	1.027122	Start ABM4	
0.4		1.064604	1.064696
0.5		1.127517	1.127662
0.6		1.223795	1.224004
0.7		1.363143	1.363439
0.8		1.557958	1.558381
0.9		1.824733	1.825350
1.0		2.186134	2.187052

Another well-known predictor–corrector method is the fourth-order Milne’s method:

$$\text{Predictor } y_{i+1}^{(1)} = y_{i-3} + \frac{4}{3}h(2f_i - f_{i-1} + 2f_{i-2}), \quad i = 3, 4, \dots, n$$

$$\text{Corrector } y_{i+1}^{(k+1)} = y_{i-1} + \frac{1}{3}h(f_{i+1}^{(k)} + 4f_i + f_{i-1}), \quad k = 1, 2, 3, \dots$$

where $f_{i+1}^{(k)} = f(x_{i+1}, y_{i+1}^{(k)})$. As with the fourth-order Adams–Bashforth–Moulton, this method cannot self start and needs a method such as RK4 for estimating y_1, y_2 , and y_3 first.

7.6 Systems of Ordinary Differential Equations

Mathematical models of most systems in various engineering disciplines comprise one or more first- or higher-order differential equations subject to an appropriate number of initial conditions. In this section, we will achieve two tasks: (1) transform the model into a *system* of first-order differential equations and (2) numerically solve the system of first-order differential equations thus obtained.

7.6.1 Transformation into a System of First-Order ODEs

The first task is to show how a single higher-order IVP or a system of various-order IVPs may be transformed into a system of first-order IVPs. The most important tools that facilitate this process are the state variables.

7.6.1.1 State Variables

State variables form the smallest set of linearly independent variables that completely describe the state of a system. Given the mathematical model of a system, the state variables are determined as follows:

- *How many state variables are there?*

The number of state variables is the same as the number of initial conditions required to completely solve the model.

- *What are selected as state variables?*

The state variables are selected to be exactly those variables for which initial conditions are required.

7.6.1.2 Notation

State variables are represented by u_i ; for example, if there are three state variables, they will be denoted u_1, u_2 , and u_3 .

7.6.1.3 State-Variable Equations

If a system has m state variables u_1, u_2, \dots, u_m , then there are exactly m state-variable equations. Each of these equations is a first-order differential equation in the form

$$\dot{u}_i = f_i(t, u_1, u_2, \dots, u_m), \quad i = 1, 2, \dots, m \quad (7.44)$$

Therefore, the left side is the first derivative (with respect to time in most applications) of the state variable u_i , and the right side is an algebraic function of the state variables, and possibly time t explicitly. Note that only state variables and functions of time that are part of the system model are allowed to appear on the right side of Equation 7.44. The system of first-order differential equations in Equation 7.44 can be conveniently expressed in vector form, as

$$\dot{\mathbf{u}} = \mathbf{f}(t, \mathbf{u}), \quad \mathbf{u}(t) = \begin{Bmatrix} u_1 \\ u_2 \\ \dots \\ u_m \end{Bmatrix}, \quad \mathbf{f}(t, \mathbf{u}) = \begin{Bmatrix} f_1(t, \mathbf{u}) \\ f_2(t, \mathbf{u}) \\ \dots \\ f_m(t, \mathbf{u}) \end{Bmatrix} \quad (7.45)$$

where \mathbf{u} is the state vector.

EXAMPLE 7.9: A SINGLE HIGHER-ORDER IVP

Consider the third-order IVP described by

$$3y''' - y'' - 5y' - 3y = e^{-x/2}, \quad y(0) = 0, \quad y'(0) = -1, \quad y''(0) = 1$$

Three initial conditions are required, hence there are three state variables: u_1, u_2 , and u_3 . The state variables are those variables for which initial conditions are required. Therefore,

$$u_1 = y, \quad u_2 = y', \quad u_3 = y''$$

There are three state-variable equations, formed as follows:

$$\begin{aligned} u_1' &= y' = u_2 && u_1(0) = 0 \\ u_2' &= y'' = u_3 && \text{subject to } u_2(0) = -1 \\ u_3' &= y''' = \frac{1}{3}[y'' + 5y' + 3y + e^{-x/2}] = \frac{1}{3}[u_3 + 5u_2 + 3u_1 + e^{-x/2}] && u_3(0) = 1 \end{aligned}$$

In vector form,

$$\mathbf{u}' = \mathbf{f}(x, \mathbf{u}), \quad \mathbf{u} = \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix}, \quad \mathbf{f} = \begin{Bmatrix} u_2 \\ u_3 \\ \frac{1}{3}[u_3 + 5u_2 + 3u_1 + e^{-x/2}] \end{Bmatrix}, \quad \mathbf{u}_0 = \begin{Bmatrix} 0 \\ -1 \\ 1 \end{Bmatrix}$$

EXAMPLE 7.10: A SYSTEM OF DIFFERENT-ORDER IVPs

The mathematical model of a dynamic system is described by

$$\begin{aligned} \ddot{x}_1 + 2\dot{x}_1 + 2(x_1 - x_2) &= e^{-t} \sin t && \text{subject to initial conditions } x_1(0) = 0, x_2(0) = 0, \dot{x}_1(0) = 1 \\ \dot{x}_2 - 2(x_1 - x_2) &= 0 \end{aligned}$$

Three initial conditions are required, hence there are three state variables: $u_1, u_2,$ and u_3 . The state variables are those for which initial conditions are required. Therefore,

$$u_1 = x_1, \quad u_2 = x_2, \quad u_3 = \dot{x}_1$$

This is the *natural order for selecting state variables*, as the derivatives of variables are chosen after all non-derivatives have been assigned. There are three state-variable equations

$$\begin{aligned} \dot{u}_1 = \dot{x}_1 &= u_3 & u_1(0) &= 0 \\ \dot{u}_2 = \dot{x}_2 &= 2(x_1 - x_2) = 2(u_1 - u_2) & \text{subject to } u_2(0) &= 0 \\ \dot{u}_3 = \ddot{x}_1 &= -2\dot{x}_1 - 2(x_1 - x_2) + e^{-t} \sin t = -2u_3 - 2(u_1 - u_2) + e^{-t} \sin t & u_3(0) &= 1 \end{aligned}$$

In vector form,

$$\dot{\mathbf{u}} = \mathbf{f}(t, \mathbf{u}), \quad \mathbf{u} = \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix}, \quad \mathbf{f} = \begin{Bmatrix} u_3 \\ 2(u_1 - u_2) \\ -2u_3 - 2(u_1 - u_2) + e^{-t} \sin t \end{Bmatrix}, \quad \mathbf{u}_0 = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}$$

7.6.2 Numerical Solution of a System of First-Order ODEs

In Examples 7.9 and 7.10, we learned how to transform a single higher-order IVP or a system of different-order IVPs into one system of first-order IVPs in the general form

$$\mathbf{u}' = \mathbf{f}(x, \mathbf{u}), \quad \mathbf{u}(x_0) = \mathbf{u}_0, \quad a = x_0 \leq x \leq x_n = b \tag{7.46}$$

In many applications, the independent variable x is replaced with time t . Equation 7.46 is exactly in the form of Equation 7.1, where except for the independent variable x , all other quantities are vectors. And as such, the numerical methods presented so far in this chapter for solving Equation 7.1, a single first-order IVP, can be extended and applied to a system of first-order IVPs in Equation 7.46. We will present three of these methods here: Euler’s method, Heun’s method, and the classical fourth-order Runge–Kutta (RK4) method.

7.6.2.1 Euler’s Method for Systems

As before, the interval $[a, b]$ is divided into subintervals of equal length h such that

$$x_1 = x_0 + h, \quad x_2 = x_0 + 2h, \dots, \quad x_n = x_0 + nh$$

Euler’s method for a system in the form of Equation 7.46 is described by

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\mathbf{f}(x_i, \mathbf{u}_i), \quad i = 0, 1, 2, \dots, n-1 \tag{7.47}$$

The user-defined function `EulerODESystem` uses Euler’s method as outlined in Equation 7.47 to estimate the solution vector of a system of initial-value problems in the form of Equation 7.46.


```

function u = EulerODESystem(f,x,u0)
%
% EulerODESystem uses Euler's method to solve a system of first-order
% initial-value problems in the form u' = f(x,u), u(x0) = u0.
%
% u = EulerODESystem(f,x,u0), where
%
% f is an anonymous m-dim. vector function representing f(x,u),
% x is an (n+1)-dim. vector representing the mesh points,
% u0 is an m-dim. vector representing the initial state vector,
%
% u is an m-by-(n+1) matrix, each column the vector of solution
% estimates at a mesh point.
%
u(:,1) = u0; % The first column is set to be the initial vector u0
h = x(2) - x(1); n = length(x);
for i = 1:n-1,
    u(:,i+1) = u(:,i)+h*f(x(i),u(:,i));
end

```

EXAMPLE 7.11: EULER'S METHOD FOR SYSTEMS

Consider the third-order IVP in Example 7.9:

$$3y''' - y'' - 5y' - 3y = e^{-x/2}, \quad y(0) = 0, \quad y'(0) = -1, \quad y''(0) = 1, \quad 0 \leq x \leq 1$$

Using Euler's method for systems, with step size $h = 0.1$, find an estimate for $y_2 = y(0.2)$. Confirm by executing the user-defined function `EulerODESystem`.

Solution

In Example 7.9, the IVP was transformed into the standard form of a system of first-order IVPs as

$$\mathbf{u}' = \mathbf{f}(x, \mathbf{u}), \quad \mathbf{u} = \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} y \\ y' \\ y'' \end{Bmatrix}, \quad \mathbf{f} = \begin{Bmatrix} u_2 \\ u_3 \\ \frac{1}{3}[u_3 + 5u_2 + 3u_1 + e^{-x/2}] \end{Bmatrix}, \quad \mathbf{u}_0 = \begin{Bmatrix} 0 \\ -1 \\ 1 \end{Bmatrix} = \begin{Bmatrix} u_1(0) \\ u_2(0) \\ u_3(0) \end{Bmatrix}$$

To find $y(0.2)$, we need to find the solution vector $\mathbf{u}_2 = \mathbf{u}(0.2)$ and then extract its first component, which is $y(0.2)$. By Equation 7.47,

$$\mathbf{u}_1 = \mathbf{u}_0 + h\mathbf{f}(x_0, \mathbf{u}_0)$$

But

$$\mathbf{f}(x_0, \mathbf{u}_0) = \begin{Bmatrix} u_2(x_0) \\ u_3(x_0) \\ \frac{1}{3}[u_3(x_0) + 5u_2(x_0) + 3u_1(x_0) + e^{-x_0/2}] \end{Bmatrix} \Big|_{x_0=0} = \begin{Bmatrix} -1 \\ 1 \\ \frac{1}{3}(1 - 5(1) + 3(0) + 1) \end{Bmatrix} = \begin{Bmatrix} -1 \\ 1 \\ -1 \end{Bmatrix}$$

Therefore,

$$\mathbf{u}_1 = \mathbf{u}_0 + hf(x_0, \mathbf{u}_0) = \begin{Bmatrix} 0 \\ -1 \\ 1 \end{Bmatrix} + 0.1 \begin{Bmatrix} -1 \\ 1 \\ -1 \end{Bmatrix} = \begin{Bmatrix} -0.1 \\ -0.9 \\ 0.9 \end{Bmatrix}$$

In the next step, $\mathbf{u}_2 = \mathbf{u}_1 + hf(x_1, \mathbf{u}_1)$ where

$$\mathbf{f}(x_1, \mathbf{u}_1) = \begin{Bmatrix} u_2(0.1) \\ u_3(0.1) \\ \frac{1}{3}[u_3(0.1) + 5u_2(0.1) + 3u_1(0.1) + e^{-0.1/2}] \end{Bmatrix} = \begin{Bmatrix} -0.9 \\ 0.9 \\ \frac{1}{3}(0.9 + 5(-0.9) + 3(-0.1) + e^{-0.1/2}) \end{Bmatrix} = \begin{Bmatrix} -0.9 \\ 0.9 \\ -0.9829 \end{Bmatrix}$$

Therefore,

$$\mathbf{u}_2 = \mathbf{u}_1 + hf(x_1, \mathbf{u}_1) = \begin{Bmatrix} -0.1 \\ -0.9 \\ 0.9 \end{Bmatrix} + 0.1 \begin{Bmatrix} -0.9 \\ 0.9 \\ -0.9829 \end{Bmatrix} = \begin{Bmatrix} \boxed{-0.1900} \\ -0.8100 \\ 0.8017 \end{Bmatrix}$$

The first component represents $y(0.2)$, thus $y(0.2) = -0.19$. The results may be confirmed in MATLAB as follows:

```
>> f = @(x,u) ([u(2);u(3);(u(3)+5*u(2)+3*u(1)+exp(-x/2))/3]);
>> x = 0:0.1:1; % 11 mesh points
>> u0 = [0;-1;1];
>> u = EulerODESystem(f,x,u0); % Returns a 3-by-11 matrix
```

This is a 3×11 matrix because there are three state variables and 11 mesh points created. The first row contains the solution estimates for y at the 11 mesh points, the second row y' and the third row y'' . Since we are interested in $y(0.2)$, it is the first row of u that we must retain. In particular,

```
>> u(1,:)
ans =
Columns 1 through 9
    0   -0.1000   -0.1900  -0.2710  -0.3440  -0.4099  -0.4698  -0.5245  -0.5751
Columns 10 through 11
-0.6226  -0.6680
```

The boxed value of $y(0.2)$ agrees with our hand calculations. Knowing the exact value is -0.181324 (truncated), the % relative error associated with our estimate is 4.785%. In order to verify the hand-calculated vectors \mathbf{u}_1 and \mathbf{u}_2 , we proceed as follows. Since the

first column of \mathbf{u} represents the initial state vector, we will extract the second and third columns to verify the hand calculations:

```
>> u(:, [2 3])
```

```
ans =
```

```
-0.1000    -0.1900
-0.9000    -0.8100
 0.9000     0.8017
```

As expected, these exactly match our earlier findings.

7.6.2.2 Heun's Method for Systems

Heun's method for a system in the form of Equation 7.46 is defined as

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{1}{2}h(\mathbf{k}_1 + \mathbf{k}_2), \quad i = 0, 1, 2, \dots, n-1 \quad (7.48)$$

where

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(x_i, \mathbf{u}_i) \\ \mathbf{k}_2 &= \mathbf{f}(x_i + h, \mathbf{u}_i + h\mathbf{k}_1) \end{aligned}$$

The user-defined function `HeunODESystem` uses Heun's method as outlined in Equation 7.48 to estimate the solution vector of a system of initial-value problems in the form of Equation 7.46.

```
function u = HeunODESystem(f,x,u0)
%
% HeunODESystem uses Heun's method to solve a system of first-order
% initial-value problems in the form u' = f(x,u), u(x0) = u0.
%
% u = HeunODESystem(f,x,u0), where
%
% f is an anonymous m-dim. vector function representing f(x,u),
% x is an (n+1)-dim. vector representing the mesh points,
% u0 is an m-dim. vector representing the initial state vector,
%
% u is an m-by-(n+1) matrix, each column the vector of solution
% estimates at a mesh point.
%
u(:,1) = u0; % The first column is set to be the initial vector u0
h = x(2) - x(1); n = length(x);
for i = 1:n-1,
    k1 = f(x(i),u(:,i));
    k2 = f(x(i)+h,u(:,i)+h*k1);
    u(:,i+1) = u(:,i)+h*(k1+k2)/2;
end
```

EXAMPLE 7.12: HEUN'S METHOD FOR SYSTEMS

In Example 7.11, find an estimate for $y(0.2)$ by executing the user-defined function `HeunODESystem`.

Solution

```
>> f = @(x,u) ([u(2);u(3);(u(3)+5*u(2)+3*u(1)+exp(-x/2))/3]);
>> x = 0:0.1:1;
>> u0 = [0;-1;1];
>> u = HeunODESystem(f,x,u0);
>> u(1,3)      % y(0.2) is the 3rd entry in the first row

ans =
```

-0.1810

The boxed value is $y(0.2) = -0.1810$. Recall from Example 7.11 that the (truncated) exact value is -0.181324 . Therefore, the % relative error here is 0.18% as opposed to 4.785% for Euler. As expected, Heun's method returns a more accurate approximation than Euler.

7.6.2.3 Classical RK4 Method for Systems

The classical fourth-order Runge–Kutta method for a system in the form of Equation 7.46 is defined as

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{1}{6}h(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4), \quad i = 0, 1, 2, \dots, n-1 \quad (7.49)$$

where

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(x_i, \mathbf{u}_i) \\ \mathbf{k}_2 &= \mathbf{f}\left(x_i + \frac{1}{2}h, \mathbf{u}_i + \frac{1}{2}h\mathbf{k}_1\right) \\ \mathbf{k}_3 &= \mathbf{f}\left(x_i + \frac{1}{2}h, \mathbf{u}_i + \frac{1}{2}h\mathbf{k}_2\right) \\ \mathbf{k}_4 &= \mathbf{f}(x_i + h, \mathbf{u}_i + h\mathbf{k}_3) \end{aligned}$$

The user-defined function `RK4System` uses the fourth-order Runge–Kutta method as outlined in Equation 7.49 to estimate the solution vector of a system of initial-value problems in the form of Equation 7.46.

```
function u = RK4System(f,x,u0)
%
% RK4System uses RK4 method to solve a system of first-order
% initial-value problems in the form u' = f(x,u), u(x0) = u0.
%
% u = RK4System(f,x,u0), where
%
% f is an anonymous m-dim. vector function representing f(x,u),
% x is an (n+1)-dim. vector representing the mesh points,
% u0 is an m-dim. vector representing the initial state vector,
```

```

%
%           u is an m-by-(n+1) matrix, each column the vector of solution
%           estimates at a mesh point.
%
u(:,1) = u0; % The first column is set to be the initial vector u0
h = x(2) - x(1); n = length(x);
for i = 1:n-1,
    k1 = f(x(i),u(:,i));
    k2 = f(x(i)+h/2,u(:,i)+h*k1/2);
    k3 = f(x(i)+h/2,u(:,i)+h*k2/2);
    k4 = f(x(i)+h,u(:,i)+h*k3);
    u(:,i+1) = u(:,i)+h*(k1+2*k2+2*k3+k4)/6;
end

```

EXAMPLE 7.13: RK4 METHOD FOR SYSTEMS

Reconsider Example 7.11. Using RK4 method for systems, with $h = 0.1$, find an estimate for $y_1 = y(0.1)$. Confirm by executing the user-defined function `RK4System`.

Solution

Recall

$$\mathbf{u}' = \mathbf{f}(x, \mathbf{u}), \quad \mathbf{u} = \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} y \\ y' \\ y'' \end{Bmatrix}, \quad \mathbf{f}(x, \mathbf{u}) = \begin{Bmatrix} u_2 \\ u_3 \\ \frac{1}{3}[u_3 + 5u_2 + 3u_1 + e^{-x/2}] \end{Bmatrix}, \quad \mathbf{u}_0 = \begin{Bmatrix} u_1(0) \\ u_2(0) \\ u_3(0) \end{Bmatrix} = \begin{Bmatrix} 0 \\ -1 \\ 1 \end{Bmatrix}$$

We first need to find $\mathbf{u}_1 = \mathbf{u}_0 + \frac{1}{6}h(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$.

$$\mathbf{k}_1 = \mathbf{f}(x_0, \mathbf{u}_0) = \begin{Bmatrix} u_2(0) \\ u_3(0) \\ \frac{1}{3}[u_3(0) + 5u_2(0) + 3u_1(0) + e^{-0/2}] \end{Bmatrix} = \begin{Bmatrix} -1 \\ 1 \\ \frac{1}{3}(1 + 5(-1) + 3(0) + 1) \end{Bmatrix} = \begin{Bmatrix} -1 \\ 1 \\ -1 \end{Bmatrix}$$

To calculate $\mathbf{k}_2 = \mathbf{f}\left(x_0 + \frac{1}{2}h, \mathbf{u}_0 + \frac{1}{2}h\mathbf{k}_1\right) = \mathbf{f}\left(0.05, \mathbf{u}_0 + \frac{1}{2}h\mathbf{k}_1\right)$, we first find

$$\mathbf{u}_0 + \frac{1}{2}h\mathbf{k}_1 = \begin{Bmatrix} 0 \\ -1 \\ 1 \end{Bmatrix} + 0.05 \begin{Bmatrix} -1 \\ 1 \\ -1 \end{Bmatrix} = \begin{Bmatrix} -0.05 \\ -0.95 \\ 0.95 \end{Bmatrix}$$

Then,

$$\mathbf{k}_2 = \mathbf{f}\left(0.05, \mathbf{u}_0 + \frac{1}{2}h\mathbf{k}_1\right) = \begin{Bmatrix} -0.95 \\ 0.95 \\ \frac{1}{3}[0.95 + 5(-0.95) + 3(-0.05) + e^{-0.05/2}] \end{Bmatrix} = \begin{Bmatrix} -0.95 \\ 0.95 \\ -0.9916 \end{Bmatrix}$$

To calculate $\mathbf{k}_3 = \mathbf{f}(x_0 + \frac{1}{2}h, \mathbf{u}_0 + \frac{1}{2}h\mathbf{k}_2) = \mathbf{f}(0.05, \mathbf{u}_0 + \frac{1}{2}h\mathbf{k}_2)$, we first find

$$\mathbf{u}_0 + \frac{1}{2}h\mathbf{k}_2 = \begin{Bmatrix} 0 \\ -1 \\ 1 \end{Bmatrix} + 0.05 \begin{Bmatrix} -0.95 \\ 0.95 \\ -0.9916 \end{Bmatrix} = \begin{Bmatrix} -0.0475 \\ -0.9525 \\ 0.9504 \end{Bmatrix}$$

Then,

$$\mathbf{k}_3 = \mathbf{f}\left(0.05, \mathbf{u}_0 + \frac{1}{2}h\mathbf{k}_2\right) = \begin{Bmatrix} -0.9525 \\ 0.9504 \\ \frac{1}{3}[0.9504 + 5(-0.9525) + 3(-0.0475) + e^{-0.05/2}] \end{Bmatrix} = \begin{Bmatrix} -0.9525 \\ 0.9504 \\ -0.9931 \end{Bmatrix}$$

To find $\mathbf{k}_4 = \mathbf{f}(x_0 + h, \mathbf{u}_0 + h\mathbf{k}_3) = \mathbf{f}(0.1, \mathbf{u}_0 + h\mathbf{k}_3)$, we first calculate

$$\mathbf{u}_0 + h\mathbf{k}_3 = \begin{Bmatrix} 0 \\ -1 \\ 1 \end{Bmatrix} + 0.1 \begin{Bmatrix} -0.9525 \\ 0.9504 \\ -0.9931 \end{Bmatrix} = \begin{Bmatrix} -0.0953 \\ -0.9050 \\ 0.9007 \end{Bmatrix}$$

Then,

$$\mathbf{k}_4 = \mathbf{f}(0.1, \mathbf{u}_0 + h\mathbf{k}_3) = \begin{Bmatrix} -0.9050 \\ 0.9007 \\ \frac{1}{3}[0.9007 + 5(-0.9050) + 3(-0.0953) + e^{-0.1/2}] \end{Bmatrix} = \begin{Bmatrix} -0.9050 \\ 0.9007 \\ -0.9863 \end{Bmatrix}$$

Finally,

$$\mathbf{u}_1 = \mathbf{u}_0 + \frac{1}{6}h(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) = \begin{Bmatrix} \boxed{-0.0952} \\ -0.9050 \\ 0.9007 \end{Bmatrix}$$

Therefore, our estimate is $y(0.1) = -0.0952$. The result may be confirmed in MATLAB as follows:

```
>> f = @(x,u) ([u(2);u(3);(u(3)+5*u(2))+3*u(1)+exp(-x/2))/3]);
>> x = 0:0.1:1;
>> u0 = [0;-1;1];
>> u = RK4System(f,x,u0);
>> u(1,2) % y(0.1) is the 2nd entry in the first row
```

ans =

$\boxed{-0.0952}$

The boxed value agrees with the hand calculations.

EXAMPLE 7.14: RK4 METHOD FOR SYSTEMS

The pendulum system in Figure 7.5 consists of a uniform thin rod of length l and a concentrated mass m at its tip. The friction at the pivot causes the system to be damped. When the angular displacement θ is not very small, the system is described by a non-linear model in the form

$$\frac{3}{4}ml^2\ddot{\theta} + 0.18\dot{\theta} + \frac{1}{2}mgl\sin\theta = 0$$

Assume, in consistent physical units, that $ml^2 = 1.28$, $\frac{g}{l} = 7.45$.

1. Transform the model into a system of first-order initial-value problems.
2. Write a MATLAB script that utilizes the user-defined function `RK4System` to solve the system in (1). Two sets of initial conditions are to be considered: (1) $\theta(0) = 15^\circ$, $\dot{\theta}(0) = 0$ and (2) $\theta(0) = 30^\circ$, $\dot{\theta}(0) = 0$. The file must return the plots of the two angular displacements corresponding to the two sets of initial conditions versus $0 \leq t \leq 5$ in the same graph. Angle measures must be converted to radians. Use at least 100 points for plotting purposes.

Solution

1. First note that

$$mgl = (ml^2)\left(\frac{g}{l}\right) = (1.28)(7.45) = 9.5360$$

Therefore, the equation of motion is rewritten as

$$\frac{3}{4}(1.28)\ddot{\theta} + 0.18\dot{\theta} + \frac{1}{2}(9.5360)\sin\theta = 0 \Rightarrow 0.96\ddot{\theta} + 0.18\dot{\theta} + 4.7680\sin\theta = 0$$

There are two state variables: $u_1 = \theta$, $u_2 = \dot{\theta}$. The state-variable equations are formed as

$$\begin{aligned} \dot{u}_1 &= u_2 \\ \dot{u}_2 &= -0.1875u_2 - 4.9667\sin u_1 \end{aligned}$$

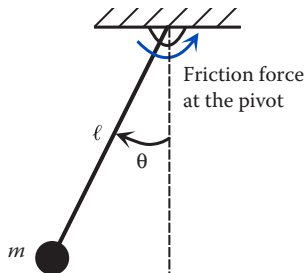


FIGURE 7.5
Pendulum system.

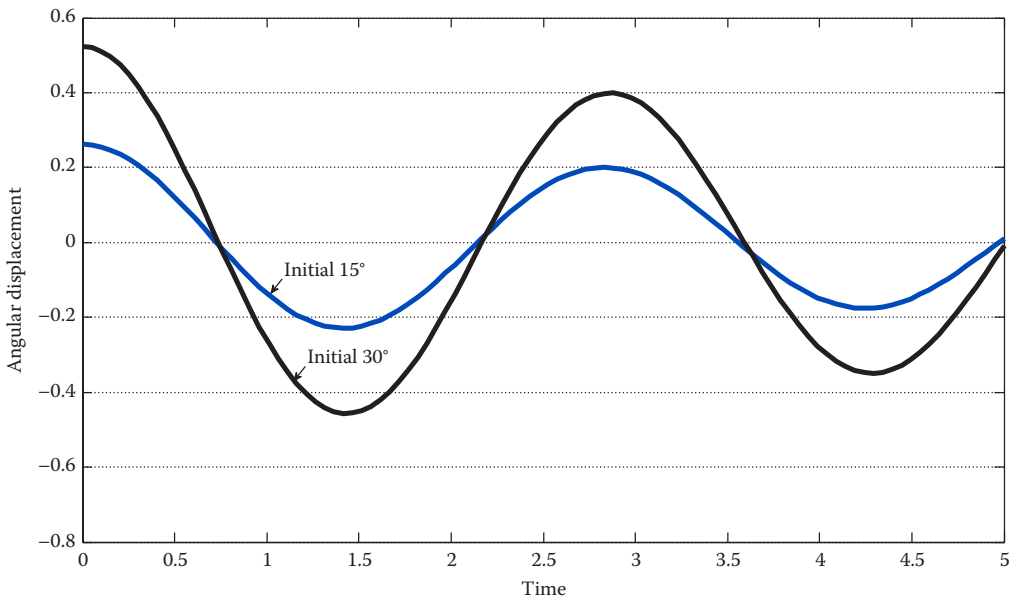


FIGURE 7.6

Angular displacements of the rod for two sets of initial conditions.

The two sets of initial conditions are

$$u_1(0) = 15(\pi/180) \text{ radians} \quad \text{and} \quad u_1(0) = 30(\pi/180) \text{ radians}$$

$$u_2(0) = 0 \quad \quad \quad u_2(0) = 0$$

In vector form,

$$\dot{\mathbf{u}} = \mathbf{f}(t, \mathbf{u}), \quad \mathbf{u} = \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix}, \quad \mathbf{f}(t, \mathbf{u}) = \begin{Bmatrix} u_2 \\ -0.1875u_2 - 4.9667\sin u_1 \end{Bmatrix}, \quad \mathbf{u}_0 = \begin{Bmatrix} \frac{15}{180}\pi \\ 0 \end{Bmatrix} \text{ and } \begin{Bmatrix} \frac{30}{180}\pi \\ 0 \end{Bmatrix}$$

2.

```
t = linspace(0,5); u0 = [15*pi/180;0];
f = @(t,u) ([u(2);-0.1875*u(2)-4.9667*sin(u(1))]);
uRK4 = RK4System(f,t,u0);
theta1 = uRK4(1,:); plot(t,theta1)
hold on
u0 = [30*pi/180;0];
uRK4 = RK4System(f,t,u0);
theta2 = uRK4(1,:); plot(t,theta2)    % Figure 7.6
```

7.7 Stability

Numerical stability is a desirable property of numerical methods. A numerical method is stable if errors incurred in one step do not magnify in later steps. Stability analysis of a

numerical method often boils down to the error analysis of the method when applied to a basic initial-value problem, which serves as a model, in the form

$$y' = -\lambda y, \quad y(0) = y_0, \quad \lambda = \text{const} > 0 \quad (7.50)$$

Since the exact solution $y(x) = y_0 e^{-\lambda x}$ exponentially decays toward zero, it is desired that the error also approaches zero as x gets sufficiently large. If a method is unstable when applied to this model, it is likely to have difficulty when applied to other initial-value problems.

7.7.1 Euler's Method

Suppose Euler's method with step size h is applied to this model. This means each subinterval $[x_i, x_{i+1}]$ has length $h = x_{i+1} - x_i$. Noting that $f(x, y) = -\lambda y$, the solution estimate y_{i+1} at x_{i+1} provided by Euler's method is

$$y_{i+1} = y_i + hf(x_i, y_i) = y_i - h\lambda y_i = (1 - \lambda h)y_i \quad (7.51)$$

At x_{i+1} , the exact solution is

$$y_{i+1}^{\text{Exact}} = y_0 e^{-\lambda x_{i+1}} \stackrel{x_{i+1} = x_i + h}{=} \left[y_0 e^{-\lambda x_i} \right] e^{-\lambda h} = y_i^{\text{Exact}} e^{-\lambda h} \quad (7.52)$$

Comparison of Equations 7.51 and 7.52 reveals that $1 - \lambda h$ in the computed solution is an approximation for $e^{-\lambda h}$ in the exact solution*. From Equation 7.51, it is also observed that error will not be magnified if $|1 - \lambda h| < 1$. This implies that the numerical method (in this case, Euler's method) is stable if

$$|1 - \lambda h| < 1 \Rightarrow -1 < 1 - \lambda h < 1 \Rightarrow 0 < \lambda h < 2 \quad (7.53)$$

which acts as a stability criterion for Euler's method when applied to the IVP in Equation 7.50. Equation 7.53 describes a region of absolute stability for Euler's method. The wider the region of stability, the less limitation imposed on the step size h .

7.7.2 Euler's Implicit Method

Euler's implicit method is described by

$$y_{i+1} = y_i + hf(x_{i+1}, y_{i+1}), \quad i = 0, 1, 2, \dots, n-1 \quad (7.54)$$

so that y_{i+1} appears on both sides of the equation. As with other implicit methods discussed so far, y_{i+1} is normally found numerically via the methods of [Chapter 3](#), and can only be solved analytically if the function $f(x, y)$ has a simple structure. This is certainly the case when applied to the model in Equation 7.50 where $f(x, y) = -\lambda y$.

* Taylor series of $e^{-\lambda h}$ is $e^{-\lambda h} = 1 - \lambda h + \frac{1}{2!}(\lambda h)^2 - \dots$ so that for small λh , we have $e^{-\lambda h} \cong 1 - \lambda h$.

$$y_{i+1} = y_i + hf(x_{i+1}, y_{i+1}) = y_i + h(-\lambda y_{i+1}) \Rightarrow y_{i+1} = \frac{1}{1 + \lambda h} y_i$$

Therefore, Euler's implicit method is stable if

$$\left| \frac{1}{1 + \lambda h} \right| < 1 \Rightarrow |1 + \lambda h| > 1 \stackrel{h, \lambda > 0}{\Rightarrow} h > 0$$

which implies it is stable regardless of the step size.

EXAMPLE 7.15: STABILITY OF EULER'S METHODS

Consider the initial-value problem

$$y' = -4y, \quad y(0) = 2, \quad 0 \leq x \leq 5$$

1. Solve using Euler's method with step size $h = 0.3$ and again with $h = 0.55$, plot the estimated solutions together with the exact solution $y(x) = 2e^{-4x}$, and discuss stability.
2. Repeat using the Euler's implicit method, Equation 7.54.

Solution

1. Comparing the IVP at hand with the model in Equation 7.50, we have $\lambda = 4$. The stability criterion for Euler's method is

$$4h < 2 \Rightarrow h < \frac{1}{2}$$

Therefore, Euler's method is stable if $h < 0.5$, and is unstable otherwise. As observed from the first plot in [Figure 7.7](#), Euler's method with $h = 0.3 < 0.5$ produces estimates that closely follow the exact solution, while those generated by $h = 0.55 > 0.5$ grow larger in each step, indicating instability of the method.

2. As mentioned earlier, when applied to a simple IVP such as the one here, Euler's implicit method is stable for all step sizes. The second plot in [Figure 7.7](#) clearly shows that the errors associated with both step sizes decay to zero as x increases, indicating stability. The following MATLAB script completely generates the results illustrated in [Figure 7.7](#).

```

y0 = 2; f = @(x,y)(-4*y);
yExact = matlabFunction(dsolve('Dy+4*y=0','y(0)=2','x')); % Exact solution
h1 = 0.3; x1 = 0:h1:5; h2 = 0.55; x2 = 0:h2:5;
% Euler's method with h = 0.3 and h = 0.55
y1 = EulerODE(f,x1,y0); y2 = EulerODE(f,x2,y0);

y1I(1) = y0;
for i = 2:length(x1),
    y1I(i) = y1I(i-1)/(1+4*h1); % Implicit Euler with h = 0.3
end
y2I(1) = y0;
for i = 2:length(x2),
    y2I(i) = y2I(i-1)/(1+4*h2); % Implicit Euler with h = 0.55
end

```

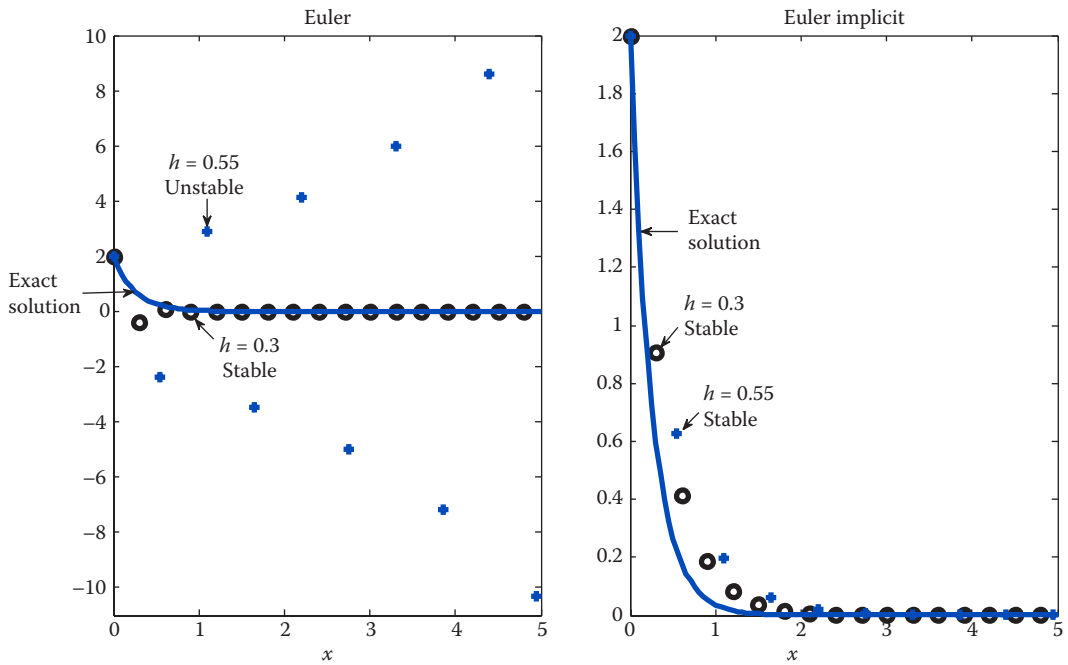


FIGURE 7.7
Stability analysis of Euler's and Euler's implicit methods in Example 7.15.

```

x3 = linspace(0,5);
ye = zeros(100,1);
for i = 1:100,
    ye(i) = yExact(x3(i));
end
subplot (1,2,1), plot(x1,y1,'o',x2,y2,'+',x3,ye,'-') % Figure 7.7
title('Euler')
subplot (1,2,2), plot(x1,y1I,'o',x2,y2I,'+',x3,ye,'-')
title('Euler implicit')

```

7.8 Stiff Differential Equations

In many engineering applications, such as chemical kinetics, mass–spring–damper systems, and control system analysis, we encounter systems of differential equations whose solutions contain terms with magnitudes that vary at considerably different rates. Such differential equations are known as stiff differential equations. For example, if a solution includes the terms e^{-at} and e^{-bt} , with $a, b > 0$, where the magnitude of a is much larger than b , then e^{-at} decays to zero at a much faster rate than e^{-bt} does. In the presence of a rapidly decaying transient solution, some numerical methods become unstable unless the step size is unreasonably small. Explicit methods generally are subjected to this stability constraint, which requires them to use an extremely small step size for accuracy. But using a very small step size not only substantially increases the number of operations to find a

solution, but it also causes the round-off error to grow, thus leading to limited accuracy. Implicit methods, on the other hand, are free of stability restrictions and are therefore preferred for solving stiff differential equations.

EXAMPLE 7.16: STIFF SYSTEM OF ODEs

Consider

$$\begin{aligned} \dot{v} &= 790v - 1590w & \text{subject to} & & v_0 &= v(0) = 1 \\ \dot{w} &= 793v - 1593w & & & w_0 &= w(0) = -1 \end{aligned}$$

The exact solution of this system can be found in closed form, as

$$\begin{aligned} v(t) &= \frac{3180}{797} e^{-3t} - \frac{2383}{797} e^{-800t} \\ w(t) &= \frac{1586}{797} e^{-3t} - \frac{2383}{797} e^{-800t} \end{aligned}$$

Exact solution at t = 0.1:

$$\begin{aligned} v_1 = v(0.1) &= \frac{3180}{797} e^{-3(0.1)} - \frac{2383}{797} e^{-800(0.1)} = 2.955836815 \\ w_1 = w(0.1) &= \frac{1586}{797} e^{-3(0.1)} - \frac{2383}{797} e^{-800(0.1)} = 1.474200374 \end{aligned}$$

Euler's method with h = 0.1: We first express the system in vector form

$$\dot{\mathbf{u}} = \mathbf{f}(t, \mathbf{u}), \quad \mathbf{u} = \begin{Bmatrix} v \\ w \end{Bmatrix}, \quad \mathbf{f} = \begin{Bmatrix} 790v - 1590w \\ 793v - 1593w \end{Bmatrix}, \quad \mathbf{u}_0 = \begin{Bmatrix} 1 \\ -1 \end{Bmatrix}$$

Then, by Euler's method

$$\mathbf{u}_1 = \mathbf{u}_0 + h\mathbf{f}(x_0, \mathbf{u}_0) = \begin{Bmatrix} 1 \\ -1 \end{Bmatrix} + 0.1 \begin{Bmatrix} 790v_0 - 1590w_0 \\ 793v_0 - 1593w_0 \end{Bmatrix} = \begin{Bmatrix} 239 \\ 237.6 \end{Bmatrix} \Rightarrow \begin{aligned} v_1 &= 239 \\ w_1 &= 237.6 \end{aligned}$$

which are totally incorrect. Euler's method is an explicit method, and as such, it is unstable unless a very small step size is selected. The main contribution to numerical instability is by the rapidly decaying e^{-800t} . Reduction of the step size can dramatically improve accuracy.

Euler's method with h = 0.001 results in

$$\mathbf{u}_1 = \mathbf{u}_0 + h\mathbf{f}(x_0, \mathbf{u}_0) = \begin{Bmatrix} 1 \\ -1 \end{Bmatrix} + 0.001 \begin{Bmatrix} 790v_0 - 1590w_0 \\ 793v_0 - 1593w_0 \end{Bmatrix} = \begin{Bmatrix} 3.380 \\ 1.386 \end{Bmatrix} \Rightarrow \begin{aligned} v_1 &= 3.380 \\ w_1 &= 1.386 \end{aligned}$$

which are much more accurate than before. Further reduction to $h = 0.0001$ allows round-off error to grow, causing the estimates to be much less accurate.

Euler's implicit method with $h = 0.1$: Using the vector form of Equation 7.54,

$$\mathbf{u}_1 = \mathbf{u}_0 + h\mathbf{f}(x_1, \mathbf{u}_1) = \begin{Bmatrix} 1 \\ -1 \end{Bmatrix} + 0.1 \begin{Bmatrix} 790v_1 - 1590w_1 \\ 793v_1 - 1593w_1 \end{Bmatrix} \Rightarrow \begin{matrix} v_1 = 1 + 79v_1 - 159w_1 \\ w_1 = -1 + 79.3v_1 - 159.3w_1 \end{matrix} \Rightarrow \begin{matrix} v_1 = 3.032288699 \\ w_1 = 1.493827160 \end{matrix}$$

These are closer to the exact values than those generated by Euler's method with a much smaller h .

7.9 MATLAB Built-In Functions for Solving Initial-Value Problems

There are several MATLAB built-in functions designed for solving a single first-order initial-value problem, as well as a system of first-order initial-value problems. These are known as ODE solvers and include `ode23`, `ode45`, and `ode113` for non-stiff equations, and `ode15s` for stiff equations. Most of these built-in functions use highly developed techniques that allow for the use of an optimal step size, or in some cases, adjusted step size for error minimization in each step.

7.9.1 Non-Stiff Equations

We will first show how the built-in functions can be used to solve a single first-order IVP, then extend their applications to systems of first-order IVPs.

7.9.2 A Single First-Order IVP

The initial-value problem is

$$\dot{y} = f(t, y), \quad y(t_0) = y_0 \quad (7.55)$$

Note that, as mentioned before, we are using t in place of x since in most applied problems time is the independent variable. For non-stiff equations, MATLAB built-in ODE solvers are `ode23`, `ode45`, and `ode113`.

`ode23` is a single-step method based on second- and third-order Runge–Kutta methods. As always, MATLAB help file should be regularly consulted for detailed information.

`ode23` Solve non-stiff differential equations, low order method.

`[TOUT, YOUT] = ode23(ODEFUN, TSPAN, Y0)` with `TSPAN = [T0 TFINAL]` integrates the system of differential equations $y' = f(t, y)$ from time `T0` to `TFINAL` with initial conditions `Y0`. `ODEFUN` is a function handle. For a scalar `T` and a vector `Y`, `ODEFUN(T, Y)` must return a column vector corresponding to $f(t, y)$. Each row in the solution array `YOUT` corresponds to a time returned in the column vector `TOUT`. To obtain solutions at specific times `T0, T1, ..., TFINAL` (all increasing or all decreasing), use `TSPAN = [T0 T1 ... TFINAL]`.

Note that if `TSPAN` has only two elements (the left and right endpoints for t), then vector `YOUT` will contain solutions calculated by `ode23` at all steps.

ode45 is a single-step method based on fourth- and fifth-order Runge–Kutta methods. Refer to MATLAB help file for more detailed information.

ode45 Solve non-stiff differential equations, medium order method.

The function call is

$$[TOUT, YOUT] = \text{ode45}(\text{ODEFUN}, TSPAN, Y0)$$

with all input and output argument descriptions as in ode23 above. ode45 can handle most initial-value problems and *should be the first solver attempted for a given problem*.

ode113 is a multistep method based on Adams–Bashforth–Moulton (ABM) methods. Refer to MATLAB help file for more detailed information.

ode113 Solve non-stiff differential equations, variable order method.

The function call is

$$[TOUT, YOUT] = \text{ode113}(\text{ODEFUN}, TSPAN, Y0)$$

with all input and output argument descriptions as in ode23 and ode45 above.

EXAMPLE 7.17: ODE23, ODE45

Write a MATLAB script that solves the following initial-value problem using ode23 and ode45 solvers and returns a table that includes the solution estimates at $t=1:0.2:3$, as well as the exact solution and the % relative error for both methods at each point.

$$t\dot{y} + y = e^{-t}, \quad y(1) = 1, \quad 1 \leq t \leq 3$$

Solution

```
disp('  t          yode23      yode45      yExact      e_23      e_45 ')
```

```
t = 1:0.2:3; y0 = 1;
```

```
f = @(t,y)((exp(-t)-y)/t);
```

```
[t,y23] = ode23(f,t,y0);
```

```
[t,y45] = ode45(f,t,y0);
```

```
yExact = matlabFunction(dsolve('t*Dy+y=exp(-t)','y(1)=1'));
```

```
for k=1:length(t),
```

```
    t_coord = t(k);
```

```
    yode23 = y23(k);
```

```
    yode45 = y45(k);
```

```
    yEx = yExact(t(k));
```

```
    e_23 = (yEx - yode23)/yEx*100;
```

```
    e_45 = (yEx - yode45)/yEx*100;
```

```
fprintf('%6.2f %11.6f%11.6f %11.6f %11.6f %11.8f\n',t_coord,yode23,
```

```
yode45,yEx,e_23,e_45)
```

```
end
```

t	yode23	yode45	yExact	e_23	e_45
1.00	1.000000	1.000000	1.000000	0.000000	0.00000000
1.20	0.888872	0.888904	0.888904	0.003589	-0.00000000
1.40	0.800866	0.800916	0.800916	0.006202	-0.00000001
1.60	0.728684	0.728739	0.728739	0.007554	-0.00000001
1.80	0.668045	0.668100	0.668100	0.008314	-0.00000001
2.00	0.616218	0.616272	0.616272	0.008770	-0.00000001
2.20	0.571347	0.571398	0.571398	0.009058	-0.00000001
2.40	0.532101	0.532151	0.532151	0.009250	-0.00000001
2.60	0.497494	0.497541	0.497541	0.009383	-0.00000001
2.80	0.466766	0.466810	0.466810	0.009479	-0.00000001
3.00	0.439323	0.439364	0.439364	0.009470	-0.00000001

As expected, ode45 produces much more accurate estimates but is often slower than ode23.

7.9.3 Setting ODE Solver Options

The `ode23`, `ode45`, and `ode113` functions accept an optional input argument called an “options structure” that allows many properties of the solution method to be specified. Two examples of such properties are the relative tolerance and the minimum step size. The `odeset` function creates these options structures.

`odeset` Create/alter ODE OPTIONS structure.

`OPTIONS = odeset('NAME1',VALUE1,'NAME2',VALUE2,...)` creates an integrator options structure `OPTIONS` in which the named properties have the specified values. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names.

A complete list of the various properties and their descriptions are available in the help function. A default options structure is created using

```
>> options = odeset;
```

The default relative tolerance `RelTol` is 10^{-3} . `RelTol` can either be specified upon creation of the options structure

```
>> options = odeset('RelTol', 1e-7);
```

or can be changed by including the current options as the first input argument.

```
>> options = odeset(options, 'RelTol', 1e-6);
```

The option structure is then specified as the fourth input argument to the ODE solvers.

```
>> [t,y] = ode45(@myfunc,t,y0,options);
```

Therefore, we can trade speed for accuracy by specifying the relative tolerance. Other parameters may also be specified.

7.9.4 A System of First-Order IVPs

The ODE solvers discussed here can also handle systems of first-order initial-value problems.

EXAMPLE 7.18: ode45 FOR A SYSTEM

Using ode45 (with step size of 0.1), solve the following system:

$$\begin{aligned} \dot{u}_1 &= u_2 \\ \dot{u}_2 &= \frac{2}{3}(-2u_2 - tu_1 + \cos t) \end{aligned} \quad \text{subject to initial conditions} \quad \begin{aligned} u_1(0) &= 1 \\ u_2(0) &= -1 \end{aligned}$$

Solution

We first express the system in vector form

$$\dot{\mathbf{u}} = \mathbf{f}(t, \mathbf{u}), \quad \mathbf{u} = \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix}, \quad \mathbf{f} = \begin{Bmatrix} u_2 \\ \frac{2}{3}(-2u_2 - tu_1 + \cos t) \end{Bmatrix}, \quad \mathbf{u}_0 = \begin{Bmatrix} 1 \\ -1 \end{Bmatrix}$$

```
u0 = [1;-1]; t = 0:0.1:1;
f = @(t,u) ([u(2); 2*(-2*u(2)-t*u(1)+cos(t))/3]);
[t,u45] = ode45(f,t,u0)
```

t =

```
0
0.1000
0.2000
0.3000
0.4000
0.5000
0.6000
0.7000
0.8000
0.9000
1.0000
```

% u45 has two columns: First column is u1, second column is u2

u45 =

```
1.0000    -1.0000
0.9095    -0.8159
0.8359    -0.6605
0.7765    -0.5302
0.7291    -0.4219
0.6915    -0.3330
```



```

0.6619    -0.2615
0.6387    -0.2056
0.6203    -0.1639
0.6055    -0.1348
0.5930    -0.1173

```

7.9.5 Stiff Equations

The ODE solver `ode15s` can be used to solve stiff equations. `ode15s` is a multistep, variable-order method. Refer to MATLAB help file for more detailed information.

`ode15s` Solve stiff differential equations and DAEs, variable order method.

The function call is

$$[TOUT, YOUT] = \text{ode15s}(\text{ODEFUN}, TSPAN, Y0)$$

with all input and output argument descriptions as in `ode23` and others.

EXAMPLE 7.19: `ode15s`

Consider the stiff system in Example 7.16:

$$\begin{aligned} \dot{v} &= 790v - 1590w \\ \dot{w} &= 793v - 1593w \end{aligned} \quad \text{subject to} \quad \begin{aligned} v_0 &= v(0) = 1 \\ w_0 &= w(0) = -1 \end{aligned}$$

Write a MATLAB script that solves the system using `ode15s` and `ode45` and returns a table that includes the solution estimates for $v(t)$ at $t=0:0.1:1$, as well as the exact solution and the % relative error for both methods at each point. The exact solution was provided in Example 7.16.

Solution

```

disp('    t          v15s          v45          vExact          e_15s          e_45')

t = 0:0.1:1; u0 = [1;-1];
f = @(t,u)([790*u(1)-1590*u(2);793*u(1)-1593*u(2)]);

[t,u15s] = ode15s(f,t,u0);
% Values of v are in the first column of u15s

[t,u45] = ode45(f,t,u0);
% Values of v are in the first column of u45

uExact = @(t)([3180/797*exp(-3*t)-2383/797*exp(-800*t);1586/797*exp(-3*t)-2383/797*exp(-800*t)]);
for i = 1:length(t),
    uex(:,i) = uExact(t(i));    % Evaluate exact solution vector at each t
end

```

```

for k=1:length(t),
    t_coord = t(k);
    v15s = u15s(k,1); % Retain the first column of u15s: values of v
    v45 = u45(k,1); % Retain the first column of u45: values of v
    vExact = uex(1,k); % Retain the exact values of v

    e_15s = (vExact - v15s)/vExact*100;
    e_45 = (vExact - v45)/vExact*100;

    fprintf('%6.2f %11.6f%11.6f %11.6f %11.6f %11.8f\n',t_coord,v15s,
v45,vExact,e_15s,e_45)
end

```

t	v15s	v45	vExact	e_15s	e_45
0.00	1.0000000	1.0000000	1.0000000	0.0000000	0.000000000
0.10	2.955055	2.955511	2.955837	0.026464	0.01102274
0.20	2.187674	2.188567	2.189738	0.094262	0.05345813
0.30	1.620781	1.622657	1.622198	0.087342	-0.02830662
0.40	1.201627	1.201785	1.201754	0.010535	-0.00261497
0.50	0.890826	0.890278	0.890281	-0.061233	0.00033632
0.60	0.660191	0.659508	0.659536	-0.099254	0.00427387
0.70	0.489188	0.488525	0.488597	-0.121031	0.01461519
0.80	0.362521	0.361836	0.361961	-0.154754	0.03468108
0.90	0.268708	0.268171	0.268147	-0.209208	-0.00866986
1.00	0.199060	0.198645	0.198649	-0.207140	0.00178337

It is easy to see that even for this stiff system of ODEs, the solver ode45 still outperforms ode15s, which is specially designed to handle such systems.

PROBLEM SET (CHAPTER 7)

Euler's Method (Section 7.3)

In Problems 1 through 6, given each initial-value problem,

- Using Euler's method with the indicated step size h , calculate the solution estimates at the first two mesh points (beyond the initial point), as well as the local and global truncation errors at those locations.
- Write a MATLAB script file that uses EulerODE to find the approximate values produced by Euler's method and returns a table that includes these values, as well as the exact values and the global % relative errors, at all mesh points in the given interval.

1. $y' + 2(x+1)y = 0$, $y(0) = 1$, $0 \leq x \leq 1$, $h = 0.1$

Exact solution is $y = e^{-x(x+2)}$.

2. $xy' + y = x$, $y(1) = 0$, $1 \leq x \leq 2$, $h = 0.1$

Exact solution is $y = x/2 - 1/(2x)$.

3. $y' = y^2 \cos x$, $y(0) = \frac{1}{3}$, $0 \leq x \leq 1$, $h = 0.1$

Exact solution is $y = 1/(3 - \sin x)$.

4. $e^x y' = y^2$, $y(0) = 1$, $0 \leq x \leq 0.5$, $h = 0.05$


Exact solution is $y = e^x$.

5. $e^x y' = x^2 y^2$, $y(0) = \frac{1}{3}$, $0 \leq x \leq 1$, $h = 0.2$


Exact solution is $y = \left[e^{-x}(x^2 + 2x + 2) + 1 \right]^{-1}$.

6. $xy' - y = y^2$, $y(2) = 1$, $2 \leq x \leq 3$, $h = 0.1$


Exact solution is $y = x/(4 - x)$.

7.  Write a MATLAB script to solve the following IVP using `EulerODE` with $h = 0.4$, and again with $h = 0.2$. The file must return a table showing the estimated solutions at 0, 0.4, 0.8, 1.2, 1.6, 2 produced by both scenarios, as well as the exact values at these points.

$$3y' + 8y = 2e^{-2x/3}, \quad y(0) = 1, \quad 0 \leq x \leq 2$$


8.  Write a MATLAB script to solve the following IVP using `EulerODE` with $h = 0.2$, and again with $h = 0.1$. The file must return a table showing the estimated solutions at 0, 0.2, 0.4, 0.6, 0.8, 1 produced by both scenarios, as well as the exact values at these points.

$$y' - xy = x^2 \sin x, \quad y(0) = 1, \quad 0 \leq x \leq 1$$

9.  The free fall of a light particle of mass m released from rest and moving with a velocity v is governed by

$$m\dot{v} = mg - bv, \quad v(0) = 0$$


where $g = 9.81 \text{ m/s}^2$ is the gravitational acceleration and $b/m = 0.2$ is the coefficient of viscous damping. Write a MATLAB script that solves the IVP using `EulerODE` with $h = 0.05$ and returns a table of the estimated solutions at 0, 0.2, 0.4, 0.6, 0.8, 1, as well as the exact values at these points.

10.  The free fall of a heavy particle of mass m released from rest and moving with a velocity v is governed by

$$m\dot{v} = mg - bv^2, \quad v(0) = 0$$

where $g = 9.81 \text{ m/s}^2$ is the gravitational acceleration and $b/m = 0.2$ is the coefficient of viscous damping. Write a MATLAB script that solves the IVP using `EulerODE` with $h = 0.05$ and $h = 0.1$, and returns a table of estimated solutions at 0, 0.2, 0.4, 0.6, 0.8, 1 for both step sizes, as well as the exact values at these points.

Higher-Order Taylor Methods

 In Problems 11 through 15, for each initial-value problem, use the second-order Taylor method with the indicated step size to calculate the solution estimates at the first two mesh points (beyond the initial point).

11. $y' + 2y = x^3 y$, $y(0) = 1$, $0 \leq x \leq 2.4$, $h = 0.3$

12. $xy' = x + y$, $y(1) = 0$, $1 \leq x \leq 2$, $h = 0.1$

13. $y' = x^2 + y$, $y(0) = \frac{1}{2}$, $0 \leq x \leq 0.5$, $h = 0.05$

14. $yy' + xy^2 = x$, $y(1) = 2$, $1 \leq x \leq 2$, $h = 0.2$

15. $ty' = 2y + y^3$, $y(1) = 1$, $1 \leq t \leq 1.2$, $h = 0.05$

16. Write a user-defined function with function call `y = Taylor2_ODE(f, fp, x, y0)` that solves a first-order initial-value problem $y' = f(x, y)$, $y(x_0) = y_0$ using the second-order Taylor method. The input argument `fp` (anonymous function) is the result of implicit differentiation of `f` (anonymous function) with respect to `x`. All other inputs are as in `EulerODE`. Execute the function to solve

$$(x+1)y' = y + x, \quad y(0) = 1, \quad 0 \leq x \leq 1, \quad h = 0.1$$

In Problems 17 through 19, for each initial-value problem, write a MATLAB script that uses `EulerODE` and `Taylor2_ODE` (Problem 16) to find the approximate values produced by Euler and second-order Taylor methods and returns a table that includes these values, as well as the exact values and the global % relative errors for both methods, at all mesh points in the given interval.

17. $xy' = x - y$, $y(1) = 0$, $1 \leq x \leq 2$, $h = 0.1$

18. $2y' + y = 2e^{-x/2}$, $y(0) = 1$, $0 \leq x \leq 2$, $h = 0.2$

19. $\dot{y} = 10(2-t)(1-t)$, $y(0) = 0$, $0 \leq t \leq 1$, $h = 0.1$

20. Write a MATLAB script that solves the IVP below using `EulerODE` with $h = 0.04$ and `Taylor2_ODE` (Problem 16) with $h = 0.08$. The file must return a table that includes the estimated solutions at $x = 0:0.08:0.8$, as well as the exact values and the global % relative errors for both methods, at all mesh points in the given interval. Discuss the results.

$$(x-1)y' = x(2-y), \quad y(0) = 1, \quad 0 \leq x \leq 0.8$$

21. Write a MATLAB script file that solves the IVP below using `EulerODE` with $h = 0.05$ and `Taylor2_ODE` (Problem 16) with $h = 0.1$. The file must return a table that includes the estimated solutions at $x = 1:0.1:2$, as well as the exact values and the global % relative errors for both methods, at all mesh points in the given interval. Discuss the results.

$$xy' + y = x \ln x, \quad y(1) = \frac{3}{4}, \quad 1 \leq x \leq 2$$

Runge–Kutta Methods (Section 7.4)

RK2 Methods

In Problems 22 through 25, for each initial-value problem and the indicated step size h , use the following methods to compute the solution estimates at the first two mesh points (beyond the initial point):

- Improved Euler
- Heun
- Ralston

22. $3y' + (2x+1)y = 0$, $y(0) = 0.4$, $0 \leq x \leq 1$, $h = 0.1$

23. $e^{2x}y' = y^2$, $y(0) = \frac{1}{2}$, $0 \leq x \leq 0.5$, $h = 0.05$

24. $xy' = y - 3x$, $y(1) = 1$, $1 \leq x \leq 3$, $h = 0.1$

25. $y' = y^2 \sin(x/2)$, $y(0) = 1$, $0 \leq x \leq 4$, $h = 0.2$

26. Write a user-defined function with function call $y = \text{Imp_EulerODE}(f, x, y_0)$ that solves a first-order initial-value problem in the form $y' = f(x, y)$, $y(x_0) = y_0$ using the improved Euler's method. The input arguments are as in `HeunODE`. Execute the function to solve

$$xy' = y + xe^x, \quad y(0.5) = 1.6, \quad 0.5 \leq x \leq 1.5, \quad h = 0.05$$

27. Write a user-defined function with function call $y = \text{Ralston_ODE}(f, x, y_0)$ that solves a first-order initial-value problem $y' = f(x, y)$, $y(x_0) = y_0$ using Ralston's method. The input arguments are as in `HeunODE`. Execute the function to solve

$$e^x y' = \sin y, \quad y(0) = 1, \quad 0 \leq x \leq 4, \quad h = 0.4$$

28. Write a MATLAB script that uses `EulerODE`, `Taylor2_ODE` (Problem 16) and `HeunODE` to find the approximate values produced by Euler, second-order Taylor, and Heun's methods and returns a table that includes these values, as well as the exact values and the global relative errors for all three methods, at all mesh points in the given interval. Discuss the results.

$$xy' + y = xe^{-x/2}, \quad y(1) = 0, \quad 1 \leq x \leq 3, \quad h = 0.2$$

29. Write a MATLAB script to generate [Table 7.1](#) of Example 7.4. The file must call user-defined functions `HeunODE`, `Imp_EulerODE` (Problem 26) and `Ralston_ODE` (Problem 27).

RK3 Methods

In Problems 30 and 31, for each initial-value problem and the indicated step size h , use the following methods to compute the solution estimates at the first two mesh points (beyond the initial point).

- a. Classical RK3
- b. Heun's RK3

30. $y' = y \sin 2x$, $y(0) = 1$, $0 \leq x \leq 1$, $h = 0.1$


31. $y' = x^2 + y$, $y(0) = 1$, $0 \leq x \leq 1$, $h = 0.1$

32. Write a user-defined function with function call $y = \text{Classical_RK3}(f, x, y_0)$ that solves a first-order initial-value problem in the form $y' = f(x, y)$, $y(x_0) = y_0$ using the classical RK3 method. The input arguments are as in `HeunODE`.

33. Write a user-defined function with function call $y = \text{Heun_RK3}(f, x, y_0)$ that solves a first-order initial-value problem $y' = f(x, y)$, $y(x_0) = y_0$ using Heun's RK3 method. The input arguments are as in `HeunODE`.

34. Write a MATLAB script to generate [Table 7.2](#) of Example 7.5. The file must call user-defined functions `EulerODE`, `Classical_RK3` (Problem 32), and `Heun_RK3` (Problem 33).

RK4 Methods

 In Problems 35 through 40, for each initial-value problem and the indicated step size h , use the classical RK4 method to compute the solution estimates at the first two mesh points (beyond the initial point).

35. $xy' + (2x + 1)y = x$, $y(1) = \frac{1}{2}$, $1 \leq x \leq 2$, $h = 0.1$


36. $xy' - y = e^{y/x}$, $y(1) = 1$, $0 \leq x \leq 1.5$, $h = 0.2$


37. $y' = (1 - x)(3 - x)$, $y(0) = 1$, $0 \leq x \leq 1.6$, $h = 0.08$

38. $xy' + 2y = x \ln x$, $y(1) = 0$, $1 \leq x \leq 2$, $h = 0.1$


39. $y' + y^3 = 0$, $y(0) = \frac{2}{3}$, $0 \leq x \leq 2$, $h = 0.2$

40. $(x + y)y' + y = x$, $y(0) = \frac{2}{3}$, $0 \leq x \leq 3$, $h = 0.15$


41.  Write a MATLAB script to generate [Table 7.3](#) of Example 7.6. The file must call user-defined functions `EulerODE`, `Classical_RK3` (Problem 32), `HeunODE`, and `RK4`.

42.  Write a MATLAB script that calls user-defined functions `EulerODE`, `HeunODE`, and `RK4` to find the solution estimates produced by Euler's method, Heun's method, and the classical RK4 method and returns a table that includes these values at all mesh points in the given interval.


$$y' = y^2 + e^x, \quad y(0) = 0.1, \quad 0 \leq x \leq 0.5, \quad h = 0.04$$

43.  Write a MATLAB script that uses `EulerODE`, `HeunODE`, and `RK4` to solve the IVP by Euler's method, Heun's method, and the classical RK4 method and returns a table that includes the estimated solutions, as well as the global % relative errors for all three methods at all mesh points in the given interval. Discuss the results.

$$xy' = y + 2e^{y/x}, \quad y(1) = 0, \quad 1 \leq x \leq 1.5, \quad h = 0.05$$

44.  Write a MATLAB script that solves the initial-value problem in Problem 43 using `EulerODE` with $h = 0.025$, `HeunODE` with $h = 0.05$, and `RK4` with $h = 0.1$. The file must return a table that includes the estimated solutions, as well as the global % relative errors, at $x = 1, 1.1, 1.2, 1.3, 1.4, 1.5$ for all three methods. Discuss the results.

$$xy' = y + 2e^{y/x}, \quad y(1) = 0, \quad 1 \leq x \leq 1.5$$

45.  Write a MATLAB script that solves the following initial-value problem using `EulerODE` with $h = 0.0125$, `HeunODE` with $h = 0.025$, and `RK4` with $h = 0.1$. The file must return a table that includes the estimated solutions, as well as the global % relative errors, at $x = 1:0.1:2$ for all three methods. Discuss the results.

$$(x + 1)y' + y = x \ln x, \quad y(1) = \frac{1}{4}, \quad 1 \leq x \leq 2$$

46. ✎ For the following initial-value problem, use the Butcher's RK5 method with the indicated step size h to compute the solution estimate at the first mesh point (beyond the initial point).

$$xy' + \left(x + \frac{1}{2}\right)y = 2x, \quad y(0.5) = \frac{3}{4}, \quad 0.5 \leq x \leq 1.5, \quad h = 0.1$$

47. 🐛 Write a user-defined function with function call $y = \text{Butcher_RK5}(f, x, y_0)$ that solves the initial-value problem $y' = f(x, y)$, $y(x_0) = y_0$ using the Butcher's RK5 method. All input and output variable descriptions are similar to those in the user-defined function RK4. Apply `Butcher_RK5` to solve the IVP in Problem 46.
48. 🐛 Write a MATLAB script that uses `RK4` and `Butcher_RK5` (Problem 47) to solve the following IVP by RK4 and RK5 methods and returns a table that includes the estimated solutions, as well as the global % relative errors for both methods at all mesh points in the given interval. The estimated solutions and the errors must be displayed using six decimals. Discuss the results.

$$(x+1)y' = x(2-y), \quad y(0) = 1, \quad 0 \leq x \leq 0.5, \quad h = 0.05$$

Multistep Methods (Section 7.5)

Adams–Bashforth Method

49. ✎ Consider the initial-value problem

$$x^2y' = y, \quad y(1) = 1, \quad 1 \leq x \leq 2, \quad h = 0.1$$

- Using RK4, find an estimate for y_1 .
 - Using y_0 (initial condition) and y_1 from (a), apply the second-order Adams–Bashforth formula to find y_2 .
 - Apply the third-order Adams–Bashforth formula to find an estimate for y_3 .
 - Apply the fourth-order Adams–Bashforth formula to find an estimate for y_4 .
50. ✎ Repeat Problem 49 for the initial-value problem

$$y' + y^2 = 0, \quad y(0) = 1, \quad 0 \leq x \leq 1, \quad h = 0.1$$

51. ✎ Consider the initial-value problem

$$yy' = 2x - xy^2, \quad y(0) = 2, \quad 0 \leq x \leq 1, \quad h = 0.1$$

- Using RK4, find an estimate for y_1 and y_2 .
- Using y_0 (initial condition) and y_1 from (a), apply the second-order Adams–Bashforth formula to find y_2 .
- Knowing the exact solution is $y = \left[2(e^{-x^2} + 1)\right]^{1/2}$, compare the % relative errors associated with the y_2 estimates found in (a) and (b), and comment.

52. ✎ Consider the initial-value problem

$$y' = y \sin x, \quad y(1) = 1, \quad 1 \leq x \leq 2, \quad h = 0.1$$

- Using RK4, find an estimate for y_1 , y_2 , and y_3 .
 - Using y_0 (initial condition), and y_1 and y_2 from (a), apply the third-order Adams–Bashforth formula to find y_3 .
 - Knowing the exact solution is $y = e^{\cos 1 - \cos x}$, compare the % relative errors associated with the y_3 estimates found in (a) and (b), and comment.
53. ✎ Consider the initial-value problem

$$y' + y = \cos x, \quad y(0) = 1, \quad 0 \leq x \leq 1, \quad h = 0.1$$

- Using RK4, find an estimate for y_1 , y_2 , and y_3 .
 - Apply the fourth-order Adams–Bashforth formula to find y_4 .
54. 🖱 Write a user-defined function $y = \text{AB_3}(f, x, y_0)$ that solves an initial-value problem $y' = f(x, y)$, $y(x_0) = y_0$ using the third-order Adams–Bashforth method. Solve the IVP in Problem 52 by executing `AB_3`.
55. 🖱 Write a user-defined function with function call $y = \text{ABM_3}(f, x, y_0)$ that solves an initial-value problem $y' = f(x, y)$, $y(x_0) = y_0$ using the third-order Adams–Bashforth–Moulton predictor–corrector method. Solve the following IVP by executing `ABM_3`:

$$y' = y^2 \sin x, \quad y(0) = \frac{1}{2}, \quad 0 \leq x \leq 0.5, \quad h = 0.05$$

56. 🖱 Write a MATLAB script that calls the user-defined function `ABM4PredCorr` and generates [Table 7.4](#) in Example 7.8.

Systems of Ordinary Differential Equations (Section 7.6)

✎ In Problems 57 through 60, given each second-order initial-value problem,



- Transform into a system of first-order IVPs using state variables.
 - Apply Euler, Heun, and classical RK4 methods for systems, with the indicated step size h , to compute the solution estimate y_1 , and calculate the % relative error for each estimate.
57. $y'' = y + 2x$, $y(0) = 0$, $y'(0) = 1$, $0 \leq x \leq 1$, $h = 0.1$
Exact solution is $y(x) = 3\sinh x - 2x$.
58. $y'' + 3y' + 2y = x + 1$, $y(0) = 0$, $y'(0) = 0$, $0 \leq x \leq 1$, $h = 0.1$
Exact solution is $y(x) = \frac{1}{2}x + \frac{1}{4}(e^{-2x} - 1)$.
59. $y'' + 4y = 0$, $y(0) = 1$, $y'(0) = -1$, $0 \leq x \leq 1$, $h = 0.1$
Exact solution is $y(x) = \cos 2x - \frac{1}{2}\sin 2x$.

60. $y'' + 2y' + 2y = 0$, $y(0) = 1$, $y'(0) = 0$, $0 \leq x \leq 1$, $h = 0.1$

Exact solution is $y(x) = e^{-x}(\cos x + \sin x)$.



61. Consider the nonlinear, second-order initial-value problem

$$y'' + 2yy' = 0, \quad y(0) = 0, \quad y'(0) = 1$$

-  Using Heun's method for systems, with step size $h = 0.1$, find the estimated values of $y(0.1)$ and $y(0.2)$.
-  Confirm the results in (a) by executing the user-defined function `HeunODESystem`.



62. Consider the nonlinear, second-order initial-value problem

$$yy'' + (y')^2 = 0, \quad y(0) = 1, \quad y'(0) = -1$$

-  Using RK4 method for systems, with step size $h = 0.1$, find the estimated value of $y(0.1)$.
-  Confirm the results in (a) by executing the user-defined function `RK4System`.



63. Consider the linear, second-order IVP

$$x^2y'' = \frac{2}{3}y, \quad y(1) = 1, \quad y'(1) = -\frac{1}{2}, \quad 1 \leq x \leq 2, \quad h = 0.1$$

-  Transform into a system of first-order IVPs.
-  Write a MATLAB script that employs the user-defined functions `EulerODESystem`, `HeunODESystem`, and `RK4System` to solve the system in (a). The file must return a table of values for y generated by the three methods, as well as the exact values, at all the mesh points $x=1 : 0.1 : 2$. Use six decimal places to display all solution estimates.

64. Consider the linear, second-order IVP

$$x^2y'' + 5xy' + 3y = x^3, \quad y(1) = 0, \quad y'(1) = 0, \quad 1 \leq x \leq 1.5, \quad h = 0.05$$

-  Transform into a system of first-order IVPs.
-  Write a MATLAB script that employs the user-defined functions `EulerODESystem`, `HeunODESystem`, and `RK4System` to solve the system in (a). The file must return a table of values for y generated by the three methods, as well as the exact values, at the mesh points $x=1 : 0.05 : 1.5$. Use six decimal places to display all solution estimates.

65. Consider the mechanical system in translational motion shown in [Figure 7.8](#), where m is mass, k_1 and k_2 are stiffness coefficients, c_1 and c_2 are the coefficients of viscous

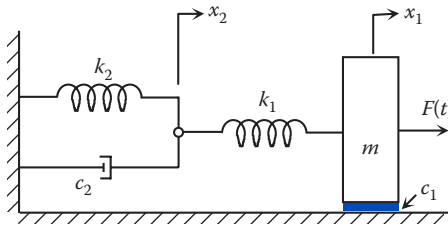


FIGURE 7.8
Problem 65.

damping, x_1 and x_2 are displacements, and $F(t)$ is the applied force. Assume, in consistent physical units, the following parameter values:

$$m = 2, \quad c_1 = 1, \quad c_2 = 1, \quad k_1 = 2, \quad k_2 = \frac{2}{3}, \quad F(t) = 4e^{-t/2} \sin t$$

The system's equations of motion are then expressed as

$$2\ddot{x}_1 + \dot{x}_1 + 2(x_1 - x_2) = 4e^{-t/2} \sin t$$

$$\dot{x}_2 + \frac{2}{3}x_2 - 2(x_1 - x_2) = 0$$

subject to initial conditions $x_1(0) = 0, x_2(0) = 1, \dot{x}_1(0) = -1$

- ✍ Transform into a system of first-order IVPs.
 - 📄 Write a MATLAB script that employs the user-defined function `RK4System` to solve the system in (a). The file must return the plot of x_1 and x_2 versus $0 \leq t \leq 8$ in the same graph. It is recommended to use at least 100 points for smoothness of curves.
66. In the mechanical system shown in [Figure 7.9](#), m is mass, c is the coefficient of viscous damping, $f_s = x^3$ is the nonlinear spring force, x is the block displacement, and $F(t)$ is the applied force. Assume, in consistent physical units, the following parameter values:

$$m = \frac{1}{2}, \quad c = 0.75, \quad F(t) = 100$$

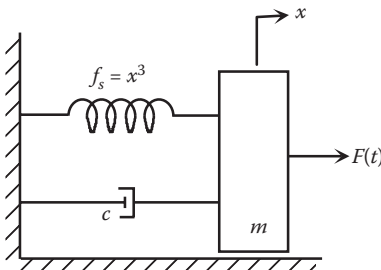


FIGURE 7.9
Problem 66.

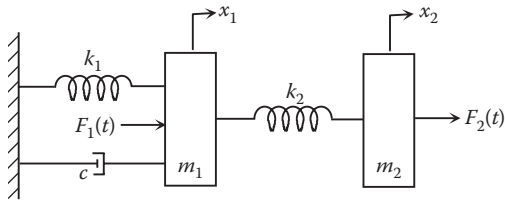


FIGURE 7.10
Problem 67.

The system's equation of motion is expressed as

$$\frac{1}{2}\ddot{x} + 0.75\dot{x} + x^3 = 100 \quad \text{subject to initial conditions} \quad x(0) = 0, \quad \dot{x}(0) = 1$$

- a. Transform into a system of first-order IVPs.
 - b. Write a MATLAB script that employs the user-defined function `RK4System` to solve the system in (a). The file must return the plot of x versus $0 \leq t \leq 5$. At least 100 points are recommended for smoothness of curves.
67. Consider the mechanical system shown in [Figure 7.10](#), where m_1 and m_2 represent mass, k_1 and k_2 are stiffness coefficients, c is the coefficient of viscous damping, x_1 and x_2 are block displacements, and $F_1(t)$ and $F_2(t)$ are the applied forces. Assume, in consistent physical units, the following parameter values:

$$m_1 = 1, \quad m_2 = 2, \quad c = 2, \quad k_1 = 1, \quad k_2 = 1, \quad F_1(t) = e^{-t}, \quad F_2(t) = 1$$

The system's equations of motion are then expressed as

$$\begin{aligned} \ddot{x}_1 + 2\dot{x}_1 + 2x_1 - x_2 &= e^{-t} \\ 2\ddot{x}_2 - x_1 + x_2 &= 1 \end{aligned} \quad \text{subject to initial conditions} \quad x_1(0) = 0, x_2(0) = 0, \dot{x}_1(0) = \frac{1}{2}, \dot{x}_2(0) = 1$$

- a. Transform into a system of first-order IVPs.
 - b. Write a MATLAB script that employs the user-defined function `RK4System` to solve the system in (a). The file must return the plot of x_1 and x_2 versus $0 \leq t \leq 30$ in the same graph. It is recommended to use at least 100 points for smoothness of curves.
68. The governing equations for an armature-controlled DC motor with a rigid shaft are derived as

$$\begin{aligned} J \frac{d\omega}{dt} + B\omega - K_t i &= T_L, \quad \omega(0) = 0 \\ L \frac{di}{dt} + Ri + K_e \omega &= v, \quad i(0) = 0 \end{aligned}$$

where

ω = angular velocity of the rotor

i = armature current

T_L = torque load

v = armature voltage

and J , B , L , R , K_t , and K_e represent constant parameters. Suppose, in consistent physical units,

$$J = 0.8 = L, \quad B = 0.45, \quad K_t = 0.25, \quad K_e = 1, \quad R = 1.25, \quad T_L = e^{-t}, \quad v = \sin t$$

Write a MATLAB script that employs the user-defined function `RK4System` to solve the system of governing equations. The file must return two separate plots (use `subplot`): angular velocity ω versus $0 \leq t \leq 10$, and the current i versus $0 \leq t \leq 10$.

69.  The governing equations for a field-controlled DC motor are derived as

$$J \frac{d\omega}{dt} + B\omega - K_t i = 0, \quad \omega(0) = 0$$
$$L \frac{di}{dt} + Ri = v, \quad i(0) = 0$$

where

ω = angular velocity of the rotor


i = armature current

v = armature voltage

and J , B , L , R , and K_t represent constant parameters. Suppose, in consistent physical units,

$$J = 1, \quad L = 0.5, \quad B = 0.75, \quad K_t = 0.8, \quad R = 0.45, \quad v = \frac{1}{2} \sin t$$

Write a MATLAB script that employs the user-defined function `RK4System` to solve the system of governing equations. The file must return two separate plots (use `subplot`): angular velocity ω versus $0 \leq t \leq 10$, and the current i versus $0 \leq t \leq 10$.

70.  Write a MATLAB script that solves the following second-order initial-value problem using `EulerODESystem` with $h = 0.025$, `HeunODESystem` with $h = 0.05$, and `RK4System` with $h = 0.1$. The file must return a table that includes the estimated solutions, as well as the global % relative errors, at $x = 1:0.1:2$ for all three methods. Discuss the results.

$$3x^2 y'' + 2xy' - y = 0, \quad y(1) = 0, \quad y'(1) = 1.5, \quad 1 \leq x \leq 2$$

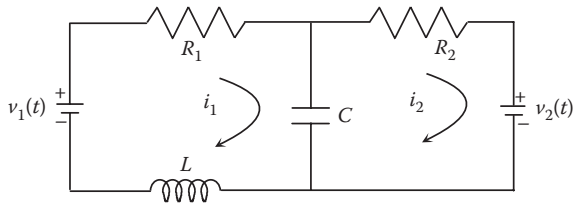


FIGURE 7.11
Problem 72.

71. Write a MATLAB script that employs the user-defined function `RK4System` with $h = 0.1$ to solve the following system of first-order IVPs. The file must return the plots of x_1 and x_3 versus $0 \leq t \leq 20$ in the same graph.

$$\begin{aligned} \dot{x}_1 &= \frac{1}{3}x_2 - (2t+1) & x_1(0) &= 0 \\ 3\dot{x}_2 &= -x_3 - 1 + t & \text{subject to } x_2(0) &= 1 \quad \text{and } 0 \leq t \leq 20 \\ \dot{x}_3 &= 2x_1 - x_3 + e^{-t/3} \sin t & x_3(0) &= -1 \end{aligned}$$

72. The two-loop electrical network shown in [Figure 7.11](#) is governed by

$$\begin{aligned} L\ddot{q}_1 + R_1\dot{q}_1 + \frac{1}{C}(q_1 - q_2) &= v_1(t) \\ R_2\dot{q}_2 - \frac{1}{C}(q_1 - q_2) &= v_2(t) \end{aligned} \quad \text{subject to } q_1(0) = 0, q_2(0) = 0, \dot{q}_1(0) = 0$$

where q_1 and q_2 are electric charges, L is inductance, R_1 and R_2 are resistances, C is capacitance, and $v_1(t)$ and $v_2(t)$ are the applied voltages. The electric charge and current are related through $i = dq/dt$. Assume, in consistent physical units, that the physical parameter values are

$$L = 0.1, \quad R_1 = 1, \quad R_2 = 1, \quad C_1 = 0.4, \quad v_1(t) = \sin t, \quad v_2(t) = \sin 2t$$

- a. Transform into a system of first-order IVPs.
- b. Write a MATLAB script that utilizes the user-defined function `RK4System` to solve the system in (a). The file must return the plot of q_1 and q_2 versus $0 \leq t \leq 5$ in the same graph. At least 100 points are recommended for plotting.
73. An inverted pendulum of length L and mass m , mounted on a motor-driven cart of mass M is shown in [Figure 7.12](#), where x is the linear displacement of the cart, φ is the angular displacement of the pendulum from the vertical, and μ is the force applied to the cart by the motor. The equations of motion are derived as

$$\begin{aligned} \ddot{x} &= \mu - \dot{x} \\ \ddot{\varphi} &= \frac{g}{L} \sin \varphi - \frac{1}{L} \ddot{x} \cos \varphi, \quad M = 1, \quad \bar{L} = \frac{J + mL^2}{mL} = 0.85, \quad g = 9.81 \end{aligned}$$

where J is the moment of inertia of the rod.

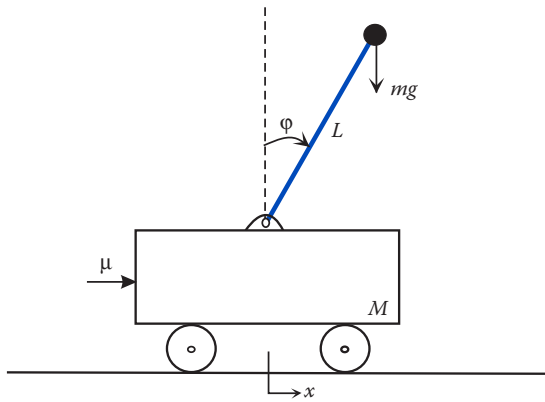


FIGURE 7.12
Problem 73.

- a. By choosing state variables $u_1 = x$, $u_2 = \dot{x}$, $u_3 = x + \bar{L}\phi$, $u_4 = \dot{x} + \bar{L}\dot{\phi}$ obtain the state-variable equations. Then insert the following into the state-variable equations:

$$\mu = -90u_1 - 10u_2 + 120u_3 + 30u_4$$

- b. Write a MATLAB script that utilizes the user-defined function `RK4System` to solve the system in (a). Three sets of initial conditions are to be considered: $\phi(0) = 10^\circ, 20^\circ, 30^\circ$, while $x(0) = 0 = \dot{x}(0) = \dot{\phi}(0)$ in all three cases. The file must return the plots of the three angular displacements corresponding to the three sets of initial conditions versus $0 \leq t \leq 1.5$ in the same graph. Angle measures must be converted to radians. Use at least 100 points for plotting purposes.
74. Write a user-defined function with function call `u = ABM_3System(f, x, u0)` that solves a system of first-order initial-value problems

$$\mathbf{u}' = \mathbf{f}(x, \mathbf{u}), \quad \mathbf{u}(a) = \mathbf{u}_0, \quad a = x_0 \leq x \leq x_n = b$$

using the third-order Adams–Bashforth–Moulton predictor–corrector method.

75. Write a MATLAB script that employs the user-defined function `ABM_3System` (Problem 74) to solve the following IVP. The file must return a table of values for y generated by the method, as well as the exact values at all the mesh points $x = 1 : 0.05 : 1.5$. Use six decimal places to display all solution estimates and exact values.

$$x^2 y'' + 5xy' + 3y = x^4, \quad y(1) = 1, \quad y'(1) = 0, \quad 1 \leq x \leq 2, \quad h = 0.1$$

Stability (Section 7.7)

76. The second-order Adams–Bashforth (AB2) method can be shown to have a stability region described by $0 < \lambda h < 1$ when applied to $y' = -\lambda y$ ($\lambda = \text{const} > 0$) subject to an initial condition.

Write a user-defined function with the function call $y = \text{AB_2}(f, x, y_0)$ that uses AB2 to solve an initial-value problem $y' = f(x, y)$, $y(a) = y_0$, where the input and output variables are described as usual. Solve the following IVP using AB_2 with step size $h = 0.2$ and again with $h = 0.55$, plot the estimated solutions together with the exact solution $y(x) = e^{-2x}$, and discuss stability:

$$y' + 2y = 0, \quad y(0) = 1, \quad 0 \leq x \leq 5$$

77.  Consider the non-self starting method

$$y_{i+1} = y_{i-1} + 2hf(x_i, y_i), \quad i = 1, 2, \dots, n-1$$

to solve an initial-value problem $y' = f(x, y)$, $y(a) = y_0$. A method such as RK4 can be used to start the iterations. Investigate the stability of this method as follows. Apply the method to $y' = -3y$, $y(0) = 1$ with $h = 0.1$ and plot (100 points) the estimated solution over the range $0 \leq x \leq 2$. Repeat with a substantially reduced step size $h = 0.02$ and plot over the range $0 \leq x \leq 4$. Fully discuss the findings.

Stiff Differential Equations (Section 7.8)

78.  Consider the IVP

$$\dot{y} + 100y = (99t + 1)e^{-t}, \quad y(0) = 1$$


The exact solution is $y = e^{-100t} + te^{-t}$ so that the first term quickly becomes negligible relative to the second term, but continues to govern stability. Apply Euler's method with $h = 0.1$ and plot (100 points) the solution estimates versus $0 \leq t \leq 1$. Repeat with $h = 0.01$ and plot versus $0 \leq t \leq 5$. Fully discuss the results as related to the stiffness of the differential equation.

79.  Consider the IVP


$$\dot{y} + 250y = 250 \sin t + \cos t, \quad y(0) = 1$$

The exact solution is $y = e^{-250t} + \sin t$ so that the first term quickly becomes negligible relative to the second term, but will continue to govern stability. Apply Euler's method with $h = 0.1$ and plot (100 points) the solution estimates versus $0 \leq t \leq 1$. Repeat with $h = 0.01$ and again with $h = 0.001$ and plot these two sets versus $0 \leq t \leq 5$. Fully discuss the results as related to the stiffness of the differential equation.

MATLAB Built-In Functions for Initial-Value Problems (Section 7.9)

80.  Write a MATLAB script that solves the following initial-value problem using ode45 and returns a table that includes the solution estimates at $x=0:0.1:1$:

$$(1 - x^2)y'' - xy' + 4y = 0, \quad y(0) = 1, \quad y'(0) = 1$$

81.  A device that plays an important role in the study of nonlinear vibrations is a van der Pol oscillator*, a system with a damping mechanism. When the amplitude of the motion is small, it acts to increase the energy. And, for large motion amplitude, it decreases the energy. The governing equation for the oscillator is the van der Pol equation

$$\ddot{y} - \mu(1 - y^2)\dot{y} + y = 0, \quad \mu = \text{const} > 0$$

Write a MATLAB script that solves the van der Pol equation with $\mu = 0.5$ and initial conditions $y(0) = 0.1$ and $\dot{y}(0) = 0$ using `ode23` and `ode45` and returns a table that includes the solution estimates at $t=0:0.1:1$.

82.  The motion of an object is described by

$$\begin{array}{l} \ddot{x} = -0.006v\dot{x} \\ \ddot{y} = -0.006v\dot{y} - 9.81 \end{array} \quad \text{subject to} \quad \begin{array}{ll} x(0) = 0 & \dot{x}(0) = 30 \\ y(0) = 0 & \dot{y}(0) = 25 \end{array}$$


initial positions initial velocities

where 9.81 represents the gravitational acceleration, and $v = \sqrt{\dot{x}^2 + \dot{y}^2}$ is the speed of the object. Initial positions of zero indicate that the object is placed at the origin of the xy -plane at the initial time. Determine the positions x and y at $t=0:0.1:1$ using `ode45`.

83.  Consider

$$\begin{array}{l} \dot{x} = 10(y - x) \\ \dot{y} = 15x - xz - y \\ \dot{z} = xy - 3z \end{array} \quad \text{subject to} \quad \begin{array}{l} x(0) = 1 \\ y(0) = -1 \\ z(0) = 1 \end{array}$$

Solve the system using any MATLAB solver and plot (100 points) each dependent variable versus $0 \leq t \leq 10$. What are the steady-state values of the three dependent variables?

84.  Consider the double pendulum system shown in [Figure 7.13](#) consisting of two identical rods and bobs attached to them, coupled with a linear spring. The system's free motion is described by

$$\begin{array}{l} mL^2\ddot{\theta}_1 + (mgL + kl^2)\theta_1 - kl^2\theta_2 = 0 \\ mL^2\ddot{\theta}_2 - kl^2\theta_1 + (mgL + kl^2)\theta_2 = 0 \end{array} \quad \text{subject to} \quad \begin{array}{ll} \theta_1(0) = 0 & \dot{\theta}_1(0) = -1 \\ \theta_2(0) = 0 & \dot{\theta}_2(0) = 0 \end{array}$$

initial angular positions initial angular velocities

Assume

$$\frac{g}{L} = 8, \quad \frac{k}{m} \left(\frac{l}{L} \right)^2 = 2$$

* A van der Pol oscillator is an electrical circuit, which consists of two DC power sources, a capacitor, resistors, inductors, and a triode composed of a cathode, an anode, and a grid controlling the electron flow.

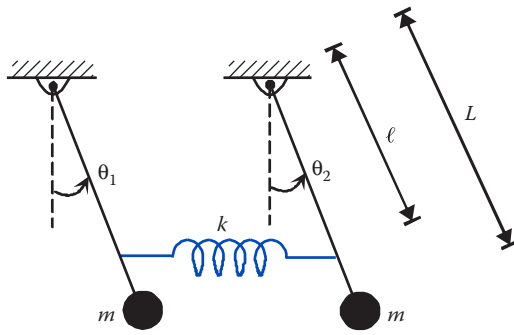



FIGURE 7.13

Problem 84.

Solve using `ode45`, and plot the angular displacements θ_1 and θ_2 versus $0 \leq t \leq 2$, and angular velocities $\dot{\theta}_1$ and $\dot{\theta}_2$ versus $0 \leq t \leq 2$, each pair in one figure.

85.  The mechanical system in [Figure 7.14](#) undergoes translational and rotational motions, described by

$$\begin{aligned}
 mL^2\ddot{\theta} + mL\ddot{x} + mgL\theta &= 0 \\
 mL\ddot{\theta} + (m + M)\ddot{x} + c\dot{x} + kx &= F(t)
 \end{aligned}
 \quad \text{subject to} \quad
 \begin{array}{ll}
 \theta(0) = 0 & \dot{\theta}(0) = 0 \\
 x(0) = 0 & \dot{x}(0) = -1
 \end{array}$$

initial positions initial velocities

where the applied force is $F(t) = e^{-t/2}$. Assume, in consistent physical units, the following parameter values:

$$m = 0.4, \quad M = 1, \quad L = 0.5, \quad g = 9.81, \quad k = 1, \quad c = 1$$

Solve using `ode45`, and plot the angular displacement θ and linear displacement x versus $0 \leq t \leq 10$, in two separate figures. Determine the steady-state values of θ and x .

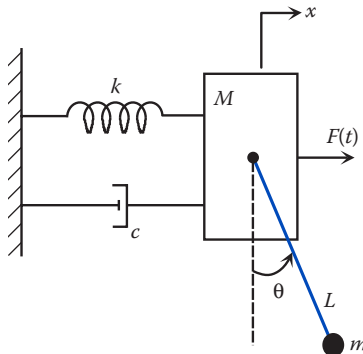


FIGURE 7.14

Problem 85.

Numerical Solution of Boundary-Value Problems

In [Chapter 7](#), we learned that an initial-value problem refers to the situation when an n th-order differential equation is accompanied by n initial conditions, specified at the same value of the independent variable. It was also mentioned that in other applications, these auxiliary conditions may be specified at different values of the independent variable, usually at the extremities of the system, and the problem is known as a boundary-value problem (BVP).

Boundary-value problems can be solved numerically by using either the shooting method or the finite-difference method. To apply the shooting method, the boundary-value problem is first made into an initial-value problem (IVP) by guessing at the initial condition(s) that are obviously absent in the description of a BVP. The IVP is then solved numerically and the solution is tested to see if the original boundary conditions are satisfied. Therefore, the shooting method relies on techniques such as the fourth-order Runge–Kutta method (RK4, [Chapter 7](#)) for solving IVPs. The finite-difference method is based on dividing the system interval into several subintervals and replacing the derivatives by the finite-difference approximations ([Chapter 6](#)). This will generate a system of algebraic equations which is then solved using the techniques of [Chapter 4](#).

8.1 Second-Order BVP

Consider a second-order differential equation in its most general form

$$y'' = f(x, y, y'), \quad a \leq x \leq b$$

subject to two boundary conditions, normally specified at the endpoints a and b . Because a boundary condition can be a given value of y or a value of y' , different forms of boundary conditions may be encountered.

8.2 Boundary Conditions

The most common boundary conditions are as follows:

Dirichlet boundary conditions (values of y at the endpoints are given):

$$y(a) = y_a, \quad y(b) = y_b$$

Neumann boundary conditions (values of y' at the endpoints are given):

$$y'(a) = y'_a, \quad y'(b) = y'_b$$

Mixed boundary conditions:

$$c_1 y'(a) + c_2 y(a) = B_a, \quad c_3 y'(b) + c_4 y(b) = B_b$$

8.3 Higher-Order BVP

Boundary-value problems can be based on differential equations with orders higher than two, which require additional boundary conditions. As an example, in the analysis of free transverse vibrations of a uniform beam of length L , simply supported (pinned) at both ends, we encounter a fourth-order ODE

$$\frac{d^4 X}{dx^4} - \gamma^4 X = 0 \quad (\gamma = \text{const} > 0)$$

subject to

$$\begin{aligned} X(0) &= 0, & X(L) &= 0 \\ X''(0) &= 0, & X''(L) &= 0 \end{aligned}$$

8.4 Shooting Method

A BVP involving an n th-order ODE comes with n boundary conditions. Using state variables, the n th-order ODE is readily transformed into a system of first-order ODEs. Solving this new system requires exactly n initial conditions. The boundary condition(s) given at the left end of the interval also serve as initial conditions at that point, while the rest of the initial conditions must be guessed. In particular, for a second-order BVP, the boundary condition at the left end serves as the initial condition there, while the other initial condition must be guessed. The system is then solved numerically via RK4 and the value of the resulting solution at the other endpoint is compared with the boundary condition there. If the accuracy is not acceptable, the initial value is guessed again and the ensuing system is solved one more time. The procedure is repeated until the solution at that end agrees with the prescribed boundary condition there.

EXAMPLE 8.1: LINEAR BVP

Solve the following BVP using the shooting method.

$$\ddot{u} = 0.02u + 1, \quad u(0) = 10, \quad u(10) = 100, \quad \ddot{u} = \frac{d^2 u}{dt^2}$$

Solution

Since the ODE is second order, there are two state variables: $x_1 = u$, $x_2 = \dot{u}$. The state-variable equations are then formed as

$$\begin{cases} \dot{x}_1 = x_2 & x_1(0) = 10 \\ \dot{x}_2 = 0.02x_1 + 1 & x_2(0) = ? \end{cases} \quad (8.1)$$

This system could be solved numerically via RK4 but that would require initial conditions $x_1(0)$ and $x_2(0)$. Of these two, however, only $x_1(0) = 10$ is available. Note that there is a *unique* value of $x_2(0)$ for which the above system yields the solution of the original BVP.

Strategy to Find $x_2(0)$

We first guess a value for $x_2(0)$ and solve the system in Equation 8.1 using RK4. From the solution, we extract the first state variable x_1 which happens to be u . The value of this variable at the right end (namely, $u(10)$) is then compared with the boundary condition at that end, that is, $u(10) = 100$, which will serve as the *target* here. Next, we guess another value for $x_2(0)$ and go through the process a second time. This way, for each guess of $x_2(0)$, we find a value for $u(10)$. Because the original ODE is linear, these values are linearly related as well. As a result, a linear interpolation of this data will provide the unique value for $x_2(0)$ that will result in the correct solution.

As a first guess, we arbitrarily choose $x_2(0) = 10$ so that Equation 8.1 becomes

$$\begin{cases} \dot{x}_1 = x_2 & x_1(0) = 10 \\ \dot{x}_2 = 0.02x_1 + 1 & \boxed{x_2(0) = 10} \end{cases}$$

```
>> f = @(t,x)([x(2);0.02*x(1)+1]);
>> x0_first = [10;10];
>> t = linspace(0,10,20);
>> x_first = RK4System(f,t,x0_first);
>> u_10_first = x_first(1,end)
```

```
u_10_first =
```

```
217.5208
```

The result overshoots the target $u(10) = 100$. As a second guess, we pick $x_2(0) = 0$ so that Equation 8.1 becomes

$$\begin{cases} \dot{x}_1 = x_2 & x_1(0) = 10 \\ \dot{x}_2 = 0.02x_1 + 1 & \boxed{x_2(0) = 0} \end{cases}$$

Note that f and t in our MATLAB code remain unchanged; only the initial state vector is modified.

```
>> x0_second = [10;0];
>> x_second = RK4System(f,t,x0_second);
>> u_10_second = x_second(1,end)
```

```
u_10_second =
```

```
80.6910
```

This time the result is below the target. In summary,

$$\begin{aligned}x_2(0) = 10 & & u(10) = 217.5208 \\x_2(0) = 0 & & u(10) = 80.6910\end{aligned}\tag{8.2}$$

Linear interpolation of this data will lead to the correct value for $x_2(0)$; see [Figure 8.1](#).

We will handle this by using linear Lagrange interpolation ([Chapter 5](#)), more specifically, the user-defined function `LagrangeInterp`:

```
>> yi = LagrangeInterp([u_10_first, u_10_second],[10, 0],100)
yi =
    1.4112
```

Therefore, $x_2(0) = 1.4112$. Using this information in Equation 8.1, we have

$$\begin{cases} \dot{x}_1 = x_2 & x_1(0) = 10 \\ \dot{x}_2 = 0.02x_1 + 1 & x_2(0) = 1.4112 \end{cases}$$

```
>> x0_final = [10;yi];
>> x_final = RK4System(f,t,x0_final);
>> x_final(1,end)
```

```
ans =
    100.0000
```

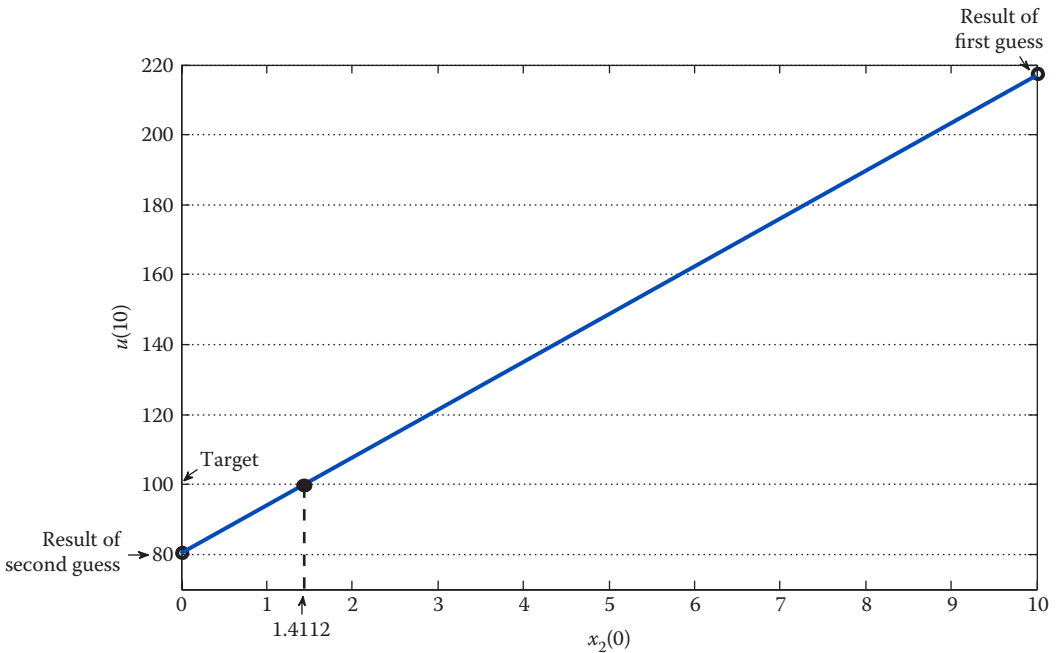


FIGURE 8.1

Linear interpolation of the data in Equation 8.2.

The numerical solution u to the original BVP is obtained by extracting the first row of `x_final`, which contains 20 values of the first state variable $x_1 = u$. In order to generate a smooth solution trajectory, however, we should use at least 100 points, as opposed to the current 20 points. Similarly, the first row of `x_first` gives the solution corresponding to the first guess of $x_2(0)$, and the first row of `x_second` gives the solution corresponding to the second guess of $x_2(0)$.

This is summarized in the script below.

```
>> t = linspace(0,10); % 100 points
>> x_first = RK4System(f,t,x0_first);
>> u_first = x_first(1,:); % Solution based on the first guess
>> x_second = RK4System(f,t,x0_second);
>> u_second = x_second(1,:); % Solution based on the second guess
>> x_final = RK4System(f,t,x0_final);
>> u_final = x_final(1,:); % True solution
>> plot(t,u_first,t,u_second,t,u_final) % Figure 8.2
```

A linear BVP such as in Example 8.1 is relatively easy to solve using the shooting method because the data generated by two initial-value estimates can be interpolated linearly leading to the correct estimate. This, however, will not be enough in the case of a nonlinear BVP. One remedy would be to apply the shooting method three times and then use a quadratic interpolating polynomial to estimate the initial value. But it is not very likely that this approach would generate the correct solution, and further iterations would probably be required.

A more viable option is the following: guess a value for the missing initial condition, solve the ensuing system, and find the value of the solution at the right end. This is either above the unused boundary condition (target) or below it. If it is above the target, we must pick a second guess that leads to a value below the target. If it is below the target, we must pick a second guess that leads to a value above the target. Subsequently, the bisection method (Chapter 3) is employed to find the unique value of the missing initial condition that will result in the true solution. Of course, the bisection

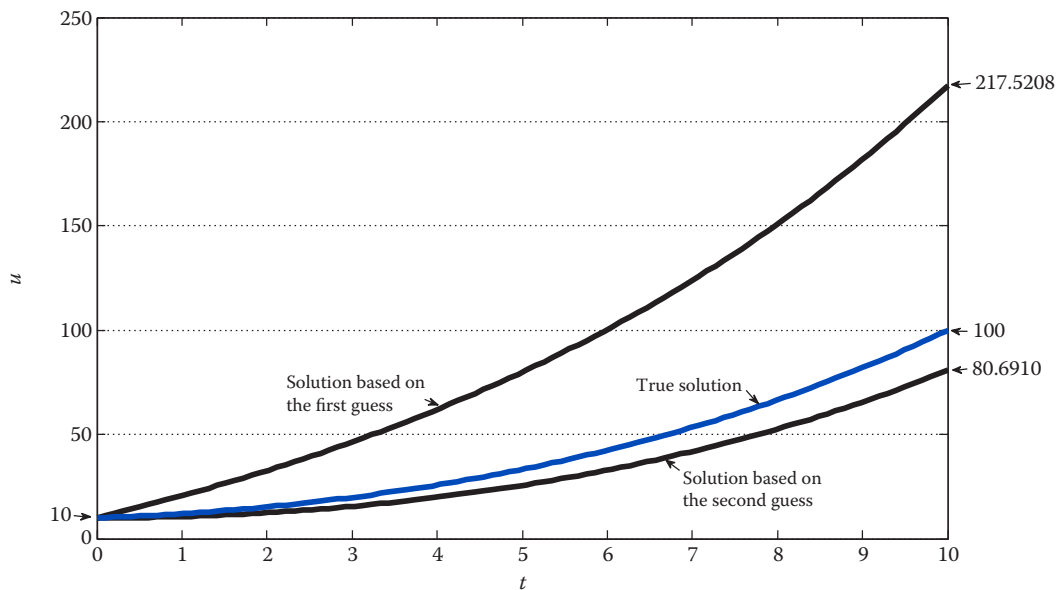


FIGURE 8.2
Solution of the boundary-value problem in Example 8.1.

method uses iterations that terminate when a tolerance is met. Therefore, the precision of the final outcome is directly dependent on the magnitude of the selected tolerance. The example that follows demonstrates the details of this approach.

EXAMPLE 8.2: NONLINEAR BVP

The temperature distribution along a fin can be modeled as a BVP

$$T_{xx} - \beta T^4 - \alpha T + \gamma = 0, \quad T(0) = 500, \quad T(0.2) = 350, \quad T_{xx} = \frac{d^2T}{dx^2}$$

where $\alpha = 20$, $\beta = 10^{-8}$, and $\gamma = 5 \times 10^3$ with all parameters in consistent physical units. Solve the nonlinear BVP using the shooting method.

Solution

There are two state variables selected $\eta_1 = T$, $\eta_2 = T_{xx}$ and the state-variable equations are formed as

$$\begin{cases} \eta_1' = \eta_2 & \eta_1(0) = 500 \\ \eta_2' = \alpha\eta_1 + \beta\eta_1^4 - \gamma & \eta_2(0) = ? \end{cases} \quad (8.3)$$

Of the two required initial conditions, only $\eta_1(0) = 500$ is available. As a first guess, we arbitrarily choose $\eta_2(0) = 100$ so that Equation 8.3 becomes

$$\begin{cases} \eta_1' = \eta_2 & \eta_1(0) = 500 \\ \eta_2' = \alpha\eta_1 + \beta\eta_1^4 - \gamma & \boxed{\eta_2(0) = 100} \end{cases}$$

We will solve this system via RK4 using 100 points.

```
>> f = @(x,eta) ([eta(2); 20*eta(1)+1e-8*eta(1)^4-5e3]);
>> x = linspace(0,0.2);
>> eta0_first = [500;100];
>> eta_first = RK4System(f,x,eta0_first);
>> T_end_first = eta_first(1,end)
```

T_end_first =

646.2081

The result overshoots the target of 350. Therefore, we must pick a second guess for $\eta_2(0)$ that leads to a value below the target. This will require at least one trial. It turns out that a second guess of $\eta_2(0) = -2000$ will work here. Then, Equation 8.3 reduces to

$$\begin{cases} \eta_1' = \eta_2 & \eta_1(0) = 500 \\ \eta_2' = \alpha\eta_1 + \beta\eta_1^4 - \gamma & \boxed{\eta_2(0) = -2000} \end{cases}$$

Note that f and x in our MATLAB code remain unchanged; only the initial state vector is modified.

```
>> eta0_second = [500;-2000];
>> eta_second = RK4System(f,x,eta0_second);
```

```
>> T_end_second = eta_second(1,end)
```

```
T_end_second =
```

```
157.0736
```

The result is below the target of 350. In summary,

$$\begin{array}{ll} \eta_2(0)=100 & T(0.2)=646.2081 \\ \eta_2(0)=-2000 & T(0.2)=157.0736 \end{array}$$

Therefore, the unique value of $\eta_2(0)$ that leads to the true solution lies in the interval $[-2000, 100]$. We will find this value using the bisection method. The iterations (maximum 30) terminate when the computed $T(0.2)$ is within 10^{-4} of the target.

```
eta20L = -2000; eta20R = 100; % Left and right end of the interval
kmax = 30; % Maximum number of iterations
tol = 1e-4; % Tolerance
T_end = 350; % Boundary condition (target)
```

```
for k = 1:kmax,
eta20 = (eta20L + eta20R)/2; % Bisection
eta0 = [500;eta20]; % Set initial state vector
eta = RK4System(f,x,eta0); % Solve the system
T(k) = eta(1,end); % Extract T(0.2)
err = T(k) - T_end; % Compare with target
```

```
% Adjust the left or right value of initial condition based on whether
% error is positive or negative
```

```
if abs(err) < tol,
break
end
if err > 0,
eta20R = eta20;
else
eta20L = eta20;
end
end
```

```
>> k
```

```
k =
```

```
21 % Number of iterations needed to meet tolerance
```

```
>> T(21)
```

```
ans =
```

```
350.0000 % Agrees with target T(0.2)=350
```

```
>> eta20
```

```
eta20 =
```

```
-1.1643e+03 % Actual value for the missing initial condition
```

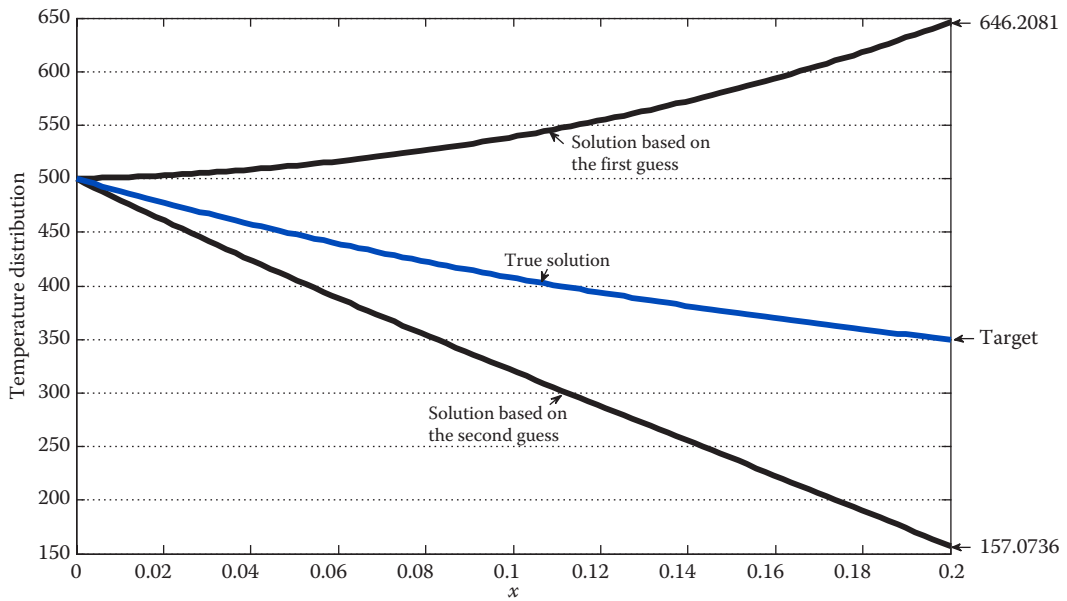



FIGURE 8.3

The temperature distribution in Example 8.2.

With this information, Equation 8.3 becomes

$$\begin{cases} \eta_1' = \eta_2 & \eta_1(0) = 500 \\ \eta_2' = \alpha\eta_1 + \beta\eta_1^4 - \gamma & \eta_2(0) = -1164.3 \end{cases}$$

This system is then solved via RK4. The numerical solution T of the original BVP is obtained by extracting the first row of the output, which represents the first state variable $\eta_1 = T$. Similarly, the first row of `eta_first` gives the solution corresponding to the first guess of $\eta_2(0)$, and the first row of `eta_second` gives the solution corresponding to the second guess of $\eta_2(0)$. This is summarized in the script below.

```
>> T_first = eta_first(1,:);
>> T_second = eta_second(1,:);
>> eta0 = [500;eta20];
>> eta_final = RK4System(f,x,eta0);
>> T_final = eta_final(1,:);
>> plot(x,T_first,x,T_second,x,T_final) % Figure 8.3
```

The shooting method loses its efficiency when applied to higher-order boundary-value problems, which will require more than one guess for the initial values. For those cases, other techniques, such as the finite-difference method, need to be employed.

8.5 Finite-Difference Method

The finite-difference method is the most commonly used alternative to the shooting method. The interval $[a, b]$ over which the BVP is to be solved is first divided into

N subintervals of length $h = (b - a)/N$. As a result, a total of $N + 1$ grid points are generated, including $x_1 = a$ and $x_{N+1} = b$, which are the left and right endpoints. The other $N - 1$ points x_2, \dots, x_N are the interior grid points. At each interior grid point, the derivatives involved in the differential equation are replaced with finite divided differences. This way, the differential equation is transformed into a system of $N - 1$ algebraic equations that can then be solved using the methods previously discussed in [Chapter 4](#).

Because of their accuracy, central-difference formulas are often used in finite-difference methods. Specifically, for the first and second derivatives of y with respect to x ,

$$\left. \frac{dy}{dx} \right|_{x_i} \cong \frac{y_{i+1} - y_{i-1}}{2h}, \quad \left. \frac{d^2y}{dx^2} \right|_{x_i} \cong \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}$$

Several difference formulas are listed in Table 6.3, Section 6.2.

EXAMPLE 8.3: FINITE-DIFFERENCE METHOD, LINEAR BVP

Consider the BVP in Example 8.1:

$$\ddot{u} = 0.02u + 1, \quad u(0) = 10, \quad u(10) = 100$$

Solve by the finite-difference method using central-difference formulas and $h = \Delta t = 2$.

Solution

At each interior grid point, the second derivative is replaced with a central-difference formula to obtain

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta t^2} = 0.02u_i + 1$$

Since the length of the interval is 10 and $\Delta t = 2$, we have $N = 10/2 = 5$, so that there are $N - 1 = 4$ interior points. Simplifying the above equation, we find

$$u_{i-1} - (2 + 0.02\Delta t^2)u_i + u_{i+1} = \Delta t^2, \quad i = 2, 3, 4, 5 \tag{8.4}$$

Note that $u_1 = 10$ and $u_6 = 100$ are available from the boundary conditions. Applying Equation 8.4 yields

$$\begin{array}{l} u_1 - 2.08u_2 + u_3 = 4 \\ u_2 - 2.08u_3 + u_4 = 4 \\ u_3 - 2.08u_4 + u_5 = 4 \\ u_4 - 2.08u_5 + u_6 = 4 \end{array} \quad \begin{array}{l} \Rightarrow \\ \Rightarrow \\ \Rightarrow \\ \Rightarrow \end{array} \quad \begin{array}{l} u_1=10 \\ u_6=100 \end{array} \quad \left[\begin{array}{cccc} -2.08 & 1 & & \\ 1 & -2.08 & 1 & \\ & 1 & -2.08 & 1 \\ & & 1 & -2.08 \end{array} \right] \begin{Bmatrix} u_2 \\ u_3 \\ u_4 \\ u_5 \end{Bmatrix} = \begin{Bmatrix} -6 \\ 4 \\ 4 \\ -96 \end{Bmatrix}$$

As projected by the structure of Equation 8.4, the resulting coefficient matrix is tridiagonal, and thus will be solved using the Thomas method (Section 4.3). That yields

$$u_2 = 15.0489, \quad u_3 = 25.3018, \quad u_4 = 41.5788, \quad u_5 = 65.1821$$

Comparison with the Shooting Method

In Example 8.1, we learned that the true solution (using the shooting method) of the current BVP is obtained by solving the following system via RK4:

$$\begin{cases} \dot{x}_1 = x_2 & x_1(0) = 10 \\ \dot{x}_2 = 0.02x_1 + 1 & x_2(0) = 1.4112 \end{cases}$$

In order to compare the numerical results generated by the shooting method with those given above by the finite-difference method, the step size in shooting method (RK4) must be adjusted to 2:

```
>> t = 0:2:10;
>> f = @(t,x)([x(2);0.02*x(1)+1]);
>> x0 = [10;yi];
% Solutions by the shooting method at interior grid points
>> u = RK4System(f,t,x0); u(1,[2:end-1])

ans =

    15.2760 25.8084 42.4455 66.5269
```

The exact values at the interior grid points are readily found as follows:

```
>> ue = matlabFunction(dsolve('D2u = 0.02*u+1','u(0)=10, u(10)=100'));
% Exact solutions at interior grid points
>> ue(t(2:end-1))

ans =

    15.2762 25.8093 42.4478 66.5315
```

A summary of the preceding calculations is presented in [Table 8.1](#), where it is immediately observed that the shooting method produces much more accurate estimates than the finite-difference method. The main reason for this is that the shooting method relies on the RK4 method, which enjoys a very high level of accuracy. The accuracy of both techniques can be improved by reducing the step size Δt .

EXAMPLE 8.4: FINITE-DIFFERENCE METHOD, NONLINEAR BVP

Consider the BVP in Example 8.2:

$$T_{xx} - \beta T^4 - \alpha T + \gamma = 0, \quad T(0) = 500, \quad T(0.2) = 350$$

TABLE 8.1

Comparison of Results: Shooting Method, Finite Difference, Actual (Example 8.3)

t	Finite Difference	Shooting Method	Actual
0	10	10	10
2	15.0489	15.2760	15.2762
4	25.3018	25.8084	25.8093
6	41.5788	42.4455	42.4478
8	65.1821	66.5269	66.5315
10	100	100	100

where $\alpha = 20$, $\beta = 10^{-8}$, and $\gamma = 5 \times 10^3$ with all parameters in consistent physical units. Solve by the finite-difference method using central-difference formulas and $h = \Delta x = 0.04$.

Solution

The interval is 0.2 in length and $\Delta x = 0.04$, hence $N = 0.2/0.04 = 5$, and there are $N - 1 = 4$ interior grid points. At each interior grid point, the second derivative is replaced with a central-difference formula to obtain

$$\frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} - \beta T_i^4 - \alpha T_i + \gamma = 0$$

Simplifying the above, we have

$$T_{i-1} - (2 + \alpha \Delta x^2)T_i + T_{i+1} - \beta \Delta x^2 T_i^4 + \gamma \Delta x^2 = 0, \quad i = 2, 3, 4, 5 \quad (8.5)$$

Applying Equation 8.5, keeping in mind that $T_1 = 500$ and $T_6 = 350$, we find

$$\begin{aligned} 508 - 2.0320T_2 - 1.6 \times 10^{-11}T_2^4 + T_3 &= 0 \\ T_2 - 2.0320T_3 - 1.6 \times 10^{-11}T_3^4 + T_4 + 8 &= 0 \\ T_3 - 2.0320T_4 - 1.6 \times 10^{-11}T_4^4 + T_5 + 8 &= 0 \\ 358 + T_4 - 2.0320T_5 - 1.6 \times 10^{-11}T_5^4 &= 0 \end{aligned}$$

We will solve this system of nonlinear equations using Newton's method for systems; see [Chapter 4](#). To that end, we first conveniently express the above system as

$$\begin{aligned} f_1(T_2, T_3, T_4, T_5) &= 0 \\ f_2(T_2, T_3, T_4, T_5) &= 0 \\ f_3(T_2, T_3, T_4, T_5) &= 0 \\ f_4(T_2, T_3, T_4, T_5) &= 0 \end{aligned}$$

Newton's method requires that in each step we solve a linear system in the form

$$\begin{bmatrix} \frac{\partial f_1}{\partial T_2} & & & \frac{\partial f_1}{\partial T_5} \\ & \cdot & \cdot & \\ & \cdot & \cdot & \\ \frac{\partial f_4}{\partial T_2} & & & \frac{\partial f_4}{\partial T_5} \end{bmatrix} \begin{Bmatrix} \Delta T_2 \\ \cdot \\ \cdot \\ \Delta T_5 \end{Bmatrix} = \begin{Bmatrix} -f_1 \\ \cdot \\ \cdot \\ -f_4 \end{Bmatrix} \quad (8.6)$$

It is readily observed that the coefficient matrix in Equation 8.6 is tridiagonal, thus the system in Equation 8.6 is most efficiently solved using the Thomas method.

Proceeding as in Example 4.20, the following script implements Newton's method to find the solution estimates at the four interior grid points. The initial values are arbitrarily selected as 400. The maximum of number of iterations, as well as the tolerance are chosen as in Example 4.20.

```

syms T2 T3 T4 T5
f1 = 508-2.0320*T2-1.6e-11*T2^4+T3; f2 = T2-2.0320*T3-1.6e-11*T3^4+T4+8;
f3 = T3-2.0320*T4-1.6e-11*T4^4+T5+8; f4 = 358+T4-2.0320*T5-1.6e-11*T5^4;
f = [f1;f2;f3;f4];
J = matlabFunction(jacobian(f,[T2,T3,T4,T5]));
F = matlabFunction(f);

tol = 1e-4; kmax = 20; T(1,:) = 400*ones(1,4);

for k = 1:kmax,
    A = J(T(k,1),T(k,2),T(k,3),T(k,4));
    b = -F(T(k,1),T(k,2),T(k,3),T(k,4));

    if abs(b(1)) < tol && abs(b(2)) < tol && abs(b(3)) < tol && abs(b(4))
    < tol,
        root = T(k,:);
        return
    end

    if det(A) == 0,
        break
    end

    delT = ThomasMethod(A,b); delT = delT';
    T(k+1,:) = T(k,:) + delT;
    if norm(delT) < tol,
        root = T(k+1,:);
        break
    end
end
end

```

Execution of this script shows that convergence is achieved after two iterations:

```
>> T
```

```
T =
```

```

400.0000    400.0000    400.0000    400.0000
457.7283    422.7499    393.8024    369.8407
457.6786    422.7049    393.7686    369.8176

```

Comparison with the Shooting Method

In order to compare these solution estimates with those obtained by the shooting method (Example 8.2), we adjust the step size used in RK4 to 0.04. A summary of the results is shown in [Table 8.2](#), where it is observed that the estimates provided by the two techniques closely follow one another.

TABLE 8.2

Comparison of Results: Shooting Method,
Finite Difference (Example 8.4)

x	Finite Difference	Shooting Method
0	500	500
0.04	457.6786	457.6375
0.08	422.7049	422.6493
0.12	393.7686	393.7178
0.16	369.8176	369.7862
0.2	350	350

8.5.1 Boundary-Value Problems with Mixed Boundary Conditions

In the case of mixed boundary conditions, information involving the derivative of the dependent variable is prescribed at one or both of the endpoints of the domain of solution. In these situations, the finite-difference method can be used as before, but the resulting system of equations cannot be solved because the values of the dependent variable at both endpoints are not available. This means there are more unknowns than there are equations. The additional equations are derived by using finite differences to discretize the one or two boundary conditions that involve the derivative. The combination of the equations already obtained at the interior grid points and those just generated at the endpoint(s) form a system of algebraic equations that can be solved as usual.

EXAMPLE 8.5: FINITE-DIFFERENCE METHOD, LINEAR BVP WITH MIXED BOUNDARY CONDITIONS

Consider the BVP

$$t\ddot{w} + \dot{w} + 2t = 0, \quad w(1) = 1, \quad \dot{w}(2.5) = -1$$

Solve by the finite-difference method and $\Delta t = 0.25$. Use central-difference approximations for all derivatives. Compare the results with the exact solution values at the grid points. Also confirm that the boundary condition at the right end is satisfied by the computed solution.

Solution

The interval is 1.5 in length and $\Delta t = 0.25$, hence $N = 1.5/0.25 = 6$, and there are five interior grid points. Replacing the first and second derivatives with central-difference formulas yields

$$t_i \frac{w_{i-1} - 2w_i + w_{i+1}}{\Delta t^2} + \frac{w_{i+1} - w_{i-1}}{2\Delta t} + 2t_i = 0$$

Simplify the above and apply at interior grid points so that

$$(2t_i - \Delta t)w_{i-1} - 4t_i w_i + (2t_i + \Delta t)w_{i+1} = -4t_i \Delta t^2, \quad i = 2, 3, 4, 5, 6$$

Consequently,

$$\begin{aligned} i = 2 & \quad (2t_2 - \Delta t)w_1 - 4t_2 w_2 + (2t_2 + \Delta t)w_3 = -4t_2 \Delta t^2 \\ i = 3 & \quad (2t_3 - \Delta t)w_2 - 4t_3 w_3 + (2t_3 + \Delta t)w_4 = -4t_3 \Delta t^2 \\ i = 4 & \quad (2t_4 - \Delta t)w_3 - 4t_4 w_4 + (2t_4 + \Delta t)w_5 = -4t_4 \Delta t^2 \\ i = 5 & \quad (2t_5 - \Delta t)w_4 - 4t_5 w_5 + (2t_5 + \Delta t)w_6 = -4t_5 \Delta t^2 \\ i = 6 & \quad (2t_6 - \Delta t)w_5 - 4t_6 w_6 + (2t_6 + \Delta t)w_7 = -4t_6 \Delta t^2 \end{aligned} \tag{8.7}$$

$w_1 = w(1) = 1$ is provided by the boundary condition at the left endpoint. At the right endpoint, however, $w_7 = w(2.5)$ is not directly available, but $\dot{w}(2.5)$ is. To approximate \dot{w} at the right end, we will use a one-sided, three-point backward difference formula so that the values at the previous points are utilized. Note that this has second-order accuracy (see Table 6.3, Section 6.2), which is in line with the central-difference formulas used in the earlier stages.

$$\dot{w}|_{t_i} \cong \frac{w_{i-2} - 4w_{i-1} + 3w_i}{2\Delta t} \quad (8.8)$$

Applying Equation 8.8 at the right end ($i = 7$),

$$\frac{w_5 - 4w_6 + 3w_7}{2\Delta t} = -1 \quad \overset{\text{Solve for } w_7}{\Rightarrow} \quad w_7 = \frac{1}{3}(-2\Delta t + 4w_6 - w_5) \quad (8.9)$$

Substitution into Equation 8.7 for w_7 , and expressing the system in matrix form, yields

$$\begin{bmatrix} -4t_2 & 2t_2 + \Delta t & 0 & 0 & 0 \\ 2t_3 - \Delta t & -4t_3 & 2t_3 + \Delta t & 0 & 0 \\ 0 & 2t_4 - \Delta t & -4t_4 & 2t_4 + \Delta t & 0 \\ 0 & 0 & 2t_5 - \Delta t & -4t_5 & 2t_5 + \Delta t \\ 0 & 0 & 0 & \frac{4}{3}(t_6 - \Delta t) & -\frac{4}{3}(t_6 - \Delta t) \end{bmatrix} \begin{bmatrix} w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} -4t_2\Delta t^2 - (2t_2 - \Delta t)w_1 \\ -4t_3\Delta t^2 \\ -4t_4\Delta t^2 \\ -4t_5\Delta t^2 \\ -4t_6\Delta t^2 + \frac{2}{3}(2t_6 + \Delta t)\Delta t \end{bmatrix} \quad (8.10)$$

The coefficient matrix is once again tridiagonal. This system is solved for the solution vector to obtain the approximate values at the interior grid points. The process is completed by attaching w_1 , which is available, and w_7 , which is found by Equation 8.9, to the vector of the interior grid values. The following MATLAB script performs these tasks:

```
a = 1; b = 2.5; dt = 0.25;
N = (b-a)/dt; t = a:dt:b;
w1 = 1; % BC at the left end

A = zeros(N-1,N-1); b = zeros(N-1,1); % Pre-allocate
b(1) = -4*t(2)*dt^2-(2*t(2)-dt)*w1;
b(N-1) = -4*t(N)*dt^2+(2/3)*(2*t(N)+dt)*dt;

for i = 1:N-2,
    A(i,i+1) = 2*t(i+1)+dt;
end
for i = 1:N-3,
    A(i+1,i) = 2*t(i+2)-dt;
    b(i+1) = -4*t(i+2)*dt^2;
end
A(N-1,N-2) = (4/3)*(t(N)-dt);
for i = 1:N-2,
    A(i,i) = -4*t(i+1);
end
A(N-1,N-1) = -(4/3)*(t(N)-dt);

w_interior = ThomasMethod(A,b);
w = [w1 w_interior'];
w(N+1) = (1/3)*(-2*dt+4*w(N)-w(N-1));

>> w

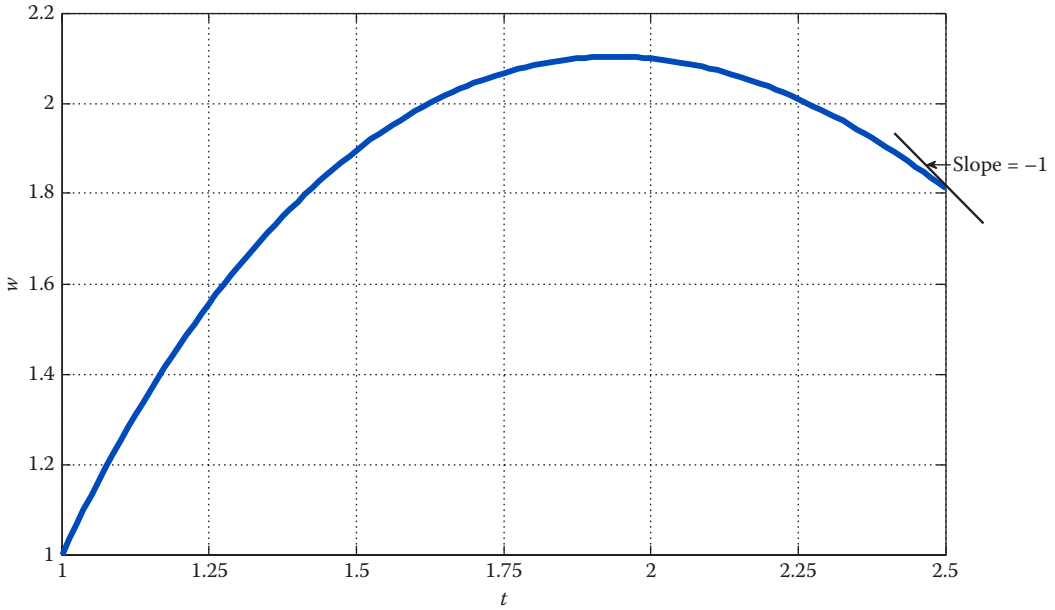
w =

    1.0000    1.5599    1.9044    2.0804    2.1164    2.0304    1.8351
```

TABLE 8.3

Comparison of Exact and Computed Values at Grid Points (Example 8.5)

Solution	$t = 1$	$t = 1.25$	$t = 1.5$	$t = 1.75$	$t = 2$	$t = 2.25$	$t = 2.5$
Computed	1.0000	1.5599	1.9044	2.0804	2.1164	2.0304	1.8351
Exact	1.0000	1.5555	1.8955	2.0673	2.0993	2.0097	1.8111

**FIGURE 8.4**

Approximate solution in Example 8.5.

```
% Exact solution
>> w_ex = matlabFunction(dsolve('t*D2w+Dw+2*t=0','w(1)=1','Dw(2.5)=-1'));
>> w_e = w_ex(t)
```

```
w_e =
```

```
1.0000    1.5555    1.8955    2.0673    2.0993    2.0097    1.8111
```

Table 8.3 summarizes the computed and exact values at the grid points.

In order to confirm that the solution obtained here satisfies the boundary condition at the right end, we run the same code with $t = 0.0125$ to generate a smooth solution curve. The result is shown in Figure 8.4.

8.6 MATLAB Built-In Function `bvp4c` for Boundary-Value Problems

MATLAB has a built-in function, `bvp4c`, which can numerically solve two-point BVPs. We will present the simplest form of this function here. The more intricate form includes

“options,” and solves the same problem with default parameters replaced by user-specified values, a structure created with the `bvpset` function; similar to the `odeset` function that we used in connection with the ode solvers in Section 7.9.

In order to use `bvp4c`, we need to have the BVP in the proper form. This process is best illustrated when applied to a basic, second-order problem.

8.6.1 Second-Order BVP

Consider, once again, a second-order differential equation in its most general form

$$y'' = f(x, y, y'), \quad a \leq x \leq b$$

subject to two boundary conditions, specified at the endpoints a and b . Using state variables $y_1 = y$ and $y_2 = y'$, we find the state-variable equations in vector form, as

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}), \quad \mathbf{y} = \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix}, \quad \mathbf{f} = \begin{Bmatrix} y_2 \\ f(x, y_1, y_2) \end{Bmatrix} \quad (8.11)$$

Once the system of first-order ODEs is formed, as it is here, the function `bvp4c` may be applied.

`bvp4c` Solve boundary value problems for ODEs by collocation.

`SOL = bvp4c(ODEFUN,BCFUN,SOLINIT)` integrates a system of ordinary differential equations of the form $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$ on the interval $[a, b]$, subject to general two-point boundary conditions of the form $\mathbf{bc}(\mathbf{y}(a), \mathbf{y}(b)) = 0$. `ODEFUN` and `BCFUN` are function handles. For a scalar X and a column vector Y , `ODEFUN(X,Y)` must return a column vector representing $\mathbf{f}(x, \mathbf{y})$. For column vectors YA and YB , `BCFUN(YA,YB)` must return a column vector representing $\mathbf{bc}(\mathbf{y}(a), \mathbf{y}(b))$. `SOLINIT` is a structure with fields

```
x -- ordered nodes of the initial mesh with
    SOLINIT.x(1) = a, SOLINIT.x(end) = b
y -- initial guess for the solution with SOLINIT.y(:,i)
    a guess for y(x(i)), the solution at the node SOLINIT.x(i)
```

`odefun` This is a user-defined function with function call `dydx = odefun(x,y)` where x is a scalar and y is the column vector comprised of the state variables; vector y in Equation 8.11. The function returns `dydx`, which is the column vector \mathbf{f} in Equation 8.11.

`bcfun` This is a user-defined function with function call `res = bcfun(ya,yb)`, where ya and yb are the column vectors of numerical solution estimates at $y(a)$ and $y(b)$. Note that

$$ya = \mathbf{y}(a) = \begin{Bmatrix} y_1(a) \\ y_2(a) \end{Bmatrix} = \begin{Bmatrix} y(a) \\ y'(a) \end{Bmatrix}, \quad yb = \mathbf{y}(b) = \begin{Bmatrix} y_1(b) \\ y_2(b) \end{Bmatrix} = \begin{Bmatrix} y(b) \\ y'(b) \end{Bmatrix}$$

Therefore, $y_a(1)$ and $y_b(1)$ represent the values of solution y at $x = a$ and $x = b$, while $y_a(2)$ and $y_b(2)$ represent the values of y' at $x = a$ and $x = b$. The function returns the so-called residual vector `res`, which is comprised of the residuals, that is, the differences between the numerical solution estimates and the prescribed boundary conditions. The function `bcfun` can be used in connection with any of the boundary conditions listed at the outset of this section:

Dirichlet boundary conditions: $y(a) = y_a, y(b) = y_b$

In this case, the `res` vector is in the form

$$\text{res} = \begin{Bmatrix} y_a(1) - y_a \\ y_b(1) - y_b \end{Bmatrix}$$

Neumann boundary conditions: $y'(a) = y'_a, y'(b) = y'_b$

In this case, the `res` vector is in the form

$$\text{res} = \begin{Bmatrix} y_a(2) - y'_a \\ y_b(2) - y'_b \end{Bmatrix}$$

Mixed boundary conditions: $c_1 y'(a) + c_2 y(a) = B_a, c_3 y'(b) + c_4 y(b) = B_b$

In this case, the `res` vector is in the form

$$\text{res} = \begin{Bmatrix} y_a(2) + \frac{c_2}{c_1} y_a(1) - \frac{B_a}{c_1} \\ y_b(2) + \frac{c_4}{c_3} y_b(1) - \frac{B_b}{c_3} \end{Bmatrix}, \quad c_1, c_3 \neq 0$$

For more simple mixed boundary conditions, such as $y(a) = y_a, y'(b) = y'_b$, we have

$$\text{res} = \begin{Bmatrix} y_a(1) - y_a \\ y_b(2) - y'_b \end{Bmatrix}$$

`Solinit` This contains the initial guess of the solution vector and is created by the MATLAB built-in function `bvpinit` with the syntax `solinit = bvpinit(x, yinit)`. The first input `x` is the vector of the initial points in the interval $[a, b]$. Normally, 10 points will suffice, hence `x = linspace(a, b, 10)`. The second input `yinit` is the vector of assigned initial guesses for the variables. In the case of the two-dimensional system in Equation 8.11, for example, `yinit` has two components: the first component is the initial guess for y , the second for y' . The initial guesses can also be assigned using a user-defined function. In that case, the function call modifies to `solinit = bvpinit(x, @yinit)`, and as before, `x = linspace(a, b, 10)`.

`Sol` This is the output of `sol = bvp4c(@odefun, @bcfun, solinit)` and consists of two fields:

`sol.x` is the vector of the x coordinates of the interior points used for calculations by MATLAB, generally different from the user-specified values in `bvpinit`.

`sol.y` is the matrix whose columns are the variables we are solving for. For example, in the case of the 2D system in Equation 8.11, there are two variables to

solve: y and y' . Then, the first column in `sol.y` represents the values for y at the interior points, and the second column consists of the values of y' at those points. The number of rows in `sol.y` is the same as the number of components in `sol.x`, namely, the points at which `bvp4c` solved the BVP.

EXAMPLE 8.6: `bvp4c` WITH DIRICHLET BOUNDARY CONDITIONS

Consider the BVP (with Dirichlet boundary conditions) in Examples 8.1 and 8.3:

$$\ddot{u} = 0.02u + 1, \quad u(0) = 10, \quad u(10) = 100$$

Replacing t by x , the ODE is written as $u'' = 0.02u + 1$. Selecting state variables $y_1 = u$ and $y_2 = u'$, we have

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}), \quad \mathbf{y} = \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix}, \quad \mathbf{f} = \begin{Bmatrix} y_2 \\ 0.02y_1 + 1 \end{Bmatrix}, \quad 0 \leq x \leq 10$$

The residual vector in this problem is

$$\mathbf{res} = \begin{Bmatrix} \mathbf{ya}(1) - 10 \\ \mathbf{yb}(1) - 100 \end{Bmatrix}$$

We now write a script that solves the BVP using `bvp4c` and follows the instructions given above.

```
x = linspace(0,10,10);
solinit = bvpinit(x,[0, 0.1]); % Assign initial values to y1 and y2
sol = bvp4c(@odefun8_6,@bcfun8_6,solinit); % Solve using bvp4c
t = sol.x; % Change the name of sol.x to t
y = sol.y; % Change the name of sol.y to y. Note that y is a 2-by-10 vector
u = y(1,:); % First row of y contains the values of y1, which is u
udot = y(2,:); % Second row of y gives the values of y2, which is udot
plot(t,u,'o') % Figure 8.5
```

```
function dydx = odefun8_6(x,y)
dydx = [y(2);0.02*y(1)+1];
```

```
function res = bcfun8_6(ya,yb)
res = [ya(1)-10;yb(1)-100];
```

Executing the script file generates the plot of the solution $u(t)$ in [Figure 8.5](#), which clearly agrees with what we obtained in [Figure 8.2](#), Example 8.1. It is also worth mentioning that the first element of the vector `udot` is

```
>> udot(1)
ans =
    1.4108
```

This is almost exactly what we obtained in Example 8.1 while using the shooting method; the value for the missing initial condition $\dot{u}(0)$ needed to construct an IVP whose solution agrees with the original BVP. The reported value of 1.4112 in Example 8.1 is more accurate because it was obtained using 20 equally-spaced points for the independent variable as opposed to 10 in the current example.

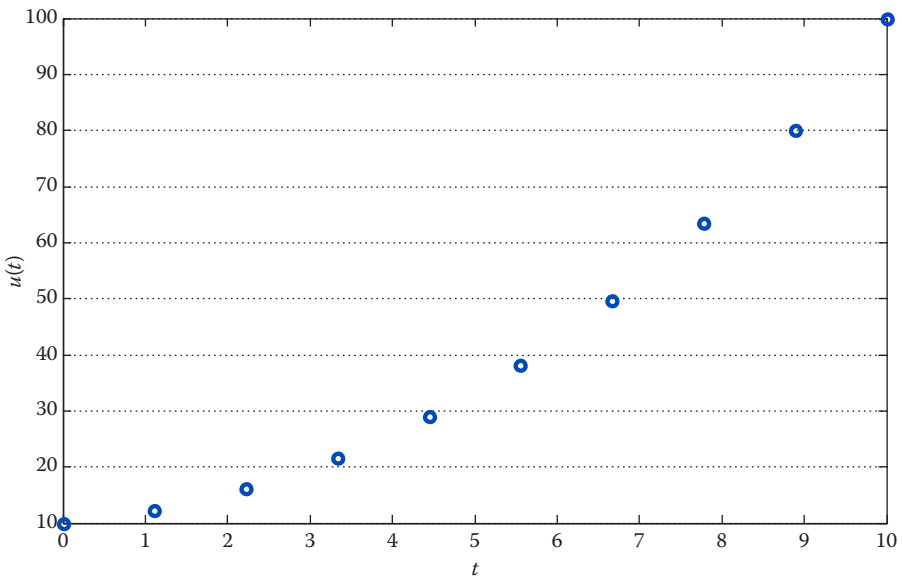


FIGURE 8.5

Solution $u(t)$ of the BVP in Example 8.6 using `bvp4c`.

EXAMPLE 8.7: `bvp4c` WITH MIXED BOUNDARY CONDITIONS

Consider the BVP (with mixed boundary conditions) in Example 8.5:

$$t\ddot{w} + \dot{w} + 2t = 0, \quad w(1) = 1, \quad \dot{w}(2.5) = -1$$

Replacing t by x , the ODE is written as $xw'' + w' + 2x = 0$. Selecting state variables $y_1 = w$ and $y_2 = w'$, we have

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}), \quad \mathbf{y} = \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix}, \quad \mathbf{f} = \begin{Bmatrix} y_2 \\ (-y_2 - 2x)/x \end{Bmatrix}, \quad 1 \leq x \leq 2.5$$

The residual vector in this problem is

$$\text{res} = \begin{Bmatrix} \mathbf{y}\mathbf{a}(1) - 1 \\ \mathbf{y}\mathbf{b}(2) + 1 \end{Bmatrix}$$

We now write a script file that solves the BVP using `bvp4c` and follows the instructions given above.

```
x = linspace(1,2.5,10);
solinit = bvpinit(x,[0.1, 0]); % Assign initial values to y1 and y2
sol = bvp4c(@odefun8_7,@bcfun8_7,solinit); % Solve using bvp4c
t = sol.x; % Change the name of sol.x to t
y = sol.y; % Change the name of sol.y to y. Note that y is a 2-by-10 vector
w = y(1,:); % First row of y contains the values of y1, which is w
wdot = y(2,:); % Second row of y gives the values of y2, which is wdot
plot(t,w,'o') % Figure 8.5
```

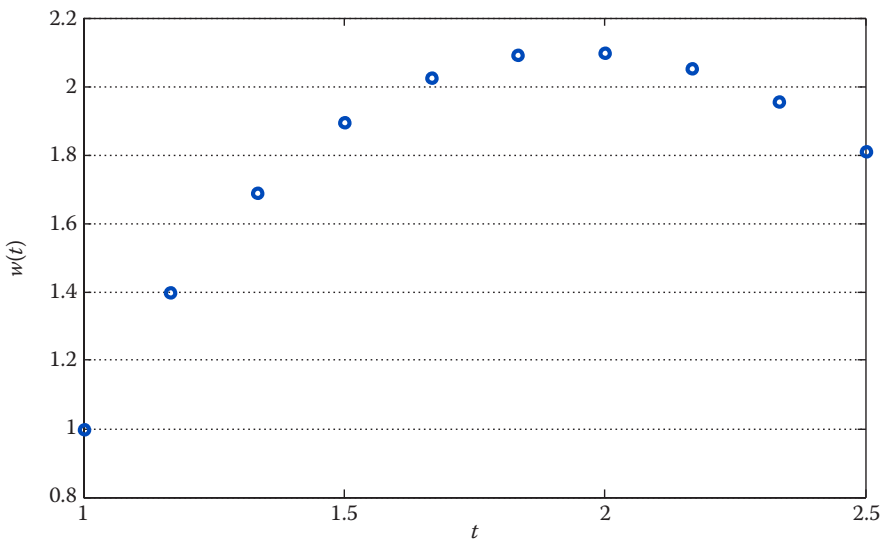


FIGURE 8.6

Solution $w(t)$ of the BVP in Example 8.7 using `bvp4c`.

```
function dydx = odefun8_7(x,y)
dydx = [y(2);-y(2)/x-2];

function res = bcfun8_7(ya,yb)
res = [ya(1)-1;yb(2)+1];
```


It is readily seen that the results are in agreement with those obtained in Example 8.5 using the finite-difference method.

PROBLEM SET (CHAPTER 8)

Shooting Method (Section 8.4)

 In Problems 1 through 4, solve the linear BVP using the shooting method.

1. $\ddot{u} + 3\dot{u} = t$, $u(0) = 1$, $u(5) = 2.4$
2. $2\ddot{u} + \dot{u} + 3u = \frac{2}{3}\sin(t/2)$, $u(0) = 0$, $u(1) = 1$
3. $3xw'' + w' = 0$, $w(1) = 2$, $w(3) = 1$
4. $(t-2)\ddot{w} - 2t\dot{w} + 4w = 0$, $w(0) = -1$, $w(1) = 2$

 In Problems 5 through 10, solve the nonlinear BVP using the shooting method combined with bisection method. Use the specified tolerance and maximum of 30 iterations. Plot the solution trajectory.

5. $\frac{1}{3}u\ddot{u} + \dot{u}^2 = 1$, $u(0) = 2$, $u(2) = 5$
tolerance: $\varepsilon = 10^{-6}$
6. $\ddot{u} + \frac{1}{2}u^2\dot{u} = t$, $u(0) = 1$, $u(2) = 3$
tolerance: $\varepsilon = 10^{-6}$


7. $\ddot{w} - e^{-2w} = 0$, $w(0) = 2$, $w(3) = 1$
tolerance: $\varepsilon = 10^{-4}$
8. $\ddot{y} + \dot{y} - 2y^2 = 0$, $y(0) = \frac{3}{2}$, $y(1) = 2$
tolerance: $\varepsilon = 10^{-4}$
9. $t\ddot{x} + \dot{x}^2 + 3t^2x = 0$, $x(1) = -1$, $x(4) = 1$
tolerance: $\varepsilon = 10^{-6}$
10. $\ddot{x} = x^3$, $x(1) = \frac{1}{3}$, $x(3) = \frac{5}{3}$
tolerance: $\varepsilon = 10^{-4}$

Finite-Difference Method (Section 8.5)


Linear Boundary-Value Problems

11.  Consider the linear BVP in Problem 1:


$$\ddot{u} + 3\dot{u} = t, \quad u(0) = 1, \quad u(5) = 2.4$$

- Solve using the finite-difference method (central difference) with $\Delta t = 1$.
 - Solve using the shooting method. In so doing, apply `RK4System` with step size of 0.25.
 - Solve using the finite-difference method with $\Delta t = 0.5$. Tabulate and compare all solution estimates, including those from (a) and (b), at $t = 1, 2, 3$, and 4. Also include the exact solution values at those points.
12.  Consider the linear BVP in Problem 2:

$$2\ddot{u} + \dot{u} + 3u = \frac{2}{3}\sin(t/2), \quad u(0) = 0, \quad u(1) = 1$$

- Solve using the finite-difference (central difference) method with $\Delta t = 0.25$.
 - Solve using the shooting method. In so doing, apply `RK4System` with step size of 0.05.
 - Solve using the finite-difference method with $\Delta t = 0.125$. Tabulate and compare all solution estimates, including those from (a) and (b), at $t = 0.25, 0.5$, and 0.75. Also include the exact solution values at those points.
13.  Consider the linear BVP


$$2xw'' + w' = 0, \quad w(1) = 3, \quad w(3) = 1$$

- Solve using the finite-difference (central difference) method with $\Delta x = 0.5$.
 - Solve using the finite-difference method with $\Delta x = 0.25$. Tabulate and compare all calculated solution values at $x = 1.5, 2.0$, and 2.5. Also include the exact solution values at those points.
14.  Consider the linear BVP

$$(2t - 1)\ddot{w} - 4t\dot{w} + 4w = 0, \quad w(1) = -1, \quad w(2) = 1$$


- a. Solve using the finite-difference (central difference) method with $\Delta t = 0.25$.
- b. Solve using the finite-difference method with $\Delta t = 0.125$. Tabulate and compare all calculated solution values at $t = 1.25, 1.5$, and 1.75 . Also include the exact solution values at those points.

Nonlinear Boundary-Value Problems; Use Central-Difference Formulas

15.  Consider the nonlinear BVP


$$u\ddot{u} + \dot{u}^2 = 0, \quad u(0) = 2, \quad u(2) = 3$$

Solve using the finite-difference (central difference) method with $\Delta t = 0.5$. When applying Newton's method, set the initial values of the unknown quantities to 2, and use $k_{\max} = 20$, $\text{tol} = 10^{-4}$.

16.  Consider the nonlinear BVP


$$2\ddot{w} - e^w = 0, \quad w(0) = 1, \quad w(2) = 0$$

Solve using the finite-difference (central difference) method with $\Delta t = 0.5$. When using Newton's method, set the initial values of the unknown quantities to 1, and let $k_{\max} = 20$, $\text{tol} = 10^{-4}$.

17.  Consider the nonlinear BVP


$$\ddot{u} + u^2\dot{u} = t, \quad u(0) = 1, \quad u(3) = 2$$

Solve using the finite-difference (central difference) method with $\Delta t = 0.5$. When using Newton's method, set the initial values of the unknown quantities to 1, and let $k_{\max} = 20$, $\text{tol} = 10^{-4}$.

18.  Consider the nonlinear BVP


$$\ddot{y} + \dot{y} - 2y^2 = 0, \quad y(0) = 1, \quad y(3) = 2$$

Solve using the finite-difference (central difference) method with $\Delta t = 0.5$. When applying Newton's method, set the initial values of the unknown quantities to 1, and use $k_{\max} = 10$, and $\text{tol} = 10^{-4}$.

19.  Consider the nonlinear BVP

$$\ddot{x} + t\dot{x}^2 + 3t^2x = 0, \quad x(0) = -1, \quad x(1) = 1$$

Solve using the finite-difference (central difference) method with $\Delta t = 0.25$. When using Newton's method, set the initial values of the unknown quantities to 0.2, and let $k_{\max} = 10$, and $\text{tol} = 10^{-4}$.

20.  Consider the nonlinear BVP

$$\ddot{w} - w^3 = 0, \quad w(1) = \frac{1}{2}, \quad w(3) = \frac{5}{2}$$

- Solve using the finite-difference (central difference) method with $\Delta t = 0.5$. When using Newton's method, set the initial values of the unknown quantities to 2, and let $k_{\max} = 10$, $\text{tol} = 10^{-4}$.
- Repeat (a) for $\Delta t = 0.25$. Tabulate the results of the two parts comparing solution estimates at $t = 1.5, 2, 2.5$.

Mixed Boundary Conditions

21.  Consider

$$\ddot{u} - \frac{1}{3}u = e^{-2t/3}, \quad u(0) = -1, \quad \dot{u}(0.5) = 1$$

Solve by the finite-difference method and $\Delta t = 0.1$. Use central-difference formula for the second-order derivative, and a three-point backward-difference formula for the first-order derivative in the right boundary condition, which also has second-order accuracy. Compare the computed results with the exact solution values at the grid points.

22.  Consider

$$t\ddot{y} + 2\dot{y} = te^{-2t}, \quad y(2) = -0.8, \quad \dot{y}(4) = 1.3$$

- Solve by the finite-difference method with $\Delta t = 0.5$. Use central-difference formula for the second-order derivative, and approximate the first-order derivative in the right boundary condition by a three-point backward-difference formula, which also has second-order accuracy. Compare the computed results with the exact solution values at the grid points.
- Repeat (a) for $\Delta t = 0.25$. Tabulate the values obtained in (a) and (b), as well as the exact values, at the interior grid points $t = 2.5, 3, 3.5$.


23.  Consider

$$t\ddot{w} + w + t^2 = 0, \quad w(0) = 0, \quad \dot{w}(1) = w(1)$$

- Solve by finite-difference method with $\Delta t = 0.25$. Use central-difference formula for the second-order derivative, and approximate the first-order derivative in the right boundary condition by a three-point backward-difference formula. Compare the results with the exact solution values at the grid points.
- Repeat (a) for $\Delta t = 0.125$. Tabulate the values obtained in (a) and (b), as well as the exact values, at the interior grid points $t = 0.25, 0.5, 0.75$.

24.  Consider

$$\ddot{y} + 2y = 0, \quad \dot{y}(0) = 0, \quad y(1) = 1$$

- a. Solve by the finite-difference method with $\Delta t = 0.125$. Use central-difference formula for the second-order derivative, and approximate the first-order derivative in the left boundary condition by a three-point forward-difference formula, which also has second-order accuracy. Compare the results with the exact solution values at the grid points.
 - b. Repeat (a) for $\Delta t = 0.0625$. Tabulate the values obtained in (a) and (b), as well as the exact values, at the interior grid points $t = 0.25, 0.5, 0.75$.
25.  Consider the nonlinear BVP with mixed boundary conditions

$$2\ddot{w} - e^w = 0, \quad \dot{w}(0) = 1, \quad \dot{w}(2) + w(2) = 0$$

Solve by finite-difference method with $\Delta t = 0.5$. Use central-difference formula for the second-order derivative, and approximate the first-order derivatives in the left and right boundary conditions by a three-point forward and a three-point backward-difference formula, respectively. Solve the ensuing nonlinear system of equations via Newton's method with all initial values set to 1, and $k_{\max} = 10$, and $\text{tol} = 10^{-4}$.

26.  Consider the nonlinear BVP with mixed boundary conditions

$$\ddot{w} - w^3 = 0, \quad w(1) = 0, \quad \dot{w}(2) + w(2) = 1$$

- a. Solve by finite-difference method with $\Delta t = 0.25$. Use central-difference formula for the second-order derivative, and approximate the first-order derivative in the right boundary condition by a three-point backward-difference formula. Solve the ensuing nonlinear system of equations via Newton's method with all initial values set to 1, and $k_{\max} = 10$, and $\text{tol} = 10^{-4}$.
- b. Repeat (a) for $\Delta t = 0.125$. Tabulate the values obtained in (a) and (b) at grid points $t = 1.25, 1.5, 1.75$.

MATLAB Built-In Function `bvp4c` for Boundary-Value Problems (Section 8.6)

 In Problems 27 through 30, solve the BVP using `bvp4c`, and plot the dependent variable versus the independent variable.


27. The BVP in Example 8.2:

$$T_{xx} - \beta T^4 - \alpha T + \gamma = 0, \quad T(0) = 500, \quad T(0.2) = 350, \quad T_{xx} = d^2T/dx^2$$

where $\alpha = 20$, $\beta = 10^{-8}$, and $\gamma = 5 \times 10^3$ with all parameters in consistent physical units.


28. $y''' + y' = y^4$, $y(0) = 0$, $y'(0) = 0.7$, $y(1) = 14$
29. $y'' + 0.8y = 0$, $y(0) + 2y'(0) = 1$, $y'(1) = 1$

30. $x^2y'' + xy' - y = 0$, $y(1) = -\frac{1}{2}$, $y'(e^2) = 1$

31.  The deflection y and rotation ψ of a uniform beam of length $L = 8$, pinned at both ends, are governed by

$$\begin{aligned} y' &= \psi \\ \psi' &= \frac{5x - x^2}{EI} \end{aligned} \quad \text{subject to} \quad \begin{aligned} y(0) &= 0 \\ y(8) &= 0 \end{aligned}$$

where EI is the flexural rigidity and is assumed to be $EI = 4000$. All parameter values are in consistent physical units. Using `bvp4c` find y and ψ , and plot them versus $0 \leq x \leq 8$ in two separate figures.

32.  In the analysis of free transverse vibrations of a uniform beam of length $L = 10$, simply supported (pinned) at both ends, we encounter a fourth-order ODE

$$\frac{d^4X}{dx^4} - 16X = 0$$

subject to

$$\begin{aligned} X(0) &= 0, & X(L) &= 0 \\ X''(0) &= 0, & X''(L) &= 0 \end{aligned}$$

Using `bvp4c`, find and plot $X(x)$.

Matrix Eigenvalue Problem

The matrix eigenvalue problem plays an important role in engineering applications. In vibration analysis, for example, eigenvalues are directly related to the system's natural frequencies, while the eigenvectors represent the mode shapes. Eigenvalues play an equally significant role in numerical methods. For example, in the iterative solution of linear systems via Jacobi and Gauss–Seidel methods (Chapter 4), the eigenvalues of the Jacobi iteration matrix, or the Gauss–Seidel iteration matrix, not only determine whether or not the respective iteration will converge to a solution, but they also establish the rate of convergence of the method. In this chapter, we will present numerical methods to approximate the eigenvalues and eigenvectors of a matrix.

9.1 Matrix Eigenvalue Problem

The eigenvalue problem (Chapter 1) associated with a square matrix $\mathbf{A}_{n \times n}$ is described by

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}, \quad \mathbf{v} \neq \mathbf{0}_{n \times 1} \quad (9.1)$$

A number λ for which Equation 9.1 has a nontrivial solution ($\mathbf{v} \neq \mathbf{0}_{n \times 1}$) is an eigenvalue or characteristic value of matrix \mathbf{A} . The corresponding solution $\mathbf{v} \neq \mathbf{0}$ of Equation 9.1 is the eigenvector or characteristic vector of \mathbf{A} corresponding to λ . The set of all eigenvalues of \mathbf{A} , denoted by $\lambda(\mathbf{A})$, is called the spectrum of \mathbf{A} .

9.2 Power Method: Estimation of the Dominant Eigenvalue

The eigenvalue of matrix \mathbf{A} with the largest magnitude is called the dominant eigenvalue of \mathbf{A} . The power method is an iterative method that estimates the dominant eigenvalue of a matrix \mathbf{A} and its corresponding eigenvector. The basic assumptions are:

- $\mathbf{A}_{n \times n}$ is a real matrix.
- \mathbf{A} has n eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$, where λ_1 is the dominant eigenvalue,

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \dots \geq |\lambda_n|$$

and the corresponding eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ are linearly independent.

- The dominant eigenvalue is real.

We present the power method as follows. Since $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ is assumed to be a linearly independent set, there exist constants c_1, c_2, \dots, c_n such that an arbitrary $n \times 1$ vector \mathbf{x}_0 can be uniquely expressed as

$$\mathbf{x}_0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \dots + c_n\mathbf{v}_n \quad (9.2)$$

The eigenvalue problem is

$$\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i \quad (i = 1, 2, \dots, n) \quad \xrightarrow{\text{Pre-multiply by A}} \quad \mathbf{A}(\mathbf{A}\mathbf{v}_i) = \mathbf{A}(\lambda_i\mathbf{v}_i) \quad \Rightarrow \quad \mathbf{A}^2\mathbf{v}_i = \lambda_i^2\mathbf{v}_i$$

In general, we have

$$\mathbf{A}^k\mathbf{v}_i = \lambda_i^k\mathbf{v}_i \quad (9.3)$$

Define the sequence of vectors

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{A}\mathbf{x}_0 \\ \mathbf{x}_2 &= \mathbf{A}\mathbf{x}_1 = \mathbf{A}^2\mathbf{x}_0 \\ &\dots \\ \mathbf{x}_k &= \mathbf{A}\mathbf{x}_{k-1} = \mathbf{A}^k\mathbf{x}_0 \end{aligned}$$

Therefore, by Equations 9.2 and 9.3,

$$\begin{aligned} \mathbf{x}_k &= \mathbf{A}^k\mathbf{x}_0 = \mathbf{A}^k[c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \dots + c_n\mathbf{v}_n] = c_1\lambda_1^k\mathbf{v}_1 + c_2\lambda_2^k\mathbf{v}_2 + \dots + c_n\lambda_n^k\mathbf{v}_n \\ &= \lambda_1^k \left[c_1\mathbf{v}_1 + c_2 \left(\frac{\lambda_2}{\lambda_1} \right)^k \mathbf{v}_2 + \dots + c_n \left(\frac{\lambda_n}{\lambda_1} \right)^k \mathbf{v}_n \right] \end{aligned} \quad (9.4)$$

Since λ_1 is the dominant eigenvalue, the ratios $\lambda_2/\lambda_1, \lambda_3/\lambda_1, \dots, \lambda_n/\lambda_1$ are less than 1 in magnitude, hence for a sufficiently large k , we have $\mathbf{x}_k \cong c_1\lambda_1^k\mathbf{v}_1$. Equation 9.4 can also be used to obtain $\mathbf{x}_{k+1} \cong c_1\lambda_1^{k+1}\mathbf{v}_1$. Thus,

$$\mathbf{x}_{k+1} \cong \lambda_1\mathbf{x}_k \quad (9.5)$$

Estimation of λ_1 will be based on Equation 9.5. Pre-multiply Equation 9.5 by \mathbf{x}_k^T to create scalars on both sides, and

$$\lambda_1 \cong \frac{\mathbf{x}_k^T \mathbf{x}_{k+1}}{\mathbf{x}_k^T \mathbf{x}_k} \quad (9.6)$$

The power method states that the sequence of scalars

$$\alpha_{k+1} = \frac{\mathbf{x}_k^T \mathbf{x}_{k+1}}{\mathbf{x}_k^T \mathbf{x}_k}, \quad \mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k \quad (9.7)$$

converges to the dominant eigenvalue λ_1 for sufficiently large k . In fact, it can be shown that the sequence $\{\alpha_k\}$ converges to λ_1 at roughly the same rate as $(\lambda_2/\lambda_1)^k \rightarrow 0$. Therefore, the larger the magnitude of λ_1 is compared to the next largest eigenvalue λ_2 , the faster the convergence. Also, because the components of vector \mathbf{x}_k grow rapidly, it is common practice to normalize \mathbf{x}_k , that is, divide each vector \mathbf{x}_k by its two-norm, $\|\mathbf{x}_k\|_2$. Consequently, the denominator in Equation 9.7 simply becomes $\mathbf{x}_k^T \mathbf{x}_k = 1$ in each step.

9.2.1 Different Cases of Dominant Eigenvalue

There are three possible scenarios involving the dominant eigenvalue(s) of a matrix:

- It may be unique; for instance, $\lambda(\mathbf{A}) = \boxed{-3}, 2, 1$
- It may be repeated; for example, $\lambda(\mathbf{A}) = \boxed{-3, -3}, 1$
- There may be two distinct dominant eigenvalues with opposite signs; for instance, $\lambda(\mathbf{A}) = \boxed{3, -3}, 1$

The power method handles the first case because one of its basic premises was that the dominant eigenvalue be unique. The power method can also find the dominant eigenvalue in the second case, but will not be able to determine that it is repeated; that will be handled by the deflation methods discussed later. In the third case, application of the power method normally results in a sequence of scalars that exhibit oscillatory behavior and do not converge to a single value. To remedy this, a combination of the inverse power and shifted power methods will be used to find the dominant eigenvalues. These techniques will be presented shortly.

9.2.2 Algorithm for the Power Method

Starting with a matrix $\mathbf{A}_{n \times n}$, an initial $n \times 1$ vector \mathbf{x}_1 , and an initial $\alpha_1 = 0$,

1. Normalize \mathbf{x}_1 to build a unit vector $\Rightarrow \mathbf{x}_1 = \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|_2}$.
- For $k = 1$ to kmax (maximum number of iterations),
2. Find $\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k$.
3. Calculate $\alpha_{k+1} = \mathbf{x}_k^T \mathbf{x}_{k+1}$.
4. Normalize $\mathbf{x}_{k+1} \Rightarrow \mathbf{x}_{k+1} = \frac{\mathbf{x}_{k+1}}{\|\mathbf{x}_{k+1}\|_2}$.
5. Terminating condition: if $|\alpha_{k+1} - \alpha_k| < \epsilon$ (prescribed tolerance), STOP. Otherwise, increment k to $k + 1$ and go to step 2.

When the iterations stop, α_{k+1} is the approximate dominant eigenvalue, and the unit vector \mathbf{x}_{k+1} is the corresponding eigenvector. Note that other terminating conditions may also be used in step 5. For example, the iterations can be forced to stop if two consecutive vectors satisfy $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_\infty < \epsilon$.

The user-defined function `PowerMethod` uses the power method to estimate the dominant eigenvalue and the associated eigenvector of a square matrix.

```

function [e_val, e_vec, k] = PowerMethod(A, x1, tol, kmax)
%
% PowerMethod approximates the dominant eigenvalue and the corresponding
% eigenvector of a square matrix.
%
% [e_val, e_vec, k] = PowerMethod(A, x1, tol, kmax), where
%
% A is an n-by-n matrix,
% x1 is the n-by-1 initial vector (default ones),
% tol is the tolerance (default 1e-4),
% kmax is the maximum number of iterations (default 50),
%
% e_val is the approximated dominant eigenvalue,
% e_vec is the corresponding eigenvector,
% k is the number of iterations required for convergence.
%
n = size(A,1);
if nargin<2 || isempty(x1), x1 = ones(n,1); end
if nargin<3 || isempty(tol), tol = 1e-4; end
if nargin<4 || isempty(kmax), kmax = 50; end
x(:,1) = x1./norm(x1, 2);
x(:,2) = A*x(:,1); alpha(2) = x(:,1).'*x(:,2);
x(:,2) = x(:,2)./norm(x(:,2),2);

for k = 2:kmax,
    x(:,k+1) = A*x(:,k); % Generate next vector
    alpha(k+1) = x(:,k).'*x(:,k+1);
    x(:,k+1) = x(:,k+1)./norm(x(:,k+1),2);
    if abs(alpha(k+1)-alpha(k)) < tol, % Terminating condition
        break
    end
end
e_val = alpha(end); e_vec = x(:,end);

```

If output k (number of iterations) is not needed in an application, the function can be executed as

```
[e_val, e_vec] = PowerMethod(A, x1, tol, kmax)
```

Also, if e_vec (eigenvector) is not needed, we can still execute

```
e_val = PowerMethod(A, x1, tol, kmax)
```

No other combination of outputs is possible. For instance, [e_val, k] still returns e_val and e_vec because when there are two outputs, the second output is automatically regarded as e_vec.

EXAMPLE 9.1: POWER METHOD

Consider

$$\mathbf{A} = \begin{bmatrix} 3 & -4 & -2 \\ -1 & 4 & 1 \\ 2 & -6 & -1 \end{bmatrix}$$

Starting with $\alpha_1 = 0$, $\mathbf{x}_1 = [1 \quad 1 \quad 1]^T$, and tolerance $\varepsilon = 10^{-4}$, we follow the algorithm outlined above:

$$\mathbf{x}_1 = \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|_2} = \frac{1}{\sqrt{3}} \begin{Bmatrix} 1 \\ 1 \\ 1 \end{Bmatrix} = \begin{Bmatrix} 0.5774 \\ 0.5774 \\ 0.5774 \end{Bmatrix}, \quad \mathbf{x}_2 = \mathbf{A}\mathbf{x}_1 = \begin{bmatrix} 3 & -4 & -2 \\ -1 & 4 & 1 \\ 2 & -6 & -1 \end{bmatrix} \begin{Bmatrix} 0.5774 \\ 0.5774 \\ 0.5774 \end{Bmatrix} = \begin{Bmatrix} -1.7321 \\ 2.3094 \\ -2.8868 \end{Bmatrix}$$

$$\alpha_2 = \mathbf{x}_1^T \mathbf{x}_2 = -1.3333$$

Normalize

$$\mathbf{x}_2 = \frac{\mathbf{x}_2}{\|\mathbf{x}_2\|_2} = \begin{Bmatrix} -0.4243 \\ 0.5657 \\ -0.7071 \end{Bmatrix}$$

Since $|\alpha_2 - \alpha_1| = 1.3333$, convergence is not observed yet and the iterations must continue.

$$\mathbf{x}_3 = \mathbf{A}\mathbf{x}_2 = \begin{bmatrix} 3 & -4 & -2 \\ -1 & 4 & 1 \\ 2 & -6 & -1 \end{bmatrix} \begin{Bmatrix} -0.4243 \\ 0.5657 \\ -0.7071 \end{Bmatrix} = \begin{Bmatrix} -2.1213 \\ 1.9799 \\ -3.5355 \end{Bmatrix}$$

$$\alpha_3 = \mathbf{x}_2^T \mathbf{x}_3 = 4.5200$$

Since $|\alpha_3 - \alpha_2| = 5.8533$, convergence is not observed yet and the iterations must continue. Normalize

$$\mathbf{x}_3 = \frac{\mathbf{x}_3}{\|\mathbf{x}_3\|_2} = \begin{Bmatrix} -0.4638 \\ 0.4329 \\ -0.7730 \end{Bmatrix}$$

and repeat the steps until the terminating condition $|\alpha_{k+1} - \alpha_k| < 10^{-4}$ is satisfied. The next few generated scalars are $\alpha_4 = 3.4742$, $\alpha_5 = 3.2083$, $\alpha_6 = 3.1097$, ..., and so on. Execution of the user-defined function `PowerMethod` yields

```
>> A = [3 -4 -2;-1 4 1;2 -6 -1];
>> [e_val, e_vec, k] = PowerMethod(A) % Default x1 and tol

e_val =

    3.0002

e_vec =

   -0.4083
    0.4082
   -0.8165

k =

    20
```

It takes 20 steps for the power method to find an estimate for the dominant eigenvalue $\lambda_1 = 3$ and its eigenvector. Note that the returned (unit vector) eigenvector estimate is equivalent to $[-1 \quad 1 \quad -2]^T$. Direct solution of the eigenvalue problem reveals that $\lambda(\mathbf{A}) = 3, 2, 1$ and the eigenvector associated with $\lambda = 3$ agrees with that obtained here.

9.3 Inverse Power Method: Estimation of the Smallest Eigenvalue

The smallest (in magnitude) eigenvalue of $\mathbf{A}_{n \times n}$ can be estimated by applying the power method to \mathbf{A}^{-1} , explained as follows. If the eigenvalues of \mathbf{A} are $\lambda_1, \lambda_2, \dots, \lambda_n$, then the eigenvalues of \mathbf{A}^{-1} are $1/\lambda_1, 1/\lambda_2, \dots, 1/\lambda_n$:

$$\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i \quad (i = 1, 2, \dots, n) \quad \xRightarrow{\text{Pre-multiply by } \mathbf{A}^{-1}} \quad (\mathbf{A}^{-1}\mathbf{A})\mathbf{v}_i = \mathbf{A}^{-1}\lambda_i\mathbf{v}_i \quad \xRightarrow[\text{Divide by } \lambda_i]{\mathbf{A}^{-1}\mathbf{A}=\mathbf{I}} \quad \mathbf{A}^{-1}\mathbf{v}_i = \frac{1}{\lambda_i}\mathbf{v}_i$$

The last equation describes the eigenvalue problem associated with \mathbf{A}^{-1} so that its eigenvalues are indeed $1/\lambda_i$ and the eigenvectors are \mathbf{v}_i , the same as those of \mathbf{A} corresponding to λ_i . Applying the power method to \mathbf{A}^{-1} yields the dominant $1/\lambda_i$ so that this particular λ_i has the smallest magnitude among all eigenvalues of \mathbf{A} .

EXAMPLE 9.2: INVERSE POWER METHOD

Consider the matrix in Example 9.1 so that

$$\mathbf{A}^{-1} = \frac{1}{6} \begin{bmatrix} 2 & 8 & 4 \\ 1 & 1 & -1 \\ -2 & 10 & 8 \end{bmatrix}$$

Applying the power method to \mathbf{A}^{-1} with $\alpha_1 = 0$ and $\mathbf{x}_1 = [1 \quad 1 \quad 1]^T$, we find

$$\mathbf{x}_1 = \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|_2} = \begin{Bmatrix} 0.5774 \\ 0.5774 \\ 0.5774 \end{Bmatrix}, \quad \mathbf{x}_2 = \mathbf{A}^{-1}\mathbf{x}_1 \quad \xRightarrow{\text{Normalize}} \quad \mathbf{x}_2 = \frac{\mathbf{x}_2}{\|\mathbf{x}_2\|_2} = \begin{Bmatrix} 0.6578 \\ 0.0470 \\ 0.7517 \end{Bmatrix}, \quad \alpha_2 = \mathbf{x}_1^T \mathbf{x}_2 = 0.8409$$

$$\mathbf{x}_3 = \mathbf{A}^{-1}\mathbf{x}_2 \quad \xRightarrow{\text{Normalize}} \quad \mathbf{x}_3 = \frac{\mathbf{x}_3}{\|\mathbf{x}_3\|_2} = \begin{Bmatrix} 0.6727 \\ -0.0067 \\ 0.7399 \end{Bmatrix}, \quad \alpha_3 = \mathbf{x}_2^T \mathbf{x}_3 = 0.9984$$

Executing `PowerMethod` as applied to \mathbf{A}^{-1} , we find

```
>> A = [3 -4 -2;-1 4 1;2 -6 -1]; Ai = inv(A);
>> [e_val, e_vec, k] = PowerMethod(Ai)
```

```
e_val =
    0.9999    % Dominant eigenvalue of A^-1
```

```
e_vec =
    0.7070
   -0.0001
    0.7072
```

```
k =
    12
```


After 12 iterations, the sequence of scalars α_k converges to the dominant eigenvalue of \mathbf{A}^{-1} , which is 1. Therefore, the reciprocal of this value, which also happens to be 1, is the smallest magnitude eigenvalue of \mathbf{A} . In addition, the sequence of unit vectors \mathbf{x}_k converges to a vector equivalent to $[1 \ 0 \ 1]^T$. Further inspection reveals that this is indeed the eigenvector of \mathbf{A} corresponding to $\lambda = 1$. In summary, the dominant eigenvalue of \mathbf{A} is $\lambda = 3$, by Example 9.1, and the smallest is $\lambda = 1$, as reported here.

9.4 Shifted Inverse Power Method: Estimation of the Eigenvalue Nearest a Specified Value

Once the largest or smallest eigenvalue of a matrix is known, the remaining eigenvalues can be approximated using the shifted inverse power method. In order to establish this method, we first recognize the fact that if the eigenvalues of \mathbf{A} are $\lambda_1, \lambda_2, \dots, \lambda_n$, then the eigenvalues of the matrix $\mathbf{A} - \alpha\mathbf{I}$ are $\lambda_1 - \alpha, \lambda_2 - \alpha, \dots, \lambda_n - \alpha$:

$$\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i \quad (i = 1, 2, \dots, n) \quad \xrightarrow{\text{Subtract } \alpha\mathbf{v}_i} \quad \mathbf{A}\mathbf{v}_i - \alpha\mathbf{v}_i = \lambda_i\mathbf{v}_i - \alpha\mathbf{v}_i \quad \Rightarrow \quad (\mathbf{A} - \alpha\mathbf{I})\mathbf{v}_i = (\lambda_i - \alpha)\mathbf{v}_i$$

This last equation is the eigenvalue problem associated with the matrix $\mathbf{A} - \alpha\mathbf{I}$; therefore, its eigenvalues are $\lambda_i - \alpha$ and the eigenvectors are \mathbf{v}_i , the same as those of \mathbf{A} corresponding to λ_i . Combining this with the fact that the eigenvalues of \mathbf{A}^{-1} are $1/\lambda_1, 1/\lambda_2, \dots, 1/\lambda_n$, we conclude that the eigenvalues of $(\mathbf{A} - \alpha\mathbf{I})^{-1}$ are

$$\mu_1 = \frac{1}{\lambda_1 - \alpha}, \quad \mu_2 = \frac{1}{\lambda_2 - \alpha}, \quad \dots, \quad \mu_n = \frac{1}{\lambda_n - \alpha} \quad (9.8)$$

while the eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ are the same as those for \mathbf{A} corresponding to $\lambda_1, \lambda_2, \dots, \lambda_n$. If the power method is applied to $(\mathbf{A} - \alpha\mathbf{I})^{-1}$, its dominant eigenvalue (say, μ_m) is estimated. Since $|\mu_m|$ is largest among all those listed in Equation 9.8, then $|\lambda_m - \alpha|$ must be the smallest among its counterparts, that is,

$$|\lambda_m - \alpha| \leq |\lambda_i - \alpha|, \quad i = 1, 2, \dots, n$$

This implies that the distance between λ_m and α is smaller than—or equal to—the distance between any λ_i and α . In conclusion, application of the power method to $(\mathbf{A} - \alpha\mathbf{I})^{-1}$ gives an estimate of λ_m which is closest to α than all the other eigenvalues of \mathbf{A} .

Inspired by this, we present the shifted inverse power method as follows. Let α be an arbitrary scalar, and \mathbf{x}_1 any initial vector. Generate the sequence of vectors

$$\mathbf{x}_{k+1} = (\mathbf{A} - \alpha\mathbf{I})^{-1}\mathbf{x}_k \quad (9.9)$$

and scalars

$$\beta_k = \frac{\mathbf{x}_k^T \mathbf{x}_{k+1}}{\mathbf{x}_k^T \mathbf{x}_k} \quad (9.10)$$

Then, $\beta_k \rightarrow \mu_m = 1/(\lambda_m - \alpha)$ so that $\lambda_m = (1/\mu_m) + \alpha$ is the eigenvalue of \mathbf{A} that is closest to α . Also, the sequence of vectors \mathbf{x}_k converges to the eigenvector corresponding to λ_m .

9.4.1 Notes on the Shifted Inverse Power Method

- The initial vector \mathbf{x}_1 and the subsequent vectors \mathbf{x}_k will be normalized to have a length of 1.
- Equation 9.9 is solved as $(\mathbf{A} - \alpha\mathbf{I})\mathbf{x}_{k+1} = \mathbf{x}_k$ using Doolittle's method (Section 4.4), which employs LU factorization of the coefficient matrix $\mathbf{A} - \alpha\mathbf{I}$. This proves useful, especially if α happens to be very close to an eigenvalue of \mathbf{A} , causing $\mathbf{A} - \alpha\mathbf{I}$ to be near singular.
- Setting $\alpha = 0$ leads to the estimation of the smallest magnitude eigenvalue of \mathbf{A} .

The user-defined function `ShiftInvPower` uses the shifted inverse power method to estimate the eigenvalue of a square matrix closest to a specified value. It also returns the eigenvector associated with the desired eigenvalue.

```
function [e_val, e_vec, k] = ShiftInvPower(A, alpha, x1, tol, kmax)
%
% ShiftInvPower uses the shifted inverse power method to find the
% eigenvalue of a matrix closest to a specified value. It also returns
% the eigenvector associated with this eigenvalue.
%
% [e_val, e_vec, k] = ShiftInvPower(A, alpha, x1, tol, kmax), where
%
% A is an n-by-n matrix,
% alpha is a specified value,
% x1 is the n-by-1 initial vector (default ones),
% tol is the tolerance (default 1e-4),
% kmax is the maximum number of iterations (default 50),
%
% e_val is the estimated eigenvalue,
% e_vec is the corresponding eigenvector,
% k is the number of iterations required for convergence.
%
n = size(A,1);
if nargin<3 || isempty(x1), x1 = ones(n,1); end
if nargin<4 || isempty(tol), tol = 1e-4; end
if nargin<5 || isempty(kmax), kmax = 50; end
x(:,1) = x1./norm(x1,2);
betas(1) = 0;
for k = 1:kmax,
    x(:,k+1) = DoolittleMethod(A-alpha*eye(n,n),x(:,k));
    betas(k+1) = x(:,k).'*x(:,k+1);
    x(:,k+1) = x(:,k+1)./norm(x(:,k+1),2);
    if abs(betas(k+1)-betas(k)) < tol, % Check for convergence
        break
    end
end
betas = betas(end);
e_val = 1/betas+alpha;
e_vec = x(:, end);
```

9.5 Shifted Power Method

The shifted power method is based on the fact that if the eigenvalues of \mathbf{A} are $\lambda_1, \lambda_2, \dots, \lambda_n$, then the eigenvalues of the matrix $\mathbf{A} - \alpha\mathbf{I}$ are $\lambda_1 - \alpha, \lambda_2 - \alpha, \dots, \lambda_n - \alpha$. In particular, the eigenvalues of $\mathbf{A} - \lambda_1\mathbf{I}$ are $0, \lambda_2 - \lambda_1, \dots, \lambda_n - \lambda_1$. If the power method is applied to $\mathbf{A} - \lambda_1\mathbf{I}$, the dominant eigenvalue of this matrix (say, μ_2) is found. But μ_2 is a member of the list $0, \lambda_2 - \lambda_1, \dots, \lambda_n - \lambda_1$. Without loss of generality, suppose $\mu_2 = \lambda_2 - \lambda_1$ so that $\lambda_2 = \lambda_1 + \mu_2$. This way, another eigenvalue of \mathbf{A} is estimated.

9.5.1 Strategy to Estimate All Eigenvalues of a Matrix

- Apply the power method to \mathbf{A} to find the dominant λ_d .
- Apply the shifted inverse power method with $\alpha = 0$ to \mathbf{A} to find the smallest λ_s .
- Apply the shifted power method to $\mathbf{A} - \lambda_d\mathbf{I}$ or $\mathbf{A} - \lambda_s\mathbf{I}$ to find at least one more eigenvalue.
- Apply the shifted inverse power method as many times as necessary with α adjusted so that an λ between any two available λ 's may be found.

EXAMPLE 9.3: POWER METHODS

Find all eigenvalues and eigenvectors of the following matrix using the power, shifted inverse power, and shifted power methods:

$$\mathbf{A} = \begin{bmatrix} 4 & -3 & 3 & -9 \\ -3 & 6 & -3 & 11 \\ 0 & 8 & -5 & 8 \\ 3 & -3 & 3 & -8 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{Bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{Bmatrix}$$

Solution

Apply the user-defined function `PowerMethod` to find the dominant eigenvalue of \mathbf{A} :

```
>> A = [4 -3 3 -9;-3 6 -3 11;0 8 -5 8;3 -3 3 -8];
>> x1 = [0;1;0;1]; % Initial vector
>> [e_val, e_vec] = PowerMethod(A, x1) % Default values for tol and kmax

e_val =

    -5.0000    % Dominant eigenvalue

e_vec =

    0.5000
   -0.5000
   -0.5000
    0.5000
```

The smallest magnitude eigenvalue of \mathbf{A} can be estimated by either applying the power method to \mathbf{A}^{-1} (see Example 9.2) or by applying the shifted inverse power method to \mathbf{A} with $\alpha = 0$.

```
>> [e_val, e_vec] = ShiftInvPower(A, 0, x1)

e_val =

    1.0001    % Smallest magnitude eigenvalue

e_vec =

   -0.8165
    0.4084
    0.0001
   -0.4082
```

The largest and smallest eigenvalues of \mathbf{A} are therefore -5 and 1 , respectively. To see if the remaining two eigenvalues are between these two values, we set α to be the average, $\alpha = (-5 + 1)/2 = -2$, and apply the shifted inverse power method. However, $\alpha = -2$ causes $\mathbf{A} - \alpha\mathbf{I}$ to be singular, meaning α is an eigenvalue of \mathbf{A} . Since we also need the eigenvector associated with this eigenvalue, we set α to a value close to -2 and apply the shifted inverse power:

```
>> [e_val, e_vec] = ShiftInvPower(A, -1.5, x1)    % alpha=-1.5

eigenval =

   -2.0000    % Third eigenvalue

eigenvec =

    0.5774
   -0.5774
    0.0000
    0.5774
```

We find the fourth eigenvalue using the shifted power method. Knowing $\lambda = -2$ is an eigenvalue of \mathbf{A} , we apply the power method to $\mathbf{A} + 2\mathbf{I}$:

```
>> A1 = A + 2*eye(4,4);
>> [e_val, e_vec] = PowerMethod(A1, x1)

e_val =

    5.0000    % Fourth eigenvalue = 5+(-2)=3

e_vec =

   -0.0000
    0.7071
    0.7071
    0.0000
```

By the reasoning behind the shifted power method, the fourth eigenvalue is $\lambda = -2 + 5 = 3$. In summary, all four eigenvalues and their eigenvectors are

$$\lambda_1 = -5, \mathbf{v}_1 = \begin{Bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{Bmatrix}, \lambda_2 = 1, \mathbf{v}_2 = \begin{Bmatrix} -2 \\ 1 \\ 0 \\ -1 \end{Bmatrix}, \lambda_3 = -2, \mathbf{v}_3 = \begin{Bmatrix} 1 \\ -1 \\ 0 \\ 1 \end{Bmatrix}, \lambda_4 = 3, \mathbf{v}_4 = \begin{Bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{Bmatrix}$$

9.6 MATLAB Built-In Function `eig`

The built-in function `eig` in MATLAB finds the eigenvalues and eigenvectors of a matrix. The function `eig` can be used in two different forms.

`eig` Eigenvalues and eigenvectors.

`E = eig(X)` is a vector containing the eigenvalues of a square matrix `X`.

`[V,D] = eig(X)` produces a diagonal matrix `D` of eigenvalues and a full matrix `V` whose columns are the corresponding eigenvectors so that $X*V = V*D$.

The first form `E = eig(A)` is used when only the eigenvalues of a matrix `A` are needed. If the eigenvalues, as well as the eigenvectors of `A` are desired, `[V,D] = eig(A)` is used. This returns a matrix `V` whose columns are the eigenvectors of `A`, each a unit vector, and a diagonal matrix `D` whose entries are the eigenvalues of `A` and whose order agrees with the columns of `V`. Applying the latter to the matrix in Example 9.3 yields

```
>> A = [4 -3 3 -9;-3 6 -3 11;0 8 -5 8;3 -3 3 -8];  
>> [V,D] = eig(A)
```

```
V =  
-0.8165 -0.5774 0.0000 -0.5000  
 0.4082  0.5774 0.7071  0.5000  
-0.0000 -0.0000 0.7071  0.5000  
-0.4082 -0.5774 0.0000 -0.5000
```

```
D =  
 1.0000         0         0         0  
         0 -2.0000         0         0  
         0         0  3.0000         0  
         0         0         0 -5.0000
```

The results clearly agree with those obtained earlier in Example 9.3.

9.7 Deflation Methods

In the last section, we learned how to estimate eigenvalues of a matrix by using different variations of the power method. Another tactic to find the eigenvalues of a matrix involves the idea of deflation. Suppose $A_{n \times n}$ has eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$, and one of them is available; for example, the dominant λ_1 obtained by the power method. The basic idea behind deflation is to generate an $(n-1) \times (n-1)$ matrix `B`, one size smaller than `A`, whose eigenvalues are $\lambda_2, \dots, \lambda_n$, meaning all the eigenvalues of `A` excluding the dominant λ_1 . We next focus on `B` and suppose its dominant eigenvalue is λ_2 , which is available through the use of the power method. With that, `B` is deflated to a yet smaller matrix, and so on. Although there are many proposed deflation methods, we will introduce the most commonly used one, known as Wielandt's deflation method.

9.7.1 Wielandt's Deflation Method

In order to deflate an $n \times n$ matrix \mathbf{A} to an $(n - 1) \times (n - 1)$ matrix \mathbf{B} , we must first construct an $n \times n$ matrix \mathbf{A}_1 whose eigenvalues are $0, \lambda_2, \dots, \lambda_n$. This is explained in the following theorem.

Theorem 9.1: Wielandt's Deflation Method

Suppose $\mathbf{A}_{n \times n}$ has eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$ and eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$. Assume that λ_1 and \mathbf{v}_1 are known and that the first component of \mathbf{v}_1 is nonzero,* which can be made into 1. If \mathbf{a}_1 is the first row of \mathbf{A} , then

$$\mathbf{A}_1 = \mathbf{A} - \mathbf{v}_1 \mathbf{a}_1 \tag{9.11}$$

has eigenvalues $0, \lambda_2, \dots, \lambda_n$.

Proof

Since the first entry of \mathbf{v}_1 is nonzero, by assumption, we will normalize \mathbf{v}_1 by dividing it by its first component. This causes the first entry of \mathbf{v}_1 to be 1. As a result, the first row of the matrix $\mathbf{v}_1 \mathbf{a}_1$ is simply the first row of \mathbf{A} , and \mathbf{A}_1 has the form in [Figure 9.1](#).

Because the entire first row of \mathbf{A}_1 is zero, \mathbf{A}_1 is singular and thus has at least one eigenvalue of 0. We next show that the remaining eigenvalues of \mathbf{A}_1 are $\lambda_2, \dots, \lambda_n$, the remaining eigenvalues of the original matrix \mathbf{A} . To do this, we realize that the eigenvectors of \mathbf{A} are either in the form

$$\mathbf{v}_i = \begin{cases} \text{Case (1)} \\ \begin{bmatrix} 1 \\ v_{i2} \\ \dots \\ v_{in} \end{bmatrix} \end{cases} \quad \text{or} \quad \begin{cases} \text{Case (2)} \\ \begin{bmatrix} 0 \\ v_{i2} \\ \dots \\ v_{in} \end{bmatrix} \end{cases}, \quad i = 2, 3, \dots, n$$

$$\mathbf{A}_1 = \begin{bmatrix} 0 & 0 & \dots & 0 \\ b_{21} & b_{22} & \dots & b_{2n} \\ b_{31} & b_{32} & \dots & b_{3n} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{bmatrix}$$

← \mathbf{B}

FIGURE 9.1

Matrix generated by Wielandt's deflation method.

* The case when the first entry of is zero can be treated similarly, as in Example 9.5.

Case (1) Consider

$$\begin{aligned}\mathbf{A}_1(\mathbf{v}_1 - \mathbf{v}_i) &= (\mathbf{A} - \mathbf{v}_1\mathbf{a}_1)(\mathbf{v}_1 - \mathbf{v}_i) \\ &= \mathbf{A}(\mathbf{v}_1 - \mathbf{v}_i) - \mathbf{v}_1\mathbf{a}_1(\mathbf{v}_1 - \mathbf{v}_i)\end{aligned}\tag{9.12}$$

In the second term on the right side of Equation 9.12, note that $\mathbf{a}_1\mathbf{v}_k$ simply gives the first component of $\mathbf{A}\mathbf{v}_k$. Noting $\mathbf{A}\mathbf{v}_k = \lambda_k\mathbf{v}_k$ and the nature of the eigenvectors in Case (1), we have $\mathbf{a}_1\mathbf{v}_1 = \lambda_1$ and $\mathbf{a}_1\mathbf{v}_i = \lambda_i$. Using these in Equation 9.12,

$$\mathbf{A}_1(\mathbf{v}_1 - \mathbf{v}_i) = \lambda_1\mathbf{v}_1 - \lambda_i\mathbf{v}_i - \mathbf{v}_1(\lambda_1 - \lambda_i) = \lambda_i(\mathbf{v}_1 - \mathbf{v}_i)$$

Letting $\mathbf{u}_i = \mathbf{v}_1 - \mathbf{v}_i$, the above equation reads $\mathbf{A}_1\mathbf{u}_i = \lambda_i\mathbf{u}_i$. Since this is the eigenvalue problem associated with \mathbf{A}_1 , the proof of Case (1) is complete. And, the eigenvectors of \mathbf{A}_1 corresponding to $\lambda_2, \dots, \lambda_n$ are in the form $\mathbf{u}_i = \mathbf{v}_1 - \mathbf{v}_i$. Therefore, all of these eigenvectors have a first component of zero.

Case (2) Consider

$$\begin{aligned}\mathbf{A}_1\mathbf{v}_i &= (\mathbf{A} - \mathbf{v}_1\mathbf{a}_1)\mathbf{v}_i \\ &= \lambda_i\mathbf{v}_i - \mathbf{v}_1\mathbf{a}_1\mathbf{v}_i\end{aligned}\tag{9.13}$$

Following an earlier reasoning, the term $\mathbf{a}_1\mathbf{v}_i$ is the first component of $\mathbf{A}\mathbf{v}_i$. Noting $\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i$ and the nature of the eigenvectors in Case (2), we conclude that $\mathbf{a}_1\mathbf{v}_i = 0$. Then, Equation 9.13 becomes $\mathbf{A}_1\mathbf{v}_i = \lambda_i\mathbf{v}_i$, indicating $\lambda_2, \dots, \lambda_n$ are eigenvalues of \mathbf{A}_1 with corresponding eigenvectors $\mathbf{v}_2, \dots, \mathbf{v}_n$. With that, the proof is complete. Once again, note that the eigenvectors all have a first component of zero. ■

9.7.2 Deflation Process

While proving Theorem 9.1, we learned that in both cases the eigenvectors of \mathbf{A}_1 corresponding to $\lambda_2, \dots, \lambda_n$ all have zeros in their first components. Thus, the first column of \mathbf{A}_1 can be dropped all together. As a result, the $(n-1) \times (n-1)$ block of \mathbf{A}_1 , called \mathbf{B} in Figure 9.1, must have eigenvalues $\lambda_2, \dots, \lambda_n$. Therefore, the problem reduces to finding the eigenvalues of \mathbf{B} , a matrix one size smaller than the original \mathbf{A} . The power method can be applied to estimate the dominant eigenvalue and the corresponding eigenvector of \mathbf{B} , which in turn may be deflated further, and so on.

EXAMPLE 9.4: WIELANDT'S DEFLATION METHOD

Consider the 4×4 matrix in Example 9.3:

$$\mathbf{A} = \begin{bmatrix} 4 & -3 & 3 & -9 \\ -3 & 6 & -3 & 11 \\ 0 & 8 & -5 & 8 \\ 3 & -3 & 3 & -8 \end{bmatrix}$$

Using the power method, the dominant eigenvalue of \mathbf{A} and its eigenvector are obtained as

$$\lambda_1 = -5, \quad \mathbf{v}_1 = \begin{Bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{Bmatrix}$$

The first component of \mathbf{v}_1 is 1, hence \mathbf{v}_1 is already normalized. Proceeding with Wielandt's deflation method,

$$\begin{aligned} \mathbf{A}_1 = \mathbf{A} - \mathbf{v}_1 \mathbf{a}_1 &= \begin{bmatrix} 4 & -3 & 3 & -9 \\ -3 & 6 & -3 & 11 \\ 0 & 8 & -5 & 8 \\ 3 & -3 & 3 & -8 \end{bmatrix} - \begin{Bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{Bmatrix} \begin{bmatrix} 4 & -3 & 3 & -9 \end{bmatrix} \\ &= \begin{bmatrix} 4 & -3 & 3 & -9 \\ -3 & 6 & -3 & 11 \\ 0 & 8 & -5 & 8 \\ 3 & -3 & 3 & -8 \end{bmatrix} - \begin{bmatrix} 4 & -3 & 3 & -9 \\ -4 & 3 & -3 & 9 \\ -4 & 3 & -3 & 9 \\ 4 & -3 & 3 & -9 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 3 & 0 & 2 \\ 4 & 5 & -2 & -1 \\ -1 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Eliminating the first column and the first row of \mathbf{A}_1 yields the new, smaller matrix

$$\mathbf{B} = \begin{bmatrix} 3 & 0 & 2 \\ 5 & -2 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

Application of the power method to \mathbf{B} produces the dominant eigenvalue $\lambda = 3$ and its eigenvector $\mathbf{v} = [1 \ 1 \ 0]^T$. Note that this is not an eigenvector of \mathbf{A} corresponding to $\lambda = 3$, which would have been a 4×1 vector. Repeating the deflation process, this time applied to \mathbf{B} , we find

$$\mathbf{B}_1 = \mathbf{B} - \mathbf{v} \mathbf{b}_1 = \begin{bmatrix} 3 & 0 & 2 \\ 5 & -2 & -1 \\ 0 & 0 & 1 \end{bmatrix} - \begin{Bmatrix} 1 \\ 1 \\ 0 \end{Bmatrix} \begin{bmatrix} 3 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 2 & -2 & -3 \\ 0 & 0 & 1 \end{bmatrix} \xrightarrow[\text{first row, first column}]{\text{Eliminate}} \mathbf{C} = \begin{bmatrix} -2 & -3 \\ 0 & 1 \end{bmatrix}$$

Since \mathbf{C} is upper triangular, its eigenvalues are the diagonal entries -2 and 1 . In summary, the four eigenvalues of the original matrix \mathbf{A} are $-5, 3, -2, 1$, as asserted.

EXAMPLE 9.5: THE FIRST COMPONENT OF \mathbf{V}_1 IS ZERO

Consider

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & -1 \\ 1 & 3 & 2 \\ -1 & 2 & 3 \end{bmatrix}$$

The power method gives the dominant eigenvalue $\lambda_1 = 5$ and eigenvector

$$\mathbf{v}_1 = \begin{Bmatrix} 0 \\ 1 \\ 1 \end{Bmatrix}$$

The first component is zero, and the second component is made into 1. The matrix \mathbf{A}_1 is now formed differently than Equation 9.11, as follows. Because the second component of \mathbf{v}_1 is 1, we consider \mathbf{a}_2 , the second row of \mathbf{A} , and perform

$$\mathbf{A}_1 = \mathbf{A} - \mathbf{v}_1 \mathbf{a}_2 = \begin{bmatrix} 2 & 1 & -1 \\ 1 & 3 & 2 \\ -1 & 2 & 3 \end{bmatrix} - \begin{Bmatrix} 0 \\ 1 \\ 1 \end{Bmatrix} [1 \quad 3 \quad 2] = \begin{bmatrix} 2 & 1 & -1 \\ 0 & 0 & 0 \\ -2 & -1 & 1 \end{bmatrix}$$

Eliminating the second row and the second column of \mathbf{A}_1 yields

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ -2 & 1 \end{bmatrix}$$

The two eigenvalues of \mathbf{B} are subsequently found to be 0, 3. In conclusion, the three eigenvalues of the original matrix \mathbf{A} are 5, 3, 0, which may be directly verified.

9.8 Householder Tridiagonalization and QR Factorization Methods

The methods presented so far to estimate eigenvalues and eigenvectors of a matrix can be tedious and are also prone to round-off errors, the latter particularly evident in the case of repeated application of power and deflation methods. The deflation process relies greatly on the available eigenvalue and its corresponding eigenvector, which are often provided by the power method. But because these are only estimates, the entries of the ensuing deflated matrix are also not exact. This approximated matrix is then subjected to the power method, which approximates its dominant eigenvalue, causing an accumulated round-off error. Therefore, repeated application of this process can pose serious round-off problems.

Other, more common, techniques to estimate the eigenvalues of a matrix are mostly two-step methods. In the first step, the original matrix is transformed into a simpler form, such as tridiagonal, which has the same eigenvalues as the original matrix. In the second step, the eigenvalues of this simpler matrix are found iteratively. A majority of these methods are designed to specifically handle symmetric matrices. Jacobi's method, for example, transforms a symmetric matrix into a diagonal one. This method, however, is not very efficient because as it zeros out an off-diagonal entry, it often creates a new, nonzero entry at the location where a zero was previously generated. A more polished technique is Given's method, which transforms a symmetric matrix into a tridiagonal matrix. It should be mentioned that Given's method can also be applied to a general, nonsymmetric matrix, but in this case, the original matrix is transformed into a special matrix that is no longer tridiagonal but one known as the Hessenberg matrix, discussed later in this section.

The most efficient and commonly used method, however, is Householder's method, which also transforms a symmetric matrix into a tridiagonal matrix. Like Given's method, Householder's method also applies to nonsymmetric matrices, transforming them into the Hessenberg form. The outcome of Householder's method is then subjected to repeated applications of QR factorization—covered later in this section—to reduce it to a matrix whose off-diagonal elements are considerably smaller than its diagonal entries. Ultimately, these diagonal entries serve as estimates of the eigenvalues of the original matrix. Householder's method is a so-called similarity transformation method. Recall from [Chapter 1](#) that matrices $\mathbf{A}_{n \times n}$ and $\mathbf{B}_{n \times n}$ are similar if there exists a non-singular matrix $\mathbf{S}_{n \times n}$ such that

$$\mathbf{B} = \mathbf{S}^{-1}\mathbf{A}\mathbf{S}$$

We say \mathbf{B} is obtained through a similarity transformation of \mathbf{A} . Similarity transformations preserve eigenvalues, that is, \mathbf{A} and \mathbf{B} have exactly the same characteristic polynomial, and hence the same eigenvalues.

9.8.1 Householder's Tridiagonalization Method (Symmetric Matrices)

Let $\mathbf{A} = [a_{ij}]_{n \times n}$ be a real, symmetric matrix whose eigenvalues are $\lambda_1, \lambda_2, \dots, \lambda_n$. Householder's method uses $n - 2$ successive similarity transformations to reduce \mathbf{A} into a tridiagonal matrix \mathbf{T} . Let $\mathbf{P}_1, \mathbf{P}_2, \dots, \mathbf{P}_{n-2}$ denote the matrices used in this process where each \mathbf{P}_k is symmetric and orthogonal, that is,

$$\mathbf{P}_k = \mathbf{P}_k^T = \mathbf{P}_k^{-1} \quad (k = 1, 2, \dots, n - 2)$$

Generate a sequence of matrices \mathbf{A}_k ($k = 1, 2, \dots, n - 2$), as

$$\begin{aligned} \mathbf{A}_0 &= \mathbf{A} \\ \mathbf{A}_1 &= \mathbf{P}_1^{-1}\mathbf{A}_0\mathbf{P}_1 = \mathbf{P}_1\mathbf{A}_0\mathbf{P}_1 \\ \mathbf{A}_2 &= \mathbf{P}_2\mathbf{A}_1\mathbf{P}_2 \\ &\dots \\ \mathbf{A}_{n-3} &= \mathbf{P}_{n-3}\mathbf{A}_{n-4}\mathbf{P}_{n-3} \\ \mathbf{A}_{n-2} &= \mathbf{P}_{n-2}\mathbf{A}_{n-3}\mathbf{P}_{n-2} \end{aligned} \tag{9.14}$$

In the first iteration, we create zeros in the appropriate slots in the first row and the first column of \mathbf{A} to obtain a new matrix $\mathbf{A}_1 = [a_{ij}^{(1)}]$, as shown below. Then, in the second iteration, zeros are generated in the appropriate locations in the second row and the second column of \mathbf{A}_1 to obtain $\mathbf{A}_2 = [a_{ij}^{(2)}]$, shown below.

$$\mathbf{A}_1 = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & 0 & \dots & 0 \\ a_{21}^{(1)} & a_{22}^{(1)} & \dots & \dots & a_{2n}^{(1)} \\ 0 & a_{32}^{(1)} & \dots & \dots & a_{3n}^{(1)} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & a_{n2}^{(1)} & \dots & \dots & a_{nn}^{(1)} \end{bmatrix}, \quad \mathbf{A}_2 = \begin{bmatrix} a_{11}^{(2)} & a_{12}^{(2)} & 0 & \dots & 0 \\ a_{21}^{(2)} & a_{22}^{(2)} & a_{23}^{(2)} & \dots & 0 \\ 0 & a_{32}^{(2)} & \dots & \dots & a_{3n}^{(2)} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & a_{n3}^{(2)} & \dots & a_{nn}^{(2)} \end{bmatrix}$$

Performing this process $n - 2$ times, yields a tridiagonal matrix $\mathbf{A}_{n-2} = [a_{ij}^{(n-2)}]$, where

$$\mathbf{T} = \mathbf{A}_{n-2} = \begin{bmatrix} a_{11}^{(n-2)} & a_{12}^{(n-2)} & 0 & \dots & 0 \\ a_{21}^{(n-2)} & a_{22}^{(n-2)} & a_{23}^{(n-2)} & 0 & \dots \\ 0 & a_{32}^{(n-2)} & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & a_{n-1,n}^{(n-2)} \\ 0 & 0 & 0 & a_{n,n-1}^{(n-2)} & a_{nn}^{(n-2)} \end{bmatrix} \quad (9.15)$$

9.8.2 Determination of Symmetric Orthogonal \mathbf{P}_k ($k = 1, 2, \dots, n - 2$)

Each matrix \mathbf{P}_k is defined by

$$\mathbf{P}_k = \mathbf{I} - 2\mathbf{v}_k\mathbf{v}_k^T, \quad k = 1, 2, \dots, n - 2 \quad (9.16)$$

where \mathbf{v}_k is a unit vector ($\mathbf{v}_k^T\mathbf{v}_k = 1$), the first k components of which are zero. Moreover, it can be verified that each \mathbf{P}_k is symmetric and orthogonal. To further understand the structure of the unit vectors \mathbf{v}_k , let us consider \mathbf{v}_1 first:

$$\mathbf{v}_1 = \begin{Bmatrix} 0 \\ v_{21} \\ \dots \\ v_{n1} \end{Bmatrix} \quad \text{where} \quad \begin{cases} v_{21} = \sqrt{\frac{1}{2} \left(1 + \frac{|a_{21}|}{\Sigma_1} \right)} & \text{where } \Sigma_1 = \sqrt{a_{21}^2 + a_{31}^2 + \dots + a_{n1}^2} \\ v_{i1} = \begin{cases} \frac{a_{i1}}{2v_{21}\Sigma_1} & \text{if } a_{21} \geq 0 \\ \frac{-a_{i1}}{2v_{21}\Sigma_1} & \text{if } a_{21} < 0 \end{cases} & , \quad i = 3, 4, \dots, n \end{cases} \quad (9.17)$$

Note that since \mathbf{v}_1 is used to form \mathbf{P}_1 , which in turn is involved in the first iteration of Equation 9.14, the entries of $\mathbf{A} = \mathbf{A}_0$ are used in Equation 9.17 in the construction of \mathbf{v}_1 . Similarly, we construct the unit vector \mathbf{v}_2 this time using the entries of $\mathbf{A}_1 = [a_{ij}^{(1)}]$ from the second iteration of Equation 9.14,

$$\mathbf{v}_2 = \begin{Bmatrix} 0 \\ 0 \\ v_{32} \\ \dots \\ v_{n2} \end{Bmatrix} \quad \text{where} \quad \begin{cases} v_{32} = \sqrt{\frac{1}{2} \left(1 + \frac{|a_{32}^{(1)}|}{\Sigma_2} \right)} & \text{where } \Sigma_2 = \sqrt{[a_{32}^{(1)}]^2 + [a_{42}^{(1)}]^2 + \dots + [a_{n2}^{(1)}]^2} \\ v_{i2} = \begin{cases} \frac{a_{i2}^{(1)}}{2v_{32}\Sigma_2} & \text{if } a_{32}^{(1)} \geq 0 \\ \frac{-a_{i2}^{(1)}}{2v_{32}\Sigma_2} & \text{if } a_{32}^{(1)} < 0 \end{cases} & , \quad i = 4, 5, \dots, n \end{cases} \quad (9.18)$$

Continuing this way \mathbf{v}_{n-2} is constructed, and subsequently, the tridiagonal matrix $\mathbf{T} = \mathbf{A}_{n-2}$ is obtained. Now, there are two possible scenarios: If the entries of \mathbf{T} along the lower- and upper-diagonals are considerably smaller in magnitude than those along the main diagonal, then \mathbf{T} is regarded as almost diagonal, and the diagonal entries roughly approximate its eigenvalues, hence those of the original matrix \mathbf{A} . If not, we proceed to

further transform \mathbf{T} into a tridiagonal matrix whose diagonal elements dominate all other entries. This will be accomplished using the QR factorization.

The user-defined function `Householder` uses Householder's method to transform a real, symmetric matrix into a tridiagonal matrix.

```
function T = Householder(A)
%
% Householder uses Householder's method to transform a symmetric matrix
% into a tridiagonal matrix.
%
% T = Householder(A), where
%
%     A is an n-by-n real, symmetric matrix,
%
%     T is an n-by-n tridiagonal matrix.
%
N = size(A,1);
for n = 1:N-2,
    S = sqrt(A(n+1:end,n)'*A(n+1:end,n)); % Compute sigma
    v(1:N,1) = 0; % Set initial set of entries to 0
    v(n+1) = sqrt((1+abs(A(n+1,n)))/S)/2; % First non-zero entry
    sn = sign(A(n+1,n)); % Determine sign of relevant entry
    v(n+2:N) = sn*A(n+2:end,n)/2/v(n+1)/S; % Compute remaining entries
    P = eye(N)-2*(v*v'); % Compute the symmetric, orthogonal matrices
    A = P\A*P; % Compute sequence of matrices
end
T = A;
```

EXAMPLE 9.6: HOUSEHOLDER'S METHOD

Consider

$$\mathbf{A} = \begin{bmatrix} 4 & 4 & 1 & 1 \\ 4 & 4 & 1 & 1 \\ 1 & 1 & 3 & 2 \\ 1 & 1 & 2 & 3 \end{bmatrix}$$

Since $n = 4$, matrices \mathbf{A}_1 and \mathbf{A}_2 are generated by Equation 9.14 using \mathbf{P}_1 and \mathbf{P}_2 , as follows. Form the unit vectors \mathbf{v}_1 and \mathbf{v}_2 by Equations 9.17 and 9.18, respectively. First,

$$\mathbf{v}_1 = \begin{Bmatrix} 0 \\ v_{21} \\ v_{31} \\ v_{41} \end{Bmatrix}, \quad \Sigma_1 = \sqrt{18}, \quad \begin{matrix} v_{21} = 0.9856 \\ v_{31} = 0.1196 = v_{41} \end{matrix} \Rightarrow \mathbf{v}_1 = \begin{Bmatrix} 0 \\ 0.9856 \\ 0.1196 \\ 0.1196 \end{Bmatrix}$$

By Equation 9.16, we find $\mathbf{P}_1 = \mathbf{I} - 2\mathbf{v}_1\mathbf{v}_1^T$, and subsequently,

$$\mathbf{A}_1 = \mathbf{P}_1\mathbf{A}\mathbf{P}_1 = \begin{bmatrix} 4 & -4.2426 & 0 & 0 \\ -4.2426 & 5 & -1 & -1 \\ 0 & -1 & 2.5 & 1.5 \\ 0 & -1 & 1.5 & 2.5 \end{bmatrix} \quad \text{so that} \quad \begin{cases} a_{32}^{(1)} = -1 \\ a_{42}^{(1)} = -1 \end{cases}$$

Using this in Equation 9.18,

$$\mathbf{v}_2 = \begin{Bmatrix} 0 \\ 0 \\ v_{32} \\ v_{42} \end{Bmatrix}, \quad \Sigma_2 = \sqrt{2}, \quad \begin{matrix} v_{32} = 0.9239 \\ v_{42} = 0.3827 \end{matrix} \Rightarrow \mathbf{v}_2 = \begin{Bmatrix} 0 \\ 0 \\ 0.9239 \\ 0.3827 \end{Bmatrix}$$

Form $\mathbf{P}_2 = \mathbf{I} - 2\mathbf{v}_2\mathbf{v}_2^T$, and

$$\mathbf{A}_2 = \mathbf{P}_2\mathbf{A}_1\mathbf{P}_2 = \begin{bmatrix} 4 & -4.2426 & 0 & 0 \\ -4.2426 & 5 & 1.4142 & 0 \\ 0 & 1.4142 & 4 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which is symmetric and tridiagonal, as projected. Executing the user-defined function Householder will confirm this.

```
>> A = [4 4 1 1; 4 4 1 1; 1 1 3 2; 1 1 2 3];
>> T = Householder(A)
```

T =

```
 4.0000  -4.2426  0.0000  0.0000
-4.2426   5.0000  1.4142   0
-0.0000   1.4142  4.0000  0.0000
 0.0000   0.0000  0.0000  1.0000
```

9.8.3 QR Factorization Method

Once a special matrix such as tridiagonal matrix \mathbf{T} is obtained via Householder's method, the goal is to transform it into a new tridiagonal matrix whose off-diagonal entries are considerably smaller (in magnitude) than the diagonal ones. For this purpose, we employ the QR factorization (or decomposition) method. This is based on the fact that any matrix \mathbf{M} can be decomposed into a product, $\mathbf{M} = \mathbf{QR}$ where \mathbf{Q} is orthogonal and \mathbf{R} is upper triangular.

Start the process by setting $\mathbf{T}_0 = \mathbf{T}$, and factorize it as $\mathbf{T}_0 = \mathbf{Q}_0\mathbf{R}_0$. Since \mathbf{Q}_0 and \mathbf{R}_0 are now available, we multiply them in reverse order to form a new matrix $\mathbf{R}_0\mathbf{Q}_0 = \mathbf{T}_1$. Then, apply the QR factorization to \mathbf{T}_1 to achieve $\mathbf{T}_1 = \mathbf{Q}_1\mathbf{R}_1$, multiply \mathbf{Q}_1 and \mathbf{R}_1 in reverse order to create $\mathbf{T}_2 = \mathbf{R}_1\mathbf{Q}_1$, and so on. In this manner, a sequence \mathbf{T}_k of tridiagonal matrices is generated, as

$$\begin{aligned} \mathbf{T}_0 &= \mathbf{T} = \mathbf{Q}_0\mathbf{R}_0 & \mathbf{T}_1 &= \mathbf{R}_0\mathbf{Q}_0 \\ \mathbf{T}_1 &= \mathbf{Q}_1\mathbf{R}_1 & \mathbf{T}_2 &= \mathbf{R}_1\mathbf{Q}_1 \\ &\dots & &\dots \\ \mathbf{T}_k &= \mathbf{Q}_k\mathbf{R}_k & \mathbf{T}_{k+1} &= \mathbf{R}_k\mathbf{Q}_k \end{aligned} \tag{9.19}$$

Using the last relation in Equation 9.19, we find

$$\mathbf{R}_k = \mathbf{Q}_k^{-1}\mathbf{T}_k \Rightarrow \mathbf{T}_{k+1} = \mathbf{Q}_k^{-1}\mathbf{T}_k\mathbf{Q}_k$$

so that \mathbf{T}_{k+1} and \mathbf{T}_k are similar matrices, and thus have the same eigenvalues. In fact, it is easy to see that \mathbf{T}_{k+1} is similar to the original tridiagonal matrix \mathbf{T} . If the eigenvalues of \mathbf{T} have the property that $|\lambda_1| > |\lambda_2| > \dots > |\lambda_n|$, it can then be shown that

$$\mathbf{T}_k \rightarrow \Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \dots & \\ & & & \lambda_n \end{bmatrix} \quad \text{as } k \rightarrow \infty$$

Therefore, theoretically speaking, the sequence generated by Equation 9.19 converges to a diagonal matrix consisting of the eigenvalues of \mathbf{T} . The primary challenge is the construction of matrices \mathbf{Q}_k and \mathbf{R}_k in each iteration step of Equation 9.19.

9.8.4 Determination of \mathbf{Q}_k and \mathbf{R}_k Matrices

We begin with the first relation in Equation 9.19, and determine \mathbf{Q}_0 and \mathbf{R}_0 for $\mathbf{T}_0 = [t_{ij}]$. Pre-multiply \mathbf{T}_0 by an $n \times n$ matrix \mathbf{L}_2 , the result denoted by $\mathbf{L}_2\mathbf{T}_0 = [t_{ij}^{(2)}]$, such that $t_{21}^{(2)} = 0$. Then, pre-multiply this matrix by \mathbf{L}_3 , denoting the product by $\mathbf{L}_3(\mathbf{L}_2\mathbf{T}_0) = [t_{ij}^{(3)}]$, so that $t_{32}^{(3)} = 0$. Performing $n - 1$ of these operations yields an upper-triangular matrix \mathbf{R}_0 , that is,

$$\mathbf{L}_n\mathbf{L}_{n-1}\dots\mathbf{L}_3\mathbf{L}_2\mathbf{T}_0 = \mathbf{R}_0 \quad (9.20)$$

We will see later that \mathbf{L}_k ($k = 2, 3, \dots, n$) are orthogonal. Manipulation of Equation 9.20 results in QR factorization of \mathbf{T}_0 ,

$$\mathbf{T}_0 = (\mathbf{L}_n\mathbf{L}_{n-1}\dots\mathbf{L}_3\mathbf{L}_2)^{-1}\mathbf{R}_0 \Rightarrow \mathbf{T}_0 = \mathbf{Q}_0\mathbf{R}_0 \quad (9.21)$$

where

$$\mathbf{Q}_0 = (\mathbf{L}_n\mathbf{L}_{n-1}\dots\mathbf{L}_3\mathbf{L}_2)^{-1} = \mathbf{L}_2^{-1}\mathbf{L}_3^{-1}\dots\mathbf{L}_n^{-1} = \mathbf{L}_2^T\mathbf{L}_3^T\dots\mathbf{L}_n^T$$

Note that the orthogonality of \mathbf{L}_k has been utilized, that is, $\mathbf{L}_k^{-1} = \mathbf{L}_k^T$.

9.8.5 Structure of \mathbf{L}_k ($k = 2, 3, \dots, n$)

The \mathbf{L}_k matrices are generally simple in nature in the sense that each \mathbf{L}_k consists of a 2×2 submatrix that occupies rows k and $k-1$, and columns k and $k-1$, and ones along the remaining portion of the main diagonal, and zeros everywhere else. The 2×2 submatrix has the form of a clockwise rotation matrix

$$\begin{bmatrix} \cos\theta_k & \sin\theta_k \\ -\sin\theta_k & \cos\theta_k \end{bmatrix} \quad \text{or simply} \quad \begin{bmatrix} c_k & s_k \\ -s_k & c_k \end{bmatrix} \quad \text{where} \quad \begin{aligned} c_k &= \cos\theta_k \\ s_k &= \sin\theta_k \end{aligned}$$

and θ_k is to be chosen appropriately. For instance, if the size of the matrices involved is $n = 5$, then

$$\mathbf{L}_2 = \begin{bmatrix} c_2 & s_2 & & & \\ -s_2 & c_2 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix}, \quad \mathbf{L}_4 = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & c_4 & s_4 & \\ & & -s_4 & c_4 & \\ & & & & 1 \end{bmatrix}$$

We now address the selection of c_k and s_k . Recall that we must have $\mathbf{L}_2\mathbf{T}_0 = [t_{ij}^{(2)}]$ such that $t_{21}^{(2)} = 0$, hence in this new matrix the only element that needs to be analyzed is the (2,1) entry. But we determine the (2,1) entry by using the second row of \mathbf{L}_2 and first column of \mathbf{T}_0 . Regardless of the size n , the second row of matrix \mathbf{L}_2 is always as shown above. Therefore, the (2,1) entry of $\mathbf{L}_2\mathbf{T}_0$ is given by

$$t_{21}^{(2)} = -s_2t_{11} + c_2t_{21}$$

Forcing it to be zero, we find

$$-s_2t_{11} + c_2t_{21} = 0 \Rightarrow \frac{s_2}{c_2} = \tan\theta_2 = \frac{t_{21}}{t_{11}} \quad (9.22)$$

Application of trigonometric identities $\cos\alpha = \frac{1}{\sqrt{1+\tan^2\alpha}}$ and $\sin\alpha = \frac{\tan\alpha}{\sqrt{1+\tan^2\alpha}}$ to Equation 9.22 yields

$$c_2 = \cos\theta_2 = \frac{1}{\sqrt{1+(t_{21}/t_{11})^2}}, \quad s_2 = \sin\theta_2 = \frac{t_{21}/t_{11}}{\sqrt{1+(t_{21}/t_{11})^2}} \quad (9.23)$$

With this, matrix \mathbf{L}_2 is completely defined. Next, consider the matrix $\mathbf{L}_3(\mathbf{L}_2\mathbf{T}_0) = [t_{ij}^{(3)}]$, of which the (3,2) entry must be made into zero. Proceeding as above, we can obtain c_3 and s_3 , and so on. This continues until \mathbf{L}_n is determined, and ultimately, $\mathbf{L}_n\mathbf{L}_{n-1} \dots \mathbf{L}_3\mathbf{L}_2\mathbf{T}_0 = \mathbf{R}_0$. Once all \mathbf{L}_k matrices have been found, we form $\mathbf{Q}_0 = \mathbf{L}_2^T\mathbf{L}_3^T \dots \mathbf{L}_n^T$, by Equation 9.21, and the first QR factorization is complete. Next, form the new symmetric tridiagonal matrix $\mathbf{T}_1 = \mathbf{R}_0\mathbf{Q}_0$. If the off-diagonal elements are much smaller than the diagonal ones, the process is terminated and the diagonal entries of \mathbf{T}_1 approximate the eigenvalues of \mathbf{A} . Otherwise, the QR factorization is repeated for \mathbf{T}_1 via the steps listed above until a desired tridiagonal matrix is achieved.

9.9 MATLAB Built-In Function `qr`

MATLAB has a built-in function that performs the QR factorization of a matrix:

`qr` Orthogonal-triangular decomposition.

`[Q,R] = qr(A)`, where A is m -by- n , produces an m -by- n upper triangular matrix R and an m -by- m unitary matrix Q so that $A = Q^*R$.

The user-defined function `HouseholderQR` uses Householder's method to transform a real, symmetric matrix into a tridiagonal matrix, to which the QR factorization is repeatedly applied in order to obtain a tridiagonal matrix whose diagonal entries are much larger in magnitude than those along the upper- and lower-diagonals.

```
function [T, Tfinal, e_vals, m] = HouseholderQR(A, tol, kmax)
%
% HouseholderQR uses Householder's method and repeated applications of
% QR factorization to estimate the eigenvalues of a real, symmetric matrix.
%
% [T, Tfinal, e_vals, m] = HouseholderQR(A, tol, kmax), where
%
%     A is an n-by-n real, symmetric matrix,
%     tol is the tolerance used in the QR process (default 1e-4),
%     kmax is the maximum number of QR iterations (default 50),
%
%     T is the tridiagonal matrix created by Householder's method,
%     Tfinal is the final tridiagonal matrix,
%     e_vals is a list of estimated eigenvalues of matrix A,
%     m is the number of iterations needed for convergence.
%
% Note that this function calls the user-defined function Householder!
%
if nargin < 2 || isempty(tol), tol = 1e-4; end
if nargin < 3 || isempty(kmax), kmax = 50; end
T = Householder(A); % Call Householder
T(:, :, 1) = T;
% QR factorization to reduce the off-diagonal entries of T
for m = 1:kmax,
    [Q,R] = qr(T(:, :, m));
    T(:, :, m+1) = R*Q;
    % Compare diagonals of two successive T matrices
    if norm(diag(T(:, :, m+1))-diag(T(:, :, m))) < tol,
        break;
    end
end
Tfinal = T(:, :, end); T = T(:, :, 1); e_vals = diag(Tfinal);
```

9.10 A Note on the Terminating Condition Used in `HouseholderQR`

The terminating condition employed here is based on the norm of the difference between two vectors whose components are the diagonal entries of two successive **T** matrices generated in the QR process. Although this works in most cases, there could be situations where the two aforementioned vectors have elements that are close to one another, hence meeting the tolerance condition, but the off-diagonal entries are not sufficiently small. One way to remedy this is to use a more firm terminating condition that inspects the ratio of the largest magnitude lower-diagonal entry to the smallest diagonal element. If this ratio is within the tolerance, the process is terminated; see Problem Set. This will ensure that the

lower-diagonal elements have considerably smaller magnitudes than the diagonal ones. Note that since symmetric matrices are involved, inspection of the lower-diagonal entries will suffice.

EXAMPLE 9.7: HOUSEHOLDER'S METHOD + QR FACTORIZATION

Consider Example 9.6 where a symmetric matrix was transformed into tridiagonal using Householder's method:

$$\mathbf{A} = \begin{bmatrix} 4 & 4 & 1 & 1 \\ 4 & 4 & 1 & 1 \\ 1 & 1 & 3 & 2 \\ 1 & 1 & 2 & 3 \end{bmatrix} \rightarrow \mathbf{T} = \begin{bmatrix} 4 & -4.2426 & 0 & 0 \\ -4.2426 & 5 & 1.4142 & 0 \\ 0 & 1.4142 & 4 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Interestingly, \mathbf{T} is in the block diagonal form (Chapter 1). Therefore, eigenvalues of \mathbf{T} consist of the eigenvalues of the upper-left 3×3 block matrix, and a 1×1 block of 1. So, one eigenvalue ($\lambda_1 = 1$) is automatically decided. As a result, we now focus on the upper-left 3×3 block and find its eigenvalues. Note that this phenomenon does not generally occur and should be investigated on a case-by-case basis.

Therefore, we will proceed with the aforementioned 3×3 block matrix and repeatedly apply the QR factorization to this matrix. The process is initiated by setting

$$\mathbf{T}_0 = \begin{bmatrix} 4 & -4.2426 & 0 \\ -4.2426 & 5 & 1.4142 \\ 0 & 1.4142 & 4 \end{bmatrix}$$

The QR factorizations listed in Equation 9.19 are performed as follows. First, by Equation 9.23,

$$c_2 = 0.6860, \quad s_2 = -0.7276$$

so that

$$\mathbf{L}_2 = \begin{bmatrix} c_2 & s_2 & 0 \\ -s_2 & c_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{L}_2 \mathbf{T}_0 = \begin{bmatrix} 5.8310 & -6.5485 & -1.0290 \\ 0 & 0.3430 & 0.9701 \\ 0 & 1.4142 & 4 \end{bmatrix}$$

Next, the (3,2) entry of $\mathbf{L}_3 \mathbf{L}_2 \mathbf{T}_0$ is forced to be zero, and yields

$$-0.3430s_3 + 1.4142c_3 = 0 \Rightarrow \begin{aligned} c_3 &= 0.2357 \\ s_3 &= 0.9718 \end{aligned}$$

so that

$$\mathbf{L}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_3 & s_3 \\ 0 & -s_3 & c_3 \end{bmatrix} \quad \text{and} \quad \mathbf{R}_0 = \mathbf{L}_3 \mathbf{L}_2 \mathbf{T}_0 = \begin{bmatrix} 5.8310 & -6.5485 & -1.0290 \\ 0 & 1.4552 & 4.1160 \\ 0 & 0 & 0 \end{bmatrix}$$

Finally, letting $\mathbf{Q}_0 = \mathbf{L}_2^T \mathbf{L}_3^T$, we obtain

$$\mathbf{T}_1 = \mathbf{R}_0 \mathbf{Q}_0 = \begin{bmatrix} 8.7647 & -1.0588 & 0 \\ -1.0588 & 4.2353 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

This tridiagonal matrix is also in a block diagonal form, including an upper-left 2×2 block and a 1×1 block of 0. This implies one of its eigenvalues must be zero, hence $\lambda_2 = 0$. The remaining two eigenvalues of \mathbf{A} are the eigenvalues of the upper-left 2×2 block matrix. We will proceed with the second iteration in Equation 9.19 using this 2×2 block matrix. So, we let

$$\mathbf{T}_1 = \begin{bmatrix} 8.7647 & -1.0588 \\ -1.0588 & 4.2353 \end{bmatrix}$$

Since the off-diagonal entries are still relatively large, matrix \mathbf{T}_2 must be formed, that is,

$$\mathbf{L}_2^{(1)} = \begin{bmatrix} c_2^{(1)} & s_2^{(1)} \\ -s_2^{(1)} & c_2^{(1)} \end{bmatrix} \text{ with } c_2^{(1)} = 0.9928 \text{ so that } \mathbf{R}_1 = \mathbf{L}_2^{(1)} \mathbf{T}_1 = \begin{bmatrix} 8.8284 & -1.5591 \\ 0 & 4.0777 \end{bmatrix}$$

Noting that $\mathbf{Q}_1 = [\mathbf{L}_2^{(1)}]^T$, we have

$$\mathbf{T}_2 = \mathbf{R}_1 \mathbf{Q}_1 = \begin{bmatrix} 8.9517 & -0.4891 \\ -0.4891 & 4.0483 \end{bmatrix}$$

Finally, because the off-diagonal entries are considerably smaller in magnitude than the diagonal ones, the eigenvalues are approximately 9 and 4. Therefore, $\lambda(\mathbf{A}) = 0, 1, 4, 9$. Executing the user-defined function `HouseholderQR` will confirm these results.

```
>> A = [4 4 1 1; 4 4 1 1; 1 1 3 2; 1 1 2 3];
>> [T, Tfinal, e_vals, m] = HouseholderQR(A)

T =          % Tridiagonal matrix generated by Householder (see Example 9.6)
    4.0000   -4.2426    0.0000    0.0000
   -4.2426    5.0000    1.4142         0
   -0.0000    1.4142    4.0000    0.0000
    0.0000    0.0000    0.0000    1.0000

Tfinal =          % Tridiagonal matrix at the conclusion of QR process
    9.0000   -0.0086   -0.0000   -0.0000
   -0.0086    4.0000   -0.0000    0.0000
   -0.0000    0.0000    1.0000   -0.0000
   -0.0000   -0.0000    0.0000    0.0000

e_vals =          % List of (estimated) eigenvalues of A
    9.0000
    4.0000
    1.0000
    0.0000

m =
    7
```

9.11 Transformation to Hessenberg Form (Nonsymmetric Matrices)

As mentioned at the outset of this section, Householder's method can also be applied to nonsymmetric matrices. Instead of a tridiagonal matrix, however, the outcome will be another special matrix known as the upper Hessenberg form

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1n} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2n} \\ & h_{32} & h_{33} & \dots & h_{3n} \\ & & \dots & \dots & \dots \\ & & & h_{n,n-1} & h_{nn} \end{bmatrix}$$

which is an upper triangular plus the lower-diagonal. In the second step of the process, repeated QR factorizations will be applied to \mathbf{H} in order to obtain an upper triangular matrix. Since the eigenvalues of an upper triangular matrix are its diagonal entries, the eigenvalues of this final matrix, and hence of the original matrix, are along its diagonal.

The user-defined function `HouseholderQR` can be applied to any nonsymmetric matrix to accomplish this task.

EXAMPLE 9.8: NONSYMMETRIC MATRIX, HESSENBERG FORM

Consider the (nonsymmetric) matrix studied in Examples 9.3 and 9.4 of the last section:

$$\mathbf{A} = \begin{bmatrix} 4 & -3 & 3 & -9 \\ -3 & 6 & -3 & 11 \\ 0 & 8 & -5 & 8 \\ 3 & -3 & 3 & -8 \end{bmatrix}$$

```
>> A = [4 -3 3 -9;-3 6 -3 11;0 8 -5 8;3 -3 3 -8];
>> [H, Hfinal, e_vals, m] = HouseholderQR(A);
```

This returns `m=50`, which means the (default) maximum number of iterations has been exhausted. Therefore, we will increase `kmax` to 60 and reexecute the function.

```
>> [H, Hfinal, e_vals, m] = HouseholderQR(A, [], 60)
```

```
H = % Hessenberg form generated by Householder
```

```
4.0000 -4.2426 3.8787 -8.1213
4.2426 -5.0000 6.8995 -12.8995
0.0000 -0.0000 4.6569 -9.6569
-0.0000 -0.0000 1.6569 -6.6569
```

```
Hfinal = % Upper triangular matrix at the conclusion of QR process
```

```
-5.0000 5.1961 13.4722 16.2634
-0.0000 -2.0000 -3.5355 2.8578
-0.0000 0.0000 3.0000 1.1547
-0.0000 0.0000 -0.0000 1.0000
```

```
e_vals =           % List of (estimated) eigenvalues of A
-5.0000
-2.0000
 3.0000
 1.0000

m =
 53
```

It took 53 iterations for convergence to be observed.

PROBLEM SET (CHAPTER 9)

Power Methods (Sections 9.2 through 9.5)

In Problems 1 through 7,

- Starting with $\alpha_1 = 0$ and an $n \times 1$ initial vector \mathbf{x}_1 comprised of all ones, apply the power method to generate the scalars α_2 and α_3 , and the normalized vectors \mathbf{x}_2 and \mathbf{x}_3 .
- Find the dominant eigenvalue and the corresponding eigenvector of \mathbf{A} by executing the user-defined function `PowerMethod` with default input arguments.

$$1. \mathbf{A} = \begin{bmatrix} 4 & -1 & -5 \\ -2 & 5 & 10 \\ 4 & 2 & 1 \end{bmatrix}$$

$$2. \mathbf{A} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$3. \mathbf{A} = \begin{bmatrix} -3 & 6 & -9 \\ -4 & 9 & -14 \\ -4 & 6 & -8 \end{bmatrix}$$

$$4. \mathbf{A} = \begin{bmatrix} 3 & 1 & -2 \\ -6 & 2 & -4 \\ -3 & -1 & 2 \end{bmatrix}$$

$$5. \mathbf{A} = \begin{bmatrix} 0 & -1 & -5 \\ -2 & 1 & 10 \\ 4 & 2 & -3 \end{bmatrix}$$

$$6. \mathbf{A} = \begin{bmatrix} 3 & 2 & -2 & 0 \\ -1 & -2 & 2 & -2 \\ 0 & -3 & 3 & -3 \\ 1 & 2 & -2 & 2 \end{bmatrix}$$

$$7. \mathbf{A} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 2 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 1 & 1 & 2 \end{bmatrix}$$

▲ In Problems 8 through 11 find all eigenvalues and eigenvectors of the matrix by applying the user-defined functions `PowerMethod` and/or `ShiftInvPower`. Use the given initial vector \mathbf{x}_1 in each application of these two functions, and default values for the remaining input arguments.

$$8. \mathbf{A} = \begin{bmatrix} 18 & -23 & -11 \\ -14 & 7 & 7 \\ 64 & -60 & -36 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{Bmatrix} 1 \\ 0 \\ 1 \end{Bmatrix}$$

$$9. \mathbf{A} = \begin{bmatrix} -10 & -6 & -2 \\ -8 & -2 & -3 \\ 84 & 42 & 21 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{Bmatrix} 0 \\ 1 \\ 1 \end{Bmatrix}$$

$$10. \mathbf{A} = \begin{bmatrix} 15 & 12 & 4 \\ -48 & -33 & -10 \\ 24 & 12 & 1 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{Bmatrix} 1 \\ 0 \\ 1 \end{Bmatrix}$$

$$11. \mathbf{A} = \begin{bmatrix} -6 & 8 & 0 & -8 \\ -2 & 8 & -2 & -2 \\ 2 & -2 & 8 & 2 \\ 14 & -10 & -2 & 16 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{Bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{Bmatrix}$$

12. ▲ Using any combination of the power methods find all eigenvalues and eigenvectors of

$$\mathbf{A} = \begin{bmatrix} -2 & -3 & 0 \\ 0 & 1 & 0 \\ 4 & 3 & 2 \end{bmatrix}$$

Deflation Methods (Section 9.7)

▲ In Problems 13 through 18, for each matrix \mathbf{A} , find the dominant eigenvalue and corresponding eigenvector by executing the user-defined function `PowerMethod` with default input arguments. Then, using Wielandt's deflation method, generate a matrix \mathbf{B} , one size smaller than \mathbf{A} , whose eigenvalues are the remaining eigenvalues of \mathbf{A} . Finally, apply the `eig` function to find the eigenvalues of \mathbf{B} . List all eigenvalues of \mathbf{A} .

$$13. \mathbf{A} = \begin{bmatrix} 2 & -1 & 1 \\ -1 & 1 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

$$14. \mathbf{A} = \begin{bmatrix} -16 & 31 & 19 \\ -22 & 31 & 22 \\ 8 & -2 & -5 \end{bmatrix}$$

$$15. \mathbf{A} = \begin{bmatrix} 6 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 6 \end{bmatrix}$$

$$16. \mathbf{A} = \begin{bmatrix} 0 & 1 & -2 \\ -6 & -1 & -4 \\ -3 & -1 & -1 \end{bmatrix}$$

$$17. \mathbf{A} = \begin{bmatrix} 9 & 0 & 0 \\ 5 & 7 & -5 \\ 5 & -2 & 4 \end{bmatrix}$$

$$18. \mathbf{A} = \begin{bmatrix} 3 & -6 & -7 \\ 0 & 4 & 1 \\ 0 & 6 & 9 \end{bmatrix}$$

19. Write a user-defined function with function call

```
e_vals = Wielandt_Deflation(A, x1, tol, kmax)
```

that uses Wielandt's deflation method to find all the eigenvalues of a matrix \mathbf{A} . It must call the user-defined function `PowerMethod` to estimate the dominant eigenvalue and corresponding eigenvector of \mathbf{A} , deflate to a smaller size matrix, and repeat the process until it reaches a 2×2 matrix. At that point, the MATLAB function `eig` must be used to find the eigenvalues of the 2×2 matrix. The function must return a list of all eigenvalues of matrix \mathbf{A} . The input arguments have the same default values as in `PowerMethod`. Apply `Wielandt_Deflation` (with default inputs) to the matrix \mathbf{A} in Example 9.4.

Note: A key part of the process is determining the first nonzero component of the eigenvector \mathbf{v}_1 in each step. But since `PowerMethod` generates this vector, it is only an approximation and as such, a component whose true value is zero will appear to have a very small value, but not exactly zero. Thus, your function must view any component with magnitude less than 10^{-4} as zero.

20. Find all eigenvalues of the matrix \mathbf{A} by executing the user-defined function `Wielandt_Deflation` (Problem 19).

$$\mathbf{A} = \begin{bmatrix} 5 & -3 & 0 & 3 \\ -3 & 7 & 8 & -3 \\ 3 & -3 & -4 & 3 \\ -9 & 11 & 8 & -7 \end{bmatrix}$$

21. Find all eigenvalues of the matrix \mathbf{A} by executing the user-defined function `Wielandt_Deflation` (Problem 19).

$$\mathbf{A} = \begin{bmatrix} -1 & 3 & -3 & 5 \\ 1 & -1 & 3 & -3 \\ 0 & -2 & 4 & -2 \\ -1 & 3 & -3 & 5 \end{bmatrix}$$

Householder Tridiagonalization and QR Factorization Methods (Section 9.8)

In Problems 22 through 24,

- Use Householder's method to transform the given matrix into tridiagonal or Hessenberg form.
- Confirm the findings in (a) by executing the user-defined function `Householder`.

22. $\mathbf{A} = \begin{bmatrix} 7 & 4 & 1 & 3 \\ 4 & 7 & 3 & 1 \\ 1 & 3 & 7 & 4 \\ 3 & 1 & 4 & 7 \end{bmatrix}$

23. $\mathbf{A} = \begin{bmatrix} 5 & -1 & 0 & 1 \\ -1 & 8 & 2 & -3 \\ 0 & 2 & -1 & 1 \\ 1 & -3 & 1 & 6 \end{bmatrix}$

24. $\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 & -2 \\ 4 & 4 & 0 & 3 \\ -5 & -5 & 0 & -2 \\ 0 & 0 & 0 & 2 \end{bmatrix}$

- ▲ In Problems 25 through 29, find all eigenvalues of the matrix by executing the user-defined function `HouseholderQR`, and verify the results using MATLAB function `eig`.

25. $\mathbf{A} = \begin{bmatrix} 5 & 4 & 1 & 3 \\ 4 & 5 & 3 & 1 \\ 1 & 3 & 5 & 4 \\ 3 & 1 & 4 & 5 \end{bmatrix}$

26. $\mathbf{A} = \begin{bmatrix} 2 & -1 & 2 & 0 & 5 \\ -1 & 4 & 0 & 0 & -3 \\ 2 & 0 & 5 & -1 & 0 \\ 0 & 0 & -1 & 1 & 2 \\ 5 & -3 & 0 & 2 & 2 \end{bmatrix}$

$$27. \mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & -2 \\ 4 & 3 & 0 & 3 \\ -5 & -5 & -1 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$28. \mathbf{A} = \begin{bmatrix} 0 & -1 & 0 & 1 \\ -1 & 3 & 2 & -3 \\ 0 & 2 & -6 & 1 \\ 1 & -3 & 1 & 1 \end{bmatrix}$$

$$29. \mathbf{A} = \begin{bmatrix} 1 & 2 & 0 & 3 & 1 \\ 2 & 4 & -2 & 0 & 4 \\ 0 & -2 & 5 & -1 & 1 \\ 3 & 0 & -1 & 8 & -3 \\ 1 & 4 & 1 & -3 & 2 \end{bmatrix}$$

30.  Write a user-defined function with function call

```
[T, Tfinal, e_vals, m] = HouseholderQR_New(A, tol, kmax)
```

that uses Householder's method and the successive QR factorization process to find all eigenvalues of a matrix, as follows. Modify `HouseholderQR` by altering the terminating condition: the iterations must stop when the ratio of the largest magnitude lower-diagonal entry to the smallest magnitude diagonal element is less than the prescribed tolerance. All input parameters have the same default as before. Apply this function to the matrix in Problem 29.

Numerical Solution of Partial Differential Equations

Partial differential equations play an important role in several areas of engineering ranging from fluid mechanics, heat transfer, and applied mechanics to electromagnetic theory. Since it is generally much more difficult to find a closed-form solution for partial differential equations than it is for ordinary differential equations, they are usually solved numerically. In this chapter, we present numerical methods for solution of partial differential equations, in particular, those that describe some fundamental problems in engineering applications, including Laplace's equation, the heat equation, and the wave equation.

10.1 Introduction

A partial differential equation (PDE) is an equation involving a function (dependent variable) of at least two independent variables, and its partial derivatives. A PDE is of order n if the highest derivative is of order n . If a PDE is of the first degree in the dependent variable and its partial derivatives, it is called linear. Otherwise, it is nonlinear. If each term in a PDE involves either the dependent variable or its partial derivatives, the PDE is called homogeneous. Otherwise, it is nonhomogeneous.

Suppose $u = u(x, y)$. Then, the following brief notations for partial derivatives are used:

$$u_x = \frac{\partial u}{\partial x}, \quad u_{xx} = \frac{\partial^2 u}{\partial x^2}, \quad u_{xy} = \frac{\partial^2 u}{\partial y \partial x}$$

The dimension of a PDE is determined by the number of spatial coordinates, not time t . For example, a PDE with $u = u(x, y, z)$ as its dependent variable is three-dimensional, while a PDE with dependent variable $u = u(x, t)$ is one-dimensional.

Consider a class of linear, second-order PDEs that appear in the form

$$au_{xx} + 2bu_{xy} + cu_{yy} = f(x, y, u, u_x, u_y) \quad (10.1)$$

A PDE in the form of Equation 10.1 is

- Elliptic if $b^2 - ac < 0$
- Parabolic if $b^2 - ac = 0$
- Hyperbolic if $b^2 - ac > 0$

The two-dimensional Poisson's equation $u_{xx} + u_{yy} = f(x, y)$ and the two-dimensional Laplace's equation $u_{xx} + u_{yy} = 0$ are elliptic PDEs; the one-dimensional heat equation $u_t = \alpha^2 u_{xx}$ ($\alpha = \text{const} > 0$) is parabolic; and the one-dimensional wave equation $u_{tt} = \alpha^2 u_{xx}$ ($\alpha = \text{const} > 0$) is hyperbolic. In our examples for parabolic and hyperbolic PDEs, the variable t has replaced y . In applications, when an elliptic PDE is involved, a boundary-value problem needs to be solved. Those involving parabolic and hyperbolic types require the solution of a boundary-initial-value problem. We will present the numerical solution of elliptic PDEs first.

10.2 Elliptic Partial Differential Equations

A Dirichlet problem refers to a problem that involves solving an elliptic equation in a specific region in the xy -plane, where the unknown function is prescribed along the boundary of the region. On the other hand, a Neumann problem refers to a boundary-value problem where the normal derivative of u , that is, $u_n = \partial u / \partial n$, is given on the boundary of the region. Note that along a vertical edge of a region, u_n is simply $u_x = \partial u / \partial x$, and along a horizontal edge it is $u_y = \partial u / \partial y$. The mixed problem refers to the situation where u is specified on certain parts of the boundary, and u_n on the others.

10.2.1 Dirichlet Problem

As a fundamental Dirichlet problem, we consider the solution of the 2D Poisson's equation

$$u_{xx} + u_{yy} = f(x, y)$$

in the rectangular region shown in [Figure 10.1](#), where $u(x, y)$ is prescribed along the boundary. The idea is to define a mesh size h and construct a grid by drawing equidistant vertical and horizontal lines of distance h . These lines are called grid lines, and the points at which they intersect are known as mesh points. Those mesh points that are located on the boundary are called boundary points. Mesh points that lie inside the region are called interior mesh points. The goal is to approximate the solution u at the interior mesh points.

Let us denote the typical mesh point (x, y) by (ih, jh) , simply labeled as (i, j) in [Figure 10.1](#). The value of u at that point is then denoted by u_{ij} . Similarly, the value of f at that point is represented by f_{ij} . Approximating the second-order partial derivatives in Poisson's equation with three-point central difference formulas ([Chapter 6](#)), we find

$$\frac{u_{i-1,j} - 2u_{ij} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{ij} + u_{i,j+1}}{h^2} = f_{ij}$$

which simplifies to

$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij} = h^2 f_{ij} \tag{10.2}$$

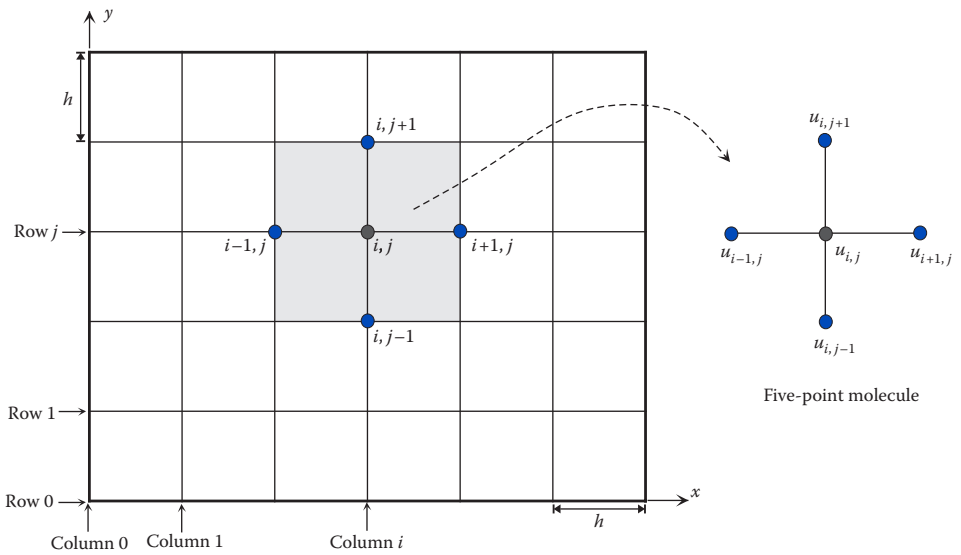


FIGURE 10.1

A grid for a rectangular region, and a five-point molecule.

This is called the difference equation for Poisson's equation, which provides a relation between the solution u at (i, j) and four adjacent points. Likewise, for Laplace's equation, $u_{xx} + u_{yy} = 0$, we have

$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij} = 0 \quad (10.3)$$

which is known as the difference equation for Laplace's equation.

The Dirichlet problem is then solved as follows. The difference equation—Equation 10.2 for Poisson's equation, Equation 10.3 for Laplace's equation—is applied at every interior mesh point. This will result in a linear system of equations whose size is the same as the total number of interior mesh points. Assuming there are n interior mesh points in the region, this linear system is in the form $\mathbf{A}\mathbf{u} = \mathbf{b}$, where $\mathbf{A}_{n \times n}$ is the coefficient matrix, $\mathbf{u}_{n \times 1}$ is the vector of the unknowns, and $\mathbf{b}_{n \times 1}$ is comprised of known quantities. Note that, due to the nature of the difference equation, matrix \mathbf{A} has at most five nonzero entries in each row. When at least one of the adjacent points in the five-point molecule (Figure 10.1) is a boundary point, the value of u is available by the boundary condition and is ultimately moved to the right side of the equation, hence becoming part of vector \mathbf{b} . In addition to the boundary points, in the case of Poisson's equation, the terms $h^2 f_{ij}$ will also be included in \mathbf{b} . In practice, a large number of mesh points are needed for better accuracy, causing the coefficient matrix \mathbf{A} to be large and sparse. A linear system $\mathbf{A}\mathbf{u} = \mathbf{b}$ with a large, sparse coefficient matrix is normally solved numerically via an iterative method such as the Gauss–Seidel method (Chapter 4).

The user-defined function `DirichletPDE` uses the difference-equation approach outlined above to numerically solve Poisson's equation—or Laplace's equation—in a rectangular region with the values of the unknown solution available on the boundary. The function returns the approximate solution at the interior mesh points, as well as the

values at the boundary points in a pattern that resembles the gridded region. It also returns the three-dimensional plot of the results.

```
function U = DirichletPDE(x,y,f,uleft,uright,ubottom,utop)
%
% DirichletPDE numerically solves an elliptic PDE with Dirichlet boundary
% conditions over a rectangular region.
%
% U = DirichletPDE(x,y,f,uleft,uright,ubottom,utop), where
%
% x is the 1-by-m vector of mesh points in the x direction,
% y is the 1-by-n vector of mesh points in the y direction,
% f is an anonymous function representing f(x,y),
% ubottom(x),utop(x),uright(y),uleft(y) are anonymous functions
% describing the boundary conditions,
%
% U is the solution at the interior mesh points.
%
y = y'; m = size(x,2); n = size(y,1); N = (m-2)*(n-2);
A = diag(-4*ones(N,1));
A = A + diag(diag(A,n-2)+1,n-2);
A = A + diag(diag(A,2-n)+1,2-n);
v = ones(N-1,1); % Create vector of ones
v(n-2:n-2:end) = 0; % Insert zeros
A = A + diag(v,1); % Add upper diagonal
A = A + diag(v,-1); % Add lower diagonal
[X,Y] = meshgrid(x(2:end-1),y(end-1:-1:2)); % Create mesh
h = x(2)-x(1);
% Define boundary conditions
for i = 2:m-1,
    utop_bound(i-1) = utop(x(i));
    ubottom_bound(i-1) = ubottom(x(i));
end
for i = 1:n,
    uleft_bound(i) = uleft(y(n+1-i));
    uright_bound(i) = uright(y(n+1-i));
end
b = 0; % Initialize vector b
for i = 1:N,
    b(i) = h^2*f(X(i),Y(i));
end
b(1:n-2:N) = b(1:n-2:N)-utop_bound;
b(n-2:n-2:N) = b(n-2:n-2:N)-ubottom_bound;
b(1:n-2) = b(1:n-2)-uleft_bound(2:n-1);
b(N-(n-3):N) = b(N-(n-3):N)-uright_bound(2:n-1);
u = A\b'; % Solve the system
U = reshape(u,n-2,m-2);
U = [utop_bound;U;ubottom_bound];
U = [uleft_bound' U uright_bound'];
[X,Y] = meshgrid(x,y(end:-1:1));
surf(X,Y,U); % 3D plot of the numerical results
xlabel('x');ylabel('y');
```

EXAMPLE 10.1: DIRICHLET PROBLEM

The Dirichlet problem in Figure 10.2 describes the steady-state temperature distribution inside a rectangular plate of length 2 and width 1. Three of the sides are kept at zero temperature, while the lower edge has a temperature profile of $\sin(\pi x/2)$. Using a mesh size of $h = 0.5$ construct a grid and find the approximate values of u at the interior mesh points, and calculate the relative errors associated with these approximate values. The exact solution in closed form is given by

$$u(x, y) = \frac{1}{\sinh(\pi/2)} \sin \frac{\pi x}{2} \sinh \frac{\pi(1-y)}{2}$$

Solution

There are three interior mesh points and eight boundary points on the grid. Therefore, the difference equation, Equation 10.3, must be applied three times, once at each interior mesh point. As a result, we have

$$\begin{aligned} (i = 1, j = 1) \quad & u_{01} + u_{10} + u_{21} + u_{12} - 4u_{11} = 0 \\ (i = 2, j = 1) \quad & u_{11} + u_{20} + u_{31} + u_{22} - 4u_{21} = 0 \\ (i = 3, j = 1) \quad & u_{21} + u_{30} + u_{41} + u_{32} - 4u_{31} = 0 \end{aligned}$$

Included in these equations are the values at the boundary points,

$$u_{12} = u_{22} = u_{32} = u_{01} = u_{41} = 0, \quad u_{10} = 0.7071 = u_{30}, \quad u_{20} = 1$$

Inserting these into the system of equations, we find

$$\begin{aligned} -4u_{11} + u_{21} &= -0.7071 \\ u_{11} - 4u_{21} + u_{31} &= -1 \\ u_{21} - 4u_{31} &= -0.7071 \end{aligned} \quad \begin{array}{l} \text{In matrix form} \\ \Rightarrow \end{array} \quad \begin{bmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & -4 \end{bmatrix} \begin{Bmatrix} u_{11} \\ u_{21} \\ u_{31} \end{Bmatrix} = \begin{Bmatrix} -0.7071 \\ -1 \\ -0.7071 \end{Bmatrix}$$

Solution of this system yields $u_{11} = 0.2735 = u_{31}$ and $u_{21} = 0.3867$. The exact values at these points are calculated as $u_{11} = 0.2669 = u_{31}$ and $u_{21} = 0.3775$, recording relative errors of 2.47% and 2.44%, respectively. The estimates turned out reasonably accurate considering the large mesh size that was used. Switching to a smaller mesh size of $h = 0.25$, for example, generates a grid that includes 21 interior mesh points and 20 boundary points. The ensuing linear system then comprises 21 equations and 21 unknowns, whose solutions are more accurate than those obtained here.

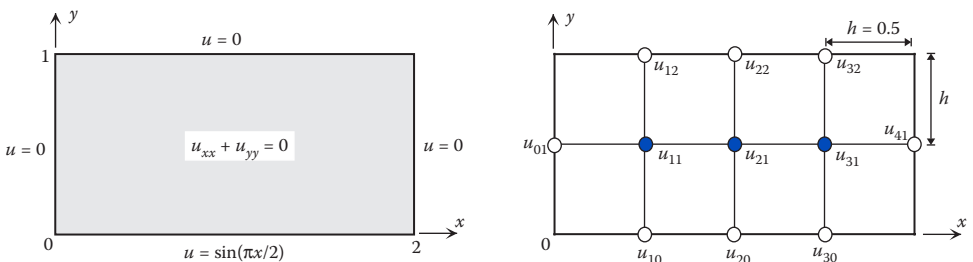


FIGURE 10.2

The Dirichlet problem in Example 10.1.

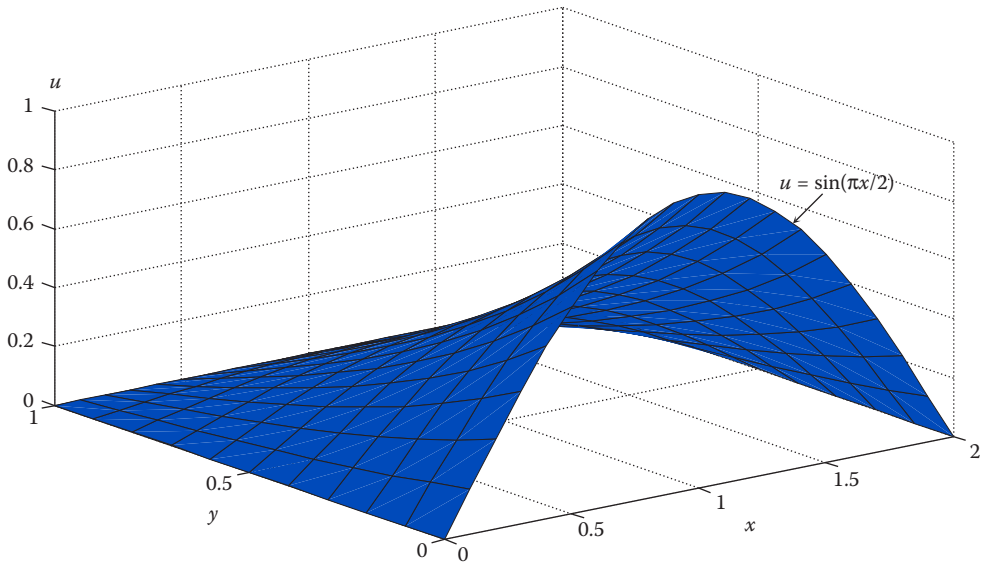


FIGURE 10.3

Steady-state temperature distribution in Example 10.1, using $h=0.1$.

To confirm the numerical results in MATLAB, we execute the user-defined function `DirichletPDE`.

```
>> x = 0:0.5:2; y = 0:0.5:1;
>> f = @(x,y)(0); ubottom = @(x)(sin(pi*x/2)); utop = @(x)(0);
>> uleft = @(y)(0); uright = @(y)(0);
>> U = DirichletPDE(x,y,f,uleft,uright,ubottom,utop) % 3D plot suppressed
U =
    0         0         0         0         0
    0    [0.2735  0.3867  0.2735]    0
    0    0.7071  1.0000  0.7071    0
```

The three values in the box are the solution estimates at the three interior mesh points, and agree with those obtained earlier. All other values correspond to the boundary points in the grid used. In order to generate a smooth plot, we reduce the mesh size to $h=0.1$ and reexecute the function. The result is shown in [Figure 10.3](#).

```
>> x = 0:0.1:2; y = 0:0.1:1;
>> U = DirichletPDE(x,y,f,uleft,uright,ubottom,utop) % Numerical results suppressed
```

10.2.2 Alternating Direction Implicit (ADI) Methods

In the foregoing analysis, we learned that the application of the difference equation for either Poisson’s equation or Laplace’s equation at the interior mesh points leads to a linear system of equations whose coefficient matrix has at most five nonzero entries in each row. Although this type of a matrix is clearly preferred to a general one, the numerical computations would be performed much more efficiently if the coefficient matrix were tridiagonal ([Chapter 4](#)), that is, if it has at most three nonzero entries in each row.

The goal is therefore to develop a scheme that leads to a system of equations with a tridiagonal coefficient matrix. We will present the idea while focusing on Laplace's equation in a rectangular region. Suppose a mesh size h generates N internal mesh points per row and M internal mesh points per column. Recall that Equation 10.3 takes into account the five elements of the five-point molecule (Figure 10.1) all at once. That is, the elements $u_{i-1,j}$, u_{ij} , $u_{i+1,j}$ along the j th row and $u_{i,j-1}$, u_{ij} , $u_{i,j+1}$ along the i th column, with u_{ij} being the common element. Since we are aiming for a tridiagonal matrix, we write Equation 10.3 as

$$u_{i-1,j} - 4u_{ij} + u_{i+1,j} = -u_{i,j-1} - u_{i,j+1} \quad (10.4)$$

so that the members on the left side belong to the j th row and those on the right to the i th column. Equation 10.3 may also be rewritten as

$$u_{i,j-1} - 4u_{ij} + u_{i,j+1} = -u_{i-1,j} - u_{i+1,j} \quad (10.5)$$

with the left side terms belonging to the i th column and the right side terms to the j th row. Alternating direction implicit (ADI) methods use this basic idea to solve the Dirichlet problem *iteratively*. A complete iteration step consists of two halves. In the first half, Equation 10.4 is applied in every row in the grid. In the second half, Equation 10.5 is applied in every column of the grid. The most commonly used ADI method is the one proposed by Peaceman and Rachford, sometimes referred to as PRADI.

10.2.2.1 Peaceman–Rachford Alternating Direction Implicit (PRADI) Method

Choose arbitrary starting value $u_{ij}^{(0)}$ at each interior mesh point (i, j) . The first iteration consists of two halves. In the first half, update the values of u_{ij} row by row, in a manner suggested by Equation 10.4,

$$j\text{th row } (j = 1, 2, \dots, M) \quad u_{i-1,j}^{(1/2)} - 4u_{ij}^{(1/2)} + u_{i+1,j}^{(1/2)} = -u_{i,j-1}^{(0)} - u_{i,j+1}^{(0)}, \quad i = 1, 2, \dots, N \quad (10.6)$$

Note that some of the u values are the known boundary values, which are not affected by the starting values and remain unchanged throughout the process. For each fixed j , Equation 10.6 is applied to all interior mesh points along row j and produces N equations. Since there are M rows, a total of MN equations will be generated. This system has a tridiagonal coefficient matrix, by design, which can then be solved efficiently using the Thomas method (Chapter 4). The solution at this stage represents the half-updated values with superscript of $(1/2)$.

In the second half, the values of $u_{ij}^{(1/2)}$ will be updated column by column, as suggested by Equation 10.5,

$$i\text{th column } (i = 1, 2, \dots, N) \quad u_{i,j-1}^{(1)} - 4u_{ij}^{(1)} + u_{i,j+1}^{(1)} = -u_{i-1,j}^{(1/2)} - u_{i+1,j}^{(1/2)}, \quad j = 1, 2, \dots, M \quad (10.7)$$

For each fixed i , Equation 10.7 is applied to all interior mesh points along column i and produces M equations. Once again, the values at the boundary points are not affected and remain unchanged. Since there are N columns, a total of MN equations will be generated. This system also has a tridiagonal coefficient matrix, which can be solved efficiently using the Thomas method. The solution at this stage represents the updated values with superscript of (1) . This completes the first iteration. The second iteration has two halves. In the first half,

$$j\text{th row } (j = 1, 2, \dots, M) \quad u_{i-1,j}^{(3/2)} - 4u_{ij}^{(3/2)} + u_{i+1,j}^{(3/2)} = -u_{i,j-1}^{(1)} - u_{i,j+1}^{(1)}, \quad i = 1, 2, \dots, N$$

generates a system of MN equations with a tridiagonal coefficient matrix. In the second half,

$$i\text{th column } (i=1,2,\dots,N) \quad u_{i,j-1}^{(2)} - 4u_{ij}^{(2)} + u_{i,j+1}^{(2)} = -u_{i-1,j}^{(3/2)} - u_{i+1,j}^{(3/2)}, \quad j=1,2,\dots,M$$

which also generates a system of MN equations with a tridiagonal coefficient matrix. The solution gives the updated values with superscript of (2). The procedure is repeated until some type of convergence is observed. This, of course, requires a terminating condition. One reasonable terminating condition is as follows. Assemble the values at the interior mesh points into a matrix, with the same configuration as the grid. If the norm of the difference between two successive such matrices is less than a prescribed tolerance, terminate the iterations.

The user-defined function PRADI uses the Peaceman–Rachford ADI method to numerically solve Poisson’s equation—or Laplace’s equation—in a rectangular region with the values of the unknown solution available on the boundary. The function returns the approximate solution at the interior mesh points, as well as the values at the boundary points in a pattern that resembles the gridded region. In addition, it returns the 3D plot of the results.

```
function [U, k] = PRADI(x,y,f,uleft,uright,ubottom,utop,tol,kmax)
%
% PRADI numerically solves an elliptic PDE with Dirichlet boundary
% conditions over a rectangular region using the Peaceman-Rachford
% alternating direction implicit method.
%
% [U, k] = PRADI(x,y,f,uleft,uright,ubottom,utop,tol,kmax), where
%
% x is the 1-by-m vector of mesh points in the x direction,
% y is the 1-by-n vector of mesh points in the y direction,
% f is an anonymous function representing f(x,y),
% ubottom,uleft,utop,uright are anonymous functions describing the
% boundary conditions,
% tol is the tolerance (default 1e-4),
% kmax is the maximum number of iterations (default 50),
%
% U is the solution at the mesh points,
% k is the number of (full) iterations needed to meet the tolerance.
%
% Note: The default starting value at all mesh points is 0.5.
%
if nargin < 9 || isempty(kmax), kmax = 50; end
if nargin < 8 || isempty(tol), tol = 1e-4; end
y = y';
[X,Y] = meshgrid(x(2:end-1),y(2:end-1)); % Create mesh grid
m = size(X,2); n = size(X,1); N = m*n;
u = 0.5*ones(n,m); % Starting values
h = x(2)-x(1);
```



```

% Define boundary conditions
for i = 2:m+1,
    utop_bound(i-1) = utop(x(i));
    ubottom_bound(i-1) = ubottom(x(i));
end
for i = 1:n+2,
    uleft_bound(i)=uleft(y(i));
    uright_bound(i)=uright(y(i));
end
U = [ubottom_bound;u;utop_bound]; U = [uleft_bound' U uright_bound'];
% Generate matrix A1 (first half) and A2 (second half).
A = diag(-4*ones(N,1));
v1 = diag(A,1)+1; v1(m:m:N-1) = 0;
v2 = diag(A,-1)+1; v2(n:n:N-1) = 0;
A2 = diag(v2,1)+diag(v2,-1)+A;
A1 = diag(v1,1)+diag(v1,-1)+A;
U1 = U;
for i = 1:N, % Initialize vector b
    b0(i) = h^2*f(X(i),Y(i));
end
b0 = reshape(b0,n,m);
for k = 1:kmax,
    % First half
    b = b0-U1(1:end-2,2:end-1)-U1(3:end,2:end-1);
    b(:,1) = b(:,1)-U(2:end-1,1);
    b(:,end) = b(:,end)-U(2:end-1,end);
    b = reshape(b',N,1);
    u = ThomasMethod(A1,b); % Solve tridiagonal system
    u = reshape(u,m,n);
    U1 = [U(1,2:end-1);u';U(end,2:end-1)];
    U1 = [U(:,1) U1 U(:,end)];
    % Second half
    b = b0-U1(2:end-1,1:end-2)-U1(2:end-1,3:end);
    b(1,:) = b(1,)-U(1,2:end-1);
    b(end,:) = b(end,)-U(end,2:end-1);
    b = reshape(b,N,1);
    u = ThomasMethod(A2,b); % Solve tridiagonal system
    u = reshape(u,n,m);
    U2 = [U(1,2:end-1);u;U(end,2:end-1)];
    U2 = [U(:,1) U2 U(:,end)];
    if norm(U2-U1,inf)<=tol, break, end;
    U1 = U2;
end
[X,Y] = meshgrid(x,y);
U = U1;
for i = 1:n+2,
    W(i,:) = U(n-i+3,:);
    YY(i) = Y(n-i+3);
end
U = W; Y = YY;
surf(X,Y,U);
xlabel('x');ylabel('y');

```

EXAMPLE 10.2: PRADI METHOD

For the Dirichlet problem described in Figure 10.4,

1. Perform one complete step of the PRADI method with $h = 1$ and starting values of 0.5 for all interior mesh points.
2. Solve the Dirichlet problem using the user-defined function PRADI with default parameter values.

Solution

1. *First half.* There are two rows and two columns in the grid, and a total of four interior mesh points. Equation 10.6 is first applied in the first row ($j = 1$). Since there are two mesh points in this row, Equation 10.6 is applied twice:

$$j = 1 \quad \begin{cases} (i = 1) & u_{01} - 4u_{11}^{(1/2)} + u_{21}^{(1/2)} = -u_{10} - u_{12}^{(0)} \\ (i = 2) & u_{11}^{(1/2)} - 4u_{21}^{(1/2)} + u_{31} = -u_{20} - u_{22}^{(0)} \end{cases} \quad (10.8)$$

Note that we have omitted superscripts for boundary values because they remain unchanged. We next apply Equation 10.6 at the two mesh points along the second row ($j = 2$):

$$j = 2 \quad \begin{cases} (i = 1) & u_{02} - 4u_{12}^{(1/2)} + u_{22}^{(1/2)} = -u_{11}^{(0)} - u_{13} \\ (i = 2) & u_{12}^{(1/2)} - 4u_{22}^{(1/2)} + u_{32} = -u_{21}^{(0)} - u_{23} \end{cases} \quad (10.9)$$

Combining Equations 10.8 and 10.9, and using the available boundary values as well as the starting values, we arrive at

$$\begin{bmatrix} -4 & 1 & 0 & 0 \\ 1 & -4 & 0 & 0 \\ 0 & 0 & -4 & 1 \\ 0 & 0 & 1 & -4 \end{bmatrix} \begin{Bmatrix} u_{11}^{(1/2)} \\ u_{21}^{(1/2)} \\ u_{12}^{(1/2)} \\ u_{22}^{(1/2)} \end{Bmatrix} = \begin{Bmatrix} -1.3660 \\ -1.3660 \\ -0.5 \\ -0.5 \end{Bmatrix} \quad \begin{matrix} \text{Thomas method} \\ \Rightarrow \end{matrix} \quad \begin{cases} u_{11}^{(1/2)} = 0.4553 = u_{21}^{(1/2)} \\ u_{12}^{(1/2)} = 0.1667 = u_{22}^{(1/2)} \end{cases}$$

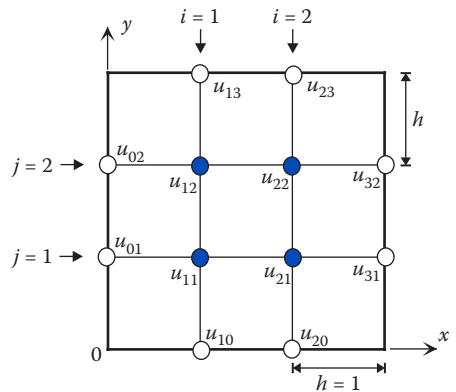
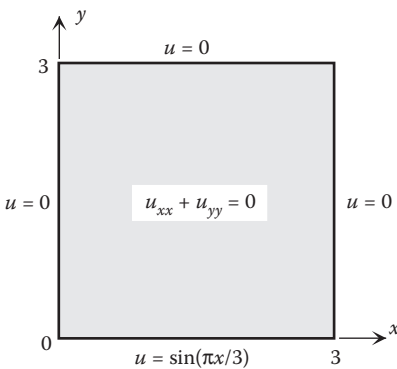


FIGURE 10.4

The Dirichlet problem in Example 10.2.

Second half. Equation 10.7 is applied in the first column ($i = 1$). Since there are two mesh points in this column, it is applied twice:

$$i = 1 \quad \begin{cases} (j = 1) & u_{10} - 4u_{11}^{(1)} + u_{12}^{(1)} = -u_{01} - u_{21}^{(1/2)} \\ (j = 2) & u_{11}^{(1)} - 4u_{12}^{(1)} + u_{13} = -u_{02} - u_{22}^{(1/2)} \end{cases} \quad (10.10)$$

We next apply Equation 10.7 at the two mesh points along the second column ($i = 2$):

$$i = 2 \quad \begin{cases} (j = 1) & u_{20} - 4u_{21}^{(1)} + u_{22}^{(1)} = -u_{11}^{(1/2)} - u_{31} \\ (j = 2) & u_{21}^{(1)} - 4u_{22}^{(1)} + u_{23} = -u_{12}^{(1/2)} - u_{32} \end{cases} \quad (10.11)$$

Combining Equations 10.10 and 10.11, and using the available boundary values as well as the updated values from the previous half iteration, we arrive at

$$\begin{bmatrix} -4 & 1 & 0 & 0 \\ 1 & -4 & 0 & 0 \\ 0 & 0 & -4 & 1 \\ 0 & 0 & 1 & -4 \end{bmatrix} \begin{Bmatrix} u_{11}^{(1)} \\ u_{12}^{(1)} \\ u_{21}^{(1)} \\ u_{22}^{(1)} \end{Bmatrix} = \begin{Bmatrix} -1.3213 \\ -0.1667 \\ -1.3213 \\ -0.1667 \end{Bmatrix} \quad \begin{array}{l} \text{Thomas method} \\ \Rightarrow \end{array} \quad \begin{array}{l} u_{11}^{(1)} = 0.3635 = u_{21}^{(1)} \\ u_{12}^{(1)} = 0.1325 = u_{22}^{(1)} \end{array}$$

2. Executing the user-defined function PRADI (with default parameters) yields the solution estimates at the interior mesh points. Note that the plot has been suppressed.

```
>> x = 0:1:3; y = 0:1:3; f = @(x,y)(0);
>> ubottom = @(x)(sin(pi*x/3)); utop = @(x)(0); uleft = @(y)(0);
uright = @(y)(0);
>> [U, k] = PRADI(x,y,f,uleft,uright,ubottom,utop)
```

```
U =
    0         0         0         0
    0    0.1083    0.1083         0
    0    0.3248    0.3248         0
    0    0.8660    0.8660         0
```

```
k =
    5
```

Therefore, convergence occurs after five iterations. The numerical results obtained in (1) can be verified by letting the function PRADI perform one iteration only. Setting kmax=1 and executing the function results in

```
>> [U] = PRADI(x,y,f,uleft,uright,ubottom,utop,[],1)
U =
    0         0         0         0
    0    0.1325    0.1325         0
    0    0.3635    0.3635         0
    0    0.8660    0.8660         0
```

The numerical values for $u_{11}^{(1)}$, $u_{21}^{(1)}$, $u_{12}^{(1)}$, $u_{22}^{(1)}$ agree with those in (1).

10.2.3 Neumann Problem

In the formulation of the Neumann problem, the values of the normal derivatives of u are prescribed along the boundary (Figure 10.5). As before, we are focusing on Laplace's

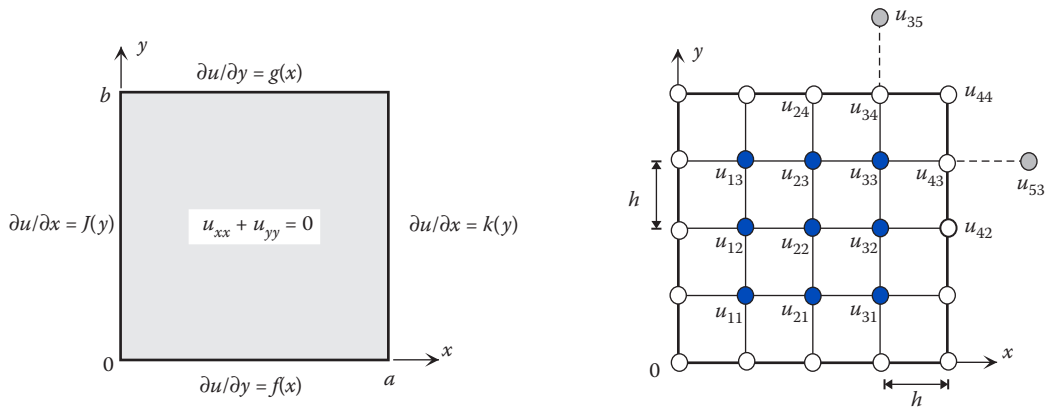


FIGURE 10.5

The Neumann problem.

equation in a rectangular region. In solving the Dirichlet problem, the objective was to find the solution estimates at all of the interior mesh points by taking advantage of the known values of u along the boundary. In solving the Neumann problem, u is no longer available on the boundary, hence the boundary points become part of the unknown vector.

Suppose the difference equation for Laplace's equation, Equation 10.3, is applied at the (3, 3) mesh point:

$$u_{23} + u_{32} - 4u_{33} + u_{43} + u_{34} = 0 \quad (10.12)$$

Interior mesh points
Boundary points

The three interior mesh points are part of the unknown vector. Since boundary values u_{43} and u_{34} are not available, they must also be included in the unknown vector. Similar situations arise when we apply Equation 10.3 at near-boundary points. However, inclusion of all boundary points that lie on the grid in the unknown vector substantially increases the size of the unknown vector. For example, in Figure 10.5, there are 25 unknowns: nine in the interior and 16 on the boundary. But since Equation 10.3 is applied at the interior mesh points, there are as many equations as there are interior mesh points. In Figure 10.5, for example, there will only be nine equations, while there are 25 unknowns. Therefore, several more equations are needed to completely solve the ensuing system of equations.

In order to generate these additional (auxiliary) equations, we apply Equation 10.3 at each of the marked boundary points. Since we are currently concentrating on u_{43} and u_{34} , we apply Equation 10.3 at these two points

$$\begin{aligned} u_{24} + u_{44} + u_{35} + u_{33} - 4u_{34} &= 0 \\ u_{42} + u_{44} + u_{53} + u_{33} - 4u_{43} &= 0 \end{aligned} \quad (10.13)$$

In Equation 10.13, there are two quantities that are not part of the grid and need to be eliminated: u_{53} and u_{35} . We will do this by extending the grid beyond the boundary of the region, and using the information on the vertical and horizontal boundary segments they are located on. At the (3, 4) boundary point, we have access to $\partial u / \partial y = g(x)$. Let us call it g_{34} . Applying the two-point central difference formula (Chapter 6) at (3, 4), we find

$$\left. \frac{\partial u}{\partial y} \right|_{(3,4)} = g_{34} = \frac{u_{35} - u_{33}}{2h} \quad \text{Solve for } u_{35} \quad \Rightarrow \quad u_{35} = u_{33} + 2hg_{34}$$

Similarly, at (4, 3),

$$\left. \frac{\partial u}{\partial x} \right|_{(4,3)} = k_{43} = \frac{u_{53} - u_{33}}{2h} \quad \text{Solve for } u_{53} \quad \Rightarrow \quad u_{53} = u_{33} + 2hk_{43}$$

Substitution of these two relations into Equation 10.13 creates two new equations that involve only the interior mesh points and the boundary points. In order to proceed with this approach, we need to assume that Laplace's equation is valid beyond the rectangular region, at least in the exterior area that contains the newly produced points such as u_{53} and u_{35} . If we continue with this strategy, we will end up with a system containing as many equations as the total number of interior mesh points and the boundary points. In Figure 10.5, for instance, the system will consist of 25 equations and 25 unknowns.

10.2.3.1 Existence of a Solution for the Neumann Problem

The existence of a solution for the Neumann problem entirely depends on the nature of the normal derivatives $u_n = \partial u / \partial n$ prescribed along different portions of the boundary. In fact, no solution is possible unless the line integral of the normal derivative taken over the boundary is zero

$$\int_C \frac{\partial u}{\partial n} ds = 0 \quad (10.14)$$

EXAMPLE 10.3: NEUMANN PROBLEM

Consider the Neumann problem described in Figure 10.6 where the grid is constructed with $h = 1$. Using the approach outlined above, 12 unknowns are generated: two interior mesh points and 10 boundary points.

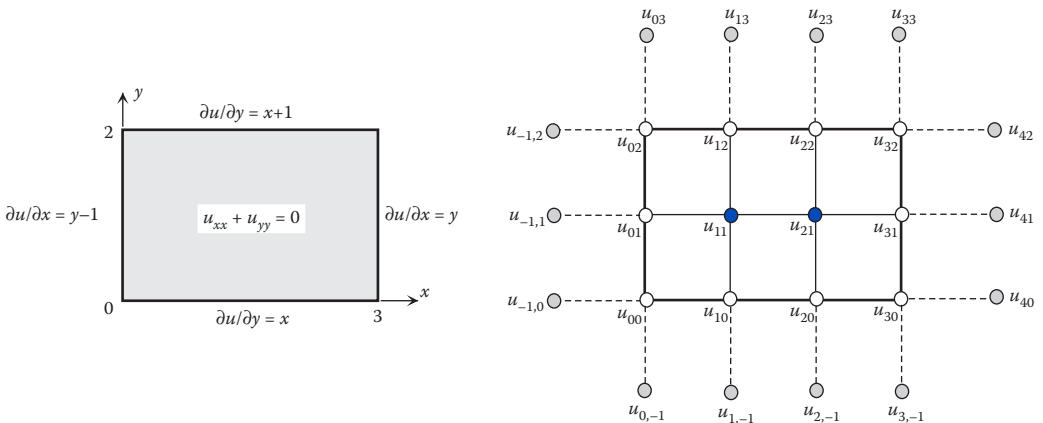


FIGURE 10.6 The Neumann problem in Example 10.3.

First, the condition for the existence of the solution, Equation 10.14, must be examined. Let the bottom edge of the region be denoted by C_1 , and continuing counterclockwise, the rest of the edges C_2 , C_3 , and C_4 . Then, it can be shown that (verify)

$$\begin{aligned} \int_C \frac{\partial u}{\partial n} ds &= \int_{C_1} \frac{\partial u}{\partial n} ds + \int_{C_2} \frac{\partial u}{\partial n} ds + \int_{C_3} \frac{\partial u}{\partial n} ds + \int_{C_4} \frac{\partial u}{\partial n} ds \\ &= \int_0^3 x dx + \int_0^2 y dy - \int_0^3 (x+1) dx - \int_0^2 (y-1) dy = -1 \neq 0 \end{aligned}$$

Therefore, the problem described in Figure 10.6 does not have a solution. In fact, if we had proceeded with the strategy introduced earlier, we would have obtained a 12×12 system of equations with a singular coefficient matrix.

10.2.4 Mixed Problem

In a mixed problem, the dependent function u is prescribed along some portions of the boundary, while $u_n = \partial u / \partial n$ is known along other portions. The numerical solution of these types of problems involves the same idea as in the case of the Neumann problem, with a lower degree of complexity. This is because u_n is only dealt with on certain segments of the boundary; hence the region does not need to be entirely extended and the ensuing linear system to be solved is not as large either.

EXAMPLE 10.4: MIXED PROBLEM

Solve the mixed problem described in Figure 10.7 using the grid with $h = 1$.

Solution

Because the Dirichlet boundary conditions are prescribed along the left, lower, and upper edges, no extension of region is needed there. The only extension pertains to the $(3, 1)$ mesh point, where u is unknown, resulting in the creation of u_{41} . Application of Equation 10.3 at the two interior mesh points, as well as at $(3, 1)$, yields

$$\begin{aligned} (1, 1) \quad & u_{01} + u_{10} + u_{21} + u_{12} - 4u_{11} = 0 \\ (2, 1) \quad & u_{11} + u_{20} + u_{31} + u_{22} - 4u_{21} = 0 \\ (3, 1) \quad & u_{21} + u_{30} + u_{41} + u_{32} - 4u_{31} = 0 \end{aligned} \tag{10.15}$$

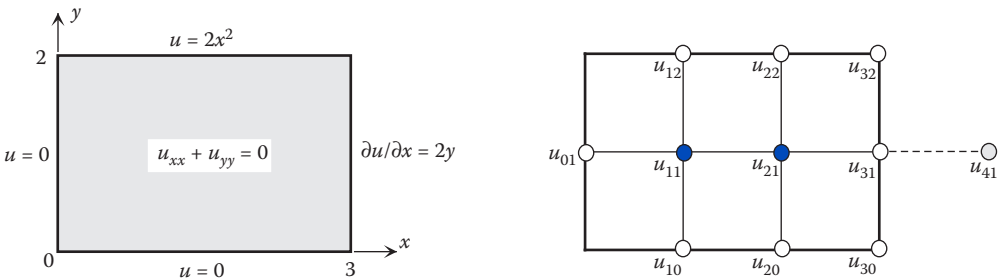


FIGURE 10.7 The mixed problem in Example 10.4.

To eliminate u_{41} , we use the two-point central difference formula at (3, 1):

$$\left. \frac{\partial u}{\partial x} \right|_{(3,1)} = [2y]_{(3,1)} = 2 = \frac{u_{41} - u_{21}}{2} \quad \text{Solve for } u_{41} \Rightarrow u_{41} = u_{21} + 4$$

Substitution of $u_{41} = u_{21} + 4$, as well as the boundary values provided by boundary conditions, into Equation 10.15 yields

$$\begin{aligned} u_{21} + 2 - 4u_{11} &= 0 \\ u_{11} + u_{31} + 8 - 4u_{21} &= 0 \\ u_{21} + (u_{21} + 4) + 18 - 4u_{31} &= 0 \end{aligned} \quad \Rightarrow \quad \text{Matrix form} \quad \begin{bmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 2 & -4 \end{bmatrix} \begin{Bmatrix} u_{11} \\ u_{21} \\ u_{31} \end{Bmatrix} = \begin{Bmatrix} -2 \\ -8 \\ -22 \end{Bmatrix} \quad \Rightarrow \quad \begin{aligned} u_{11} &= 1.5769 \\ u_{21} &= 4.3077 \\ u_{31} &= 7.6538 \end{aligned}$$

10.2.5 More Complex Regions

So far, we have studied boundary-value problems for elliptic PDEs in regions with relatively simple shapes, specifically, rectangular regions. As a result, it was possible to suitably adjust the grid so that some of the mesh points are located on the boundary of the region. But in many applications, the geometry of the problem is not as simple as a rectangle. And, as a result, the boundary of the region crosses the grid at points that are not mesh points.

As an example, consider the problem of solving the 2D Laplace's equation ($u_{xx} + u_{yy} = 0$) in the region shown in Figure 10.8a. The region has an irregular boundary in that the curved portion intersects the grid at points A and B , neither of which is a mesh point. The points M and Q are treated as before because each has four adjacent mesh points that are located on the grid. But a point such as P must be treated differently since two of its adjacent points (A and B) are not on the grid. The objective therefore is to derive expressions for $u_{xx}(P)$ and $u_{yy}(P)$, at mesh point P , to form a new difference equation for Laplace's equation. Assume that A is located a distance of αh to the right of P , and B is a distance of βh above P . Write the Taylor's series expansion for u at the four points A , B , M , and N about the point P . For example, $u(M)$ and $u(A)$ are expressed as

$$u(M) = u(P) - h \frac{\partial u(P)}{\partial x} + \frac{1}{2!} h^2 \frac{\partial^2 u(P)}{\partial x^2} - \dots \quad (10.16)$$

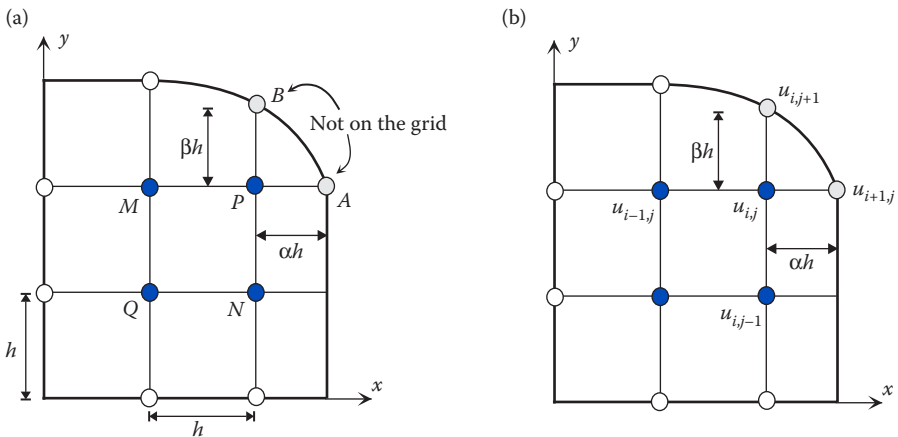


FIGURE 10.8
A region with irregular boundary.

$$u(A) = u(P) + (\alpha h) \frac{\partial u(P)}{\partial x} + \frac{1}{2!} (\alpha h)^2 \frac{\partial^2 u(P)}{\partial x^2} + \dots \quad (10.17)$$

Multiply Equation 10.16 by α and add the result to Equation 10.17, while neglecting the terms involving h^2 and higher:

$$u(A) + \alpha u(M) \cong (\alpha + 1)u(P) + \frac{h^2}{2} \alpha(\alpha + 1) \frac{\partial^2 u(P)}{\partial x^2}$$

Solving for $\partial^2 u(P)/\partial x^2$, we find

$$\frac{\partial^2 u(P)}{\partial x^2} = \frac{2}{h^2} \left[\frac{1}{\alpha(\alpha + 1)} u(A) + \frac{1}{\alpha + 1} u(M) - \frac{1}{\alpha} u(P) \right] \quad (10.18)$$

Similarly, expanding $u(N)$ and $u(B)$ about P , and proceeding as above, yields

$$\frac{\partial^2 u(P)}{\partial y^2} = \frac{2}{h^2} \left[\frac{1}{\beta(\beta + 1)} u(B) + \frac{1}{\beta + 1} u(N) - \frac{1}{\beta} u(P) \right] \quad (10.19)$$

Addition of Equations 10.18 and 10.19 gives

$$u_{xx}(P) + u_{yy}(P) = \frac{2}{h^2} \left[\frac{1}{\alpha(\alpha + 1)} u(A) + \frac{1}{\beta(\beta + 1)} u(B) + \frac{1}{\alpha + 1} u(M) + \frac{1}{\beta + 1} u(N) - \left(\frac{1}{\alpha} + \frac{1}{\beta} \right) u(P) \right] \quad (10.20)$$

If Laplace's equation is solved, then the left side of Equation 10.20 is set to zero, and we have

$$\frac{1}{\alpha(\alpha + 1)} u(A) + \frac{1}{\beta(\beta + 1)} u(B) + \frac{1}{\alpha + 1} u(M) + \frac{1}{\beta + 1} u(N) - \frac{\alpha + \beta}{\alpha\beta} u(P) = 0 \quad (10.21)$$

With the usual notations involved in [Figure 10.8b](#), the difference equation is obtained by rewriting Equation 10.21, as

$$\frac{1}{\alpha(\alpha + 1)} u_{i+1,j} + \frac{1}{\beta(\beta + 1)} u_{i,j+1} + \frac{1}{\alpha + 1} u_{i-1,j} + \frac{1}{\beta + 1} u_{i,j-1} - \frac{\alpha + \beta}{\alpha\beta} u_{i,j} = 0 \quad (10.22)$$

The case of Poisson's equation can be handled in a similar manner. Equation 10.22 is applied at any mesh point that has at least one adjacent point not located on the grid. In [Figure 10.8](#), for example, that would be points P and N . For the points M and Q , we simply apply Equation 10.3, as before, or equivalently Equation 10.22 with $\alpha = 1 = \beta$.

EXAMPLE 10.5: IRREGULAR BOUNDARY

Solve $u_{xx} + u_{yy} = 0$ in the region shown in [Figure 10.9](#) subject to the given boundary conditions. The slanting segment of the boundary obeys $y = -\frac{2}{3}x + 2$.

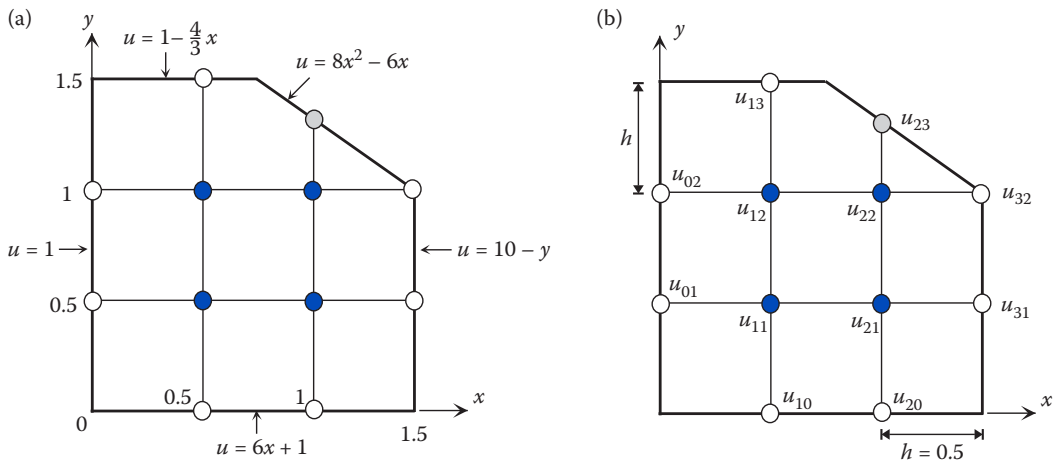


FIGURE 10.9
Example 10.5.

Solution

Based on the grid shown in Figure 10.9, Equation 10.3 can be applied at mesh points $(1, 1)$, $(2, 1)$, and $(1, 2)$ because all four neighboring points at those mesh points are on the grid. Using the boundary conditions provided, the resulting difference equations are

$$\begin{aligned} 1 + 4 + u_{12} + u_{21} - 4u_{11} &= 0 \\ u_{11} + 7 + 9.5 + u_{22} - 4u_{21} &= 0 \\ 1 + u_{11} + \frac{1}{3} + u_{22} - 4u_{12} &= 0 \end{aligned} \quad (10.23)$$

However, at $(2, 2)$, we need to use Equation 10.22. Using the equation of the slanted segment, we find that the point at the $(2, 3)$ location is a vertical distance of $\frac{1}{3}$ from the $(2, 2)$ mesh point. But since $h = \frac{1}{2}$, we have $\beta = \frac{2}{3}$. On the other hand, $\alpha = 1$ since the neighboring point to the right of $(2, 2)$ is itself a mesh point. With these, and knowing $u_{32} = 9$ by the given boundary condition, Equation 10.22 yields

$$\frac{9}{2} + \frac{1}{\frac{2}{3} \cdot \frac{5}{3}}(2) + u_{12} + \frac{1}{\frac{5}{3}}u_{21} - \frac{1 + \frac{2}{3}}{\frac{2}{3}}u_{22} = 0$$

Combining this with Equation 10.23, and simplifying, we find

$$\begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1.2 & 1 & -5 \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{21} \\ u_{12} \\ u_{22} \end{bmatrix} = \begin{bmatrix} -5 \\ -16.5 \\ -1.3333 \\ -12.6 \end{bmatrix} \Rightarrow \begin{aligned} u_{11} &= 3.3354 \\ u_{21} &= 6.0666 \\ u_{12} &= 2.2749 \\ u_{22} &= 4.4310 \end{aligned}$$

10.3 Parabolic Partial Differential Equations

In this section, we will present two techniques for the numerical solution of parabolic PDEs: the finite-difference method and Crank–Nicolson method. In both cases, the objective remains the same as before, namely, derive a suitable difference equation that can be used to generate solution estimates at mesh points. Contrary to the numerical solution of elliptic PDEs, there is no guarantee that the difference equations for parabolic equations converge, regardless of the grid size. In these situations, it turns out that convergence can be assured as long as some additional conditions are imposed.

10.3.1 Finite-Difference Method

The 1D heat equation, $u_t = \alpha^2 u_{xx}$ ($\alpha = \text{const} > 0$), is the simplest model of a physical system involving a parabolic PDE. Specifically, consider a laterally insulated wire of length L with its ends kept at zero temperature, and subjected to the initial temperature along the wire prescribed by $f(x)$. The boundary-initial-value problem at hand is

$$u_t = \alpha^2 u_{xx} \quad (\alpha = \text{const} > 0), \quad 0 \leq x \leq L, \quad t \geq 0 \quad (10.24)$$

$$u(0, t) = 0 = u(L, t) \quad (10.25)$$

$$u(x, 0) = f(x) \quad (10.26)$$

Figure 10.10 shows a grid constructed by using a mesh size of h in the x -direction and a mesh size of k in the t -direction. As usual, the partial derivatives in Equation 10.24 must be replaced by their finite-difference approximations. The term u_{xx} is approximated by the three-point central-difference formula. For the term u_t , however, we must use the two-point forward-difference formula, as opposed to the more accurate central-difference formula.

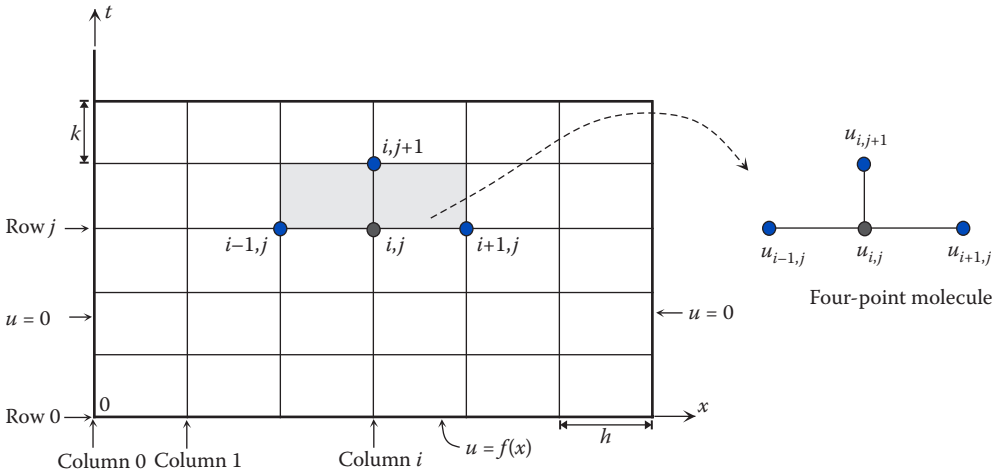


FIGURE 10.10 Region and grid used for solving the 1D heat equation.

This is because starting at $t = 0$ (the x -axis), we can only progress in the positive direction of the t -axis and have no knowledge of $u(t)$ when $t < 0$. Consequently, Equation 10.24 yields

$$\frac{1}{k}(u_{i,j+1} - u_{ij}) = \frac{\alpha^2}{h^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}) \quad (10.27)$$

Referring to the four-point molecule in [Figure 10.10](#), knowing $u_{i-1,j}$, $u_{i,j}$, and $u_{i+1,j}$, we can find $u_{i,j+1}$ at the higher level on the t -axis, via

$$u_{i,j+1} = \left[1 - \frac{2k\alpha^2}{h^2} \right] u_{i,j} + \frac{k\alpha^2}{h^2} (u_{i-1,j} + u_{i+1,j})$$

which simplifies to

$$u_{i,j+1} = (1 - 2r)u_{ij} + r(u_{i-1,j} + u_{i+1,j}), \quad r = \frac{k\alpha^2}{h^2} \quad (10.28)$$

and is known as the difference equation for the 1D heat equation using the finite-difference method.

10.3.1.1 Stability and Convergence of the Finite-Difference Method

A numerical method is said to be stable if any small changes in the initial data result in small changes in the subsequent data, or errors such as round-off introduced at any time remain bounded throughout. What is meant by convergence is that the approximate solution tends to the actual solution as the computational grid gets very fine, that is, as $h, k \rightarrow 0$. It can then be shown that the finite-difference method outlined in Equation 10.28 is both stable and convergent if *

$$r = \frac{k\alpha^2}{h^2} \leq \frac{1}{2} \quad (10.29)$$

and unstable and divergent when $r > \frac{1}{2}$.

The user-defined function `Heat1DFD` uses the finite-difference approach to solve the 1D heat equation subject to zero boundary conditions and a prescribed initial condition. The function returns the approximate solution at the interior mesh points, as well as the values at the boundary points in a pattern that resembles the gridded region. A warning message is returned if $r > \frac{1}{2}$.

```
function u = Heat1DFD(t,x,u0,alpha)
%
% Heat1DFD numerically solves the one-dimensional heat equation, with zero
% boundary conditions, using the finite-difference method.
%
```

* For details, refer to R.L. Burden and J.D. Faires, *Numerical Analysis*, Third edition, Prindle, Weber & Schmidt, 1985.

```

% u = Heat1DFD(t,x,u0,alpha), where
%
% t is the row vector of times,
% x is the row vector of x positions,
% u0 is the row vector of initial temperatures at the x positions,
% alpha is a given parameter of the heat equation,
%
% u is the approximate solution at the mesh points.
%
u = u0(:); % u must be a column vector
k = t(2)-t(1); h = x(2)-x(1); r = (alpha/h)^2*k;
if r > 0.5,
    warning('Method is unstable and divergent. Results will be inaccurate.')
end
i = 2:length(x)-1;
for j = 1:length(t)-1,
    u(i,j+1) = (1-2*r)*u(i,j) + r*(u(i-1,j)+u(i+1,j));
end
u = flipud(u');

```

EXAMPLE 10.6: FINITE-DIFFERENCE METHOD; 1D HEAT EQUATION

Consider a laterally insulated wire of length $L = 1$ and $\alpha = 0.5$, whose ends are kept at zero temperature, subjected to initial temperature $f(x) = 10 \sin \pi x$. Compute the approximate values of temperature $u(x, t)$, $0 \leq x \leq 1$, $0 \leq t \leq 0.5$, at mesh points generated by $h = 0.25$ and $k = 0.1$. All parameter values are in consistent physical units. The exact solution is given as

$$u(x, t) = (10 \sin \pi x) e^{-0.25\pi^2 t}$$

Solution

We first calculate

$$r = \frac{k\alpha^2}{h^2} = \frac{(0.1)(0.5)^2}{(0.25)^2} = 0.4 < \frac{1}{2}$$

which signifies stability and convergence for the finite-difference method described by Equation 10.28. With $r = 0.4$, Equation 10.28 reduces to

$$u_{i,j+1} = 0.2u_{ij} + 0.4(u_{i-1,j} + u_{i+1,j}) \quad (10.30)$$

This will generate approximate solutions at the mesh points marked in [Figure 10.11](#). The first application of Equation 10.30 is at the (1, 0) position so that

$$u_{11} = 0.2u_{10} + 0.4(u_{00} + u_{20}) \stackrel{\substack{\text{Using} \\ \text{boundary} \\ \text{values}}}{=} 0.2(7.0711) + 0.4(0 + 10) = 5.4142$$

It is next applied at (2, 0) to find u_{21} , and at (3, 0) to find u_{31} . This way, the values at the three mesh points along the time row $j = 1$ become available. Subsequently,

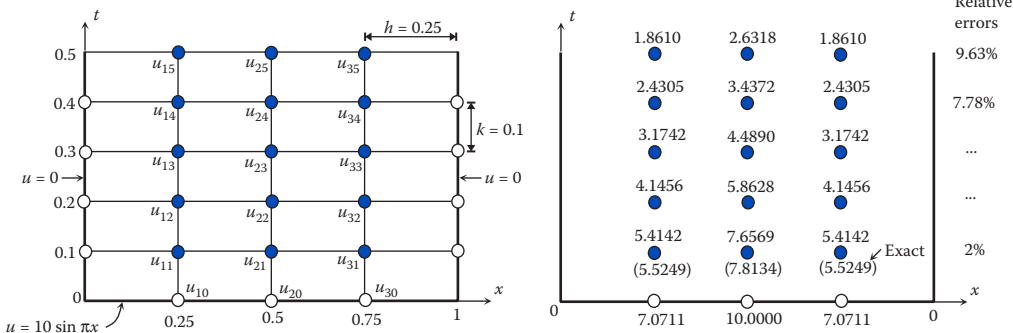


FIGURE 10.11

The grid and numerical results in Example 10.6.

Equation 10.30 is applied at the three mesh points on the current row ($j = 1$) to find the values at the next time level, $j = 2$, and so on. Continuing with this strategy, we will generate the approximate values shown in Figure 10.11. It is interesting to note that at the first time row, we are experiencing relative errors of around 2%, but this grows to about 9.63% by the time we arrive at the fifth time level.

The numerical results obtained in this manner can be readily verified by executing the user-defined function `Heat1DFD`:

```
>> t = 0:0.1:0.5; x = 0:0.25:1;
>> u0 = 10.*sin(pi*x);
>> u = Heat1DFD(t,x,u0,0.5)

u =
    0    1.8610    2.6318    1.8610    0
    0    2.4304    3.4372    2.4304    0
    0    3.1742    4.4890    3.1742    0
    0    4.1456    5.8627    4.1456    0
    0    5.4142    7.6569    5.4142    0
    0    7.0711   10.0000    7.0711    0
```

As stated earlier, the output resembles the gridded region.

10.3.2 Crank–Nicolson Method

The condition $r = \alpha^2/h^2 \leq \frac{1}{2}$, required by the finite-difference method for stability and convergence, could lead to serious computational problems. For example, suppose $\alpha = 1$ and $h = 0.2$. Then, $r \leq \frac{1}{2}$ imposes $k \leq 0.02$, requiring too many time steps. Moreover, reducing the mesh size h by half to $h = 0.1$ increases the number of time steps by a factor of 4. Generally, in order to decrease r , we must either decrease k or increase h . Decreasing k forces additional time levels to be generated which increases the amount of computations. Increasing h causes a reduction of accuracy.

The Crank–Nicolson method offers a technique for solving the 1D heat equation with no restriction on the ratio $r = \alpha^2/h^2$. The idea behind the Crank–Nicolson method is to employ a six-point molecule (Figure 10.12) as opposed to the four-point molecule (Figure 10.10) used with the finite-difference method.

To derive the difference equation associated with this method we consider Equation 10.27. On the right-hand side, we write two expressions similar to that inside parentheses:

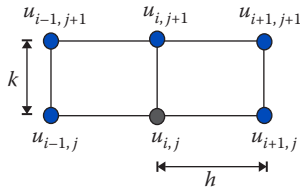


FIGURE 10.12

Six-point molecule used in Crank–Nicolson method.

one for the j th time row, one for the $(j + 1)$ st time row. Multiply each by $\alpha^2/2h^2$, and add them to obtain

$$\frac{1}{k}(u_{i,j+1} - u_{ij}) = \frac{\alpha^2}{2h^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}) + \frac{\alpha^2}{2h^2}(u_{i-1,j+1} - 2u_{i,j+1} + u_{i+1,j+1}) \quad (10.31)$$

Multiply Equation 10.31 by $2k$ and let $r = k\alpha^2/h^2$ as before, and rearrange the outcome so that the three terms associated with the higher time row [($j + 1$)st row] appear on the left side of the equation. The result is

$$2(1+r)u_{i,j+1} - r(u_{i-1,j+1} + u_{i+1,j+1}) = 2(1-r)u_{ij} + r(u_{i-1,j} + u_{i+1,j}), \quad r = \frac{k\alpha^2}{h^2} \quad (10.32)$$

This is called the difference equation for the 1D heat equation using the Crank–Nicolson method. Starting with the 0th time level ($j = 0$), each time Equation 10.32 is applied, the three values on the right side, $u_{i-1,j}$, u_{ij} , $u_{i+1,j}$, are available from the initial temperature $f(x)$, while the values at the higher time level ($j = 1$) are unknown. If there are n non-boundary mesh points in each row, the ensuing system of equations to be solved is $n \times n$ with a tridiagonal coefficient matrix. Solving the system yields the values of u at the mesh points along the $j = 1$ row. Repeating the procedure leads to the approximate values of u at all desired mesh points.

The user-defined function `Heat1DCN` uses the Crank–Nicolson method to solve the 1D heat equation subject to zero boundary conditions and a prescribed initial condition. The function returns the approximate solution at the interior mesh points, as well as the values at the boundary points in a pattern that resembles the gridded region.

```
function u = Heat1DCN(t,x,u0,alpha)
%
% Heat1DCN numerically solves the one-dimensional heat equation, with zero
% boundary conditions, using the Crank-Nicolson method.
%
% u = Heat1DCN(t,x,u0,alpha), where
%
% t is the row vector of times,
% x is the row vector of x positions,
% u0 is the row vector of initial temperatures at the x positions,
% alpha is a given parameter of the heat equation,
%
```

```

%      u is the approximate solution at the mesh points.
%
u = u0(:);    % u must be a column vector
k = t(2)-t(1); h = x(2)-x(1); r = (alpha/h)^2*k;

% Compute A
n = length(x);
A = diag(2*(1+r)*ones(n-2,1));
A = A + diag(diag(A,-1)-r,-1);
A = A + diag(diag(A,1)-r, 1);

% Compute B
B = diag(2*(1-r)*ones(n-2,1));
B = B + diag(diag(B,-1)+r,-1);
B = B + diag(diag(B,1)+r, 1);
C = A\B;

i = 2:length(x)-1;
for j = 1:length(t)-1,
    u(i,j+1) = C*u(i,j);
end
u = flipud(u');

```

EXAMPLE 10.7: CRANK–NICOLSON METHOD; 1D HEAT EQUATION

Consider the temperature distribution problem outlined in Example 10.6. Find the approximate values of $u(x, t)$ at the mesh points, and compare with the actual values as well as those obtained using the finite-difference method.

Solution

With $r = 0.4$, Equation 10.32 is written as

$$2.8u_{i,j+1} - 0.4(u_{i-1,j+1} + u_{i+1,j+1}) = 1.2u_{ij} + 0.4(u_{i-1,j} + u_{i+1,j}) \quad (10.33)$$

Applying Equation 10.33 at the $j = 0$ level, we find

$$2.8u_{11} - 0.4(u_{01} + u_{21}) = 1.2u_{10} + 0.4(u_{00} + u_{20})$$

$$2.8u_{21} - 0.4(u_{11} + u_{31}) = 1.2u_{20} + 0.4(u_{10} + u_{30})$$

$$2.8u_{31} - 0.4(u_{21} + u_{41}) = 1.2u_{30} + 0.4(u_{20} + u_{40})$$

Substituting the values from boundary and initial conditions, yields

$$\begin{bmatrix} 2.8 & -0.4 & 0 \\ -0.4 & 2.8 & -0.4 \\ 0 & -0.4 & 2.8 \end{bmatrix} \begin{Bmatrix} u_{11} \\ u_{21} \\ u_{31} \end{Bmatrix} = \begin{Bmatrix} 12.4853 \\ 17.6569 \\ 12.4853 \end{Bmatrix} \quad \begin{array}{l} \text{Solve} \\ \Rightarrow \\ \text{tridiagonal system} \end{array} \quad \begin{array}{l} u_{11} = 5.5880 = u_{31} \\ u_{21} = 7.9026 \end{array}$$

Next, Equation 10.33 is applied at the $j = 1$ level:

$$\begin{bmatrix} 2.8 & -0.4 & 0 \\ -0.4 & 2.8 & -0.4 \\ 0 & -0.4 & 2.8 \end{bmatrix} \begin{Bmatrix} u_{12} \\ u_{22} \\ u_{32} \end{Bmatrix} = \begin{Bmatrix} 9.8666 \\ 13.9535 \\ 9.8666 \end{Bmatrix} \quad \begin{array}{l} \text{Solve} \\ \Rightarrow \\ \text{tridiagonal system} \end{array} \quad \begin{array}{l} u_{12} = 4.4159 = u_{32} \\ u_{22} = 6.2451 \end{array}$$

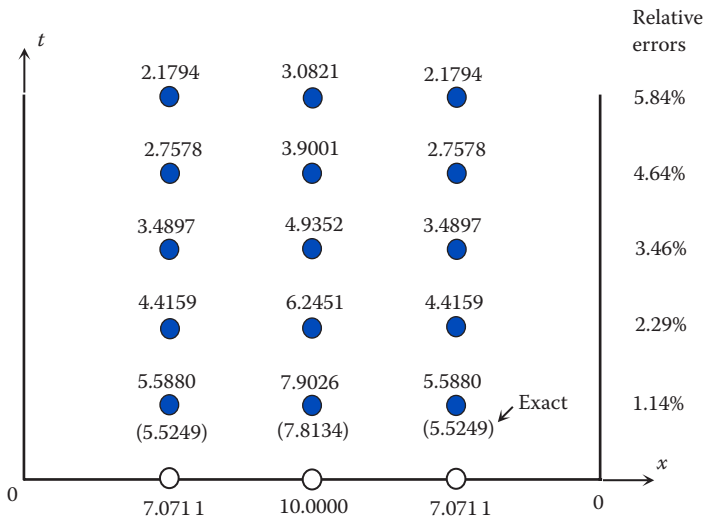


FIGURE 10.13

The grid and numerical results in Example 10.7.

This procedure is repeated until the approximate values of u at the mesh points along the $j = 5$ row are calculated. Execution of the user-defined function `Heat1DCN` yields

```
>> t = 0:0.1:0.5;
>> x = 0:0.25:1;
>> u0 = 10.*sin(pi*x);
>> u = Heat1DCN(t,x,u0,0.5)
u =
    0    2.1794    3.0821    2.1794    0
    0    2.7578    3.9001    2.7578    0
    0    3.4897    4.9352    3.4897    0
    0    4.4159    6.2451    4.4159    0
    0    5.5880    7.9026    5.5880    0
    0    7.0711   10.0000    7.0711    0
```

These numerical results, together with the associated relative errors, are shown in [Figure 10.13](#). Comparing the relative errors with those in Example 10.6, it is evident that the Crank–Nicolson method produces more accurate estimates. Note that the values returned by the Crank–Nicolson method overshoot the actual values, while those generated by the finite-difference method ([Figure 10.11](#)) undershoot the exact values.

10.3.2.1 Crank–Nicolson (CN) Method versus Finite-Difference (FD) Method

In Examples 10.6 and 10.7, the value of r satisfied the condition $r \leq \frac{1}{2}$ so that both FD and CN were successfully applied, with CN yielding more accurate estimates. There are other situations where r does not satisfy $r \leq \frac{1}{2}$, hence the FD method cannot be implemented. To implement FD, we must reduce the value of r , which is possible by either increasing h or decreasing k , causing reduction in accuracy and an increase in computations. The following example demonstrates that even with a substantial increase in the number of time steps—to assure stability and convergence of FD—the values provided by FD are not any more accurate than those provided by CN.

EXAMPLE 10.8: CRANK–NICOLSON VERSUS FINITE-DIFFERENCE

Consider the temperature distribution problem studied in Examples 10.6 and 10.7. Assume $h = 0.25$.

1. Use the CN method with $r = 1$ to find the temperature at the mesh points u_{11} , u_{21} , and u_{31} in the first time row.
2. Since $r = 1$ does not satisfy the condition of $r \leq \frac{1}{2}$, pick $r = 0.25$, for example, and $h = 0.25$ as before, and apply the FD method to find the values at the points u_{11} , u_{21} , and u_{31} in (1). Considering the number of time steps has quadrupled, decide whether FD generates more accurate results than CN.

Solution

1. We first note that

$$r = \frac{k\alpha^2}{h^2} = 1 \quad \begin{matrix} \alpha=0.5 \\ \Rightarrow \\ h=0.25 \end{matrix} \quad k = r \left(\frac{h}{\alpha} \right)^2 = 0.25$$

With $r = 1$, Equation 10.32 becomes

$$4u_{i,j+1} - u_{i-1,j+1} - u_{i+1,j+1} = u_{i-1,j} + u_{i+1,j}$$

Applying this equation with $j = 0$, we find

$$\begin{aligned} 4u_{11} - 0 - u_{21} &= 0 + 10 \\ 4u_{21} - u_{11} - u_{31} &= 2(7.0711) \\ 4u_{31} - u_{21} - 0 &= 10 + 0 \end{aligned} \Rightarrow \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix} \begin{Bmatrix} u_{11} \\ u_{21} \\ u_{31} \end{Bmatrix} = \begin{Bmatrix} 10 \\ 14.1422 \\ 10 \end{Bmatrix} \Rightarrow \begin{aligned} u_{11} &= u_{31} = 3.8673 \\ u_{21} &= 5.4692 \end{aligned}$$

2. We note that

$$r = \frac{k\alpha^2}{h^2} = 0.25 \quad \begin{matrix} \alpha=0.5 \\ \Rightarrow \\ h=0.25 \end{matrix} \quad k = r \left(\frac{h}{\alpha} \right)^2 = 0.0625$$

Therefore, the step size along the t -axis has been reduced from $k = 0.25$ to $k = 0.0625$, implying that four time-step calculations are required to find the values of u_{11} , u_{21} , and u_{31} in (1). With $r = 0.25$, Equation 10.28 reduces to

$$u_{i,j+1} = 0.5u_{ij} + 0.25(u_{i-1,j} + u_{i+1,j})$$

Proceeding as always, the solution estimates at the desired mesh points will be calculated. It should be mentioned that with the new, smaller step size, $k = 0.0625$, the old u_{11} , u_{21} , and u_{31} in (1) are now labeled u_{14} , u_{24} , and u_{34} . The computed values are

$$\begin{aligned} u_{14} &= u_{34} = 3.7533 \\ u_{24} &= 5.3079 \end{aligned}$$

The numerical results obtained in (1) and (2) are summarized in [Figure 10.14](#) where it is readily observed that although FD used four times as many time levels as CN, the accuracy of the results by CN is still superior.

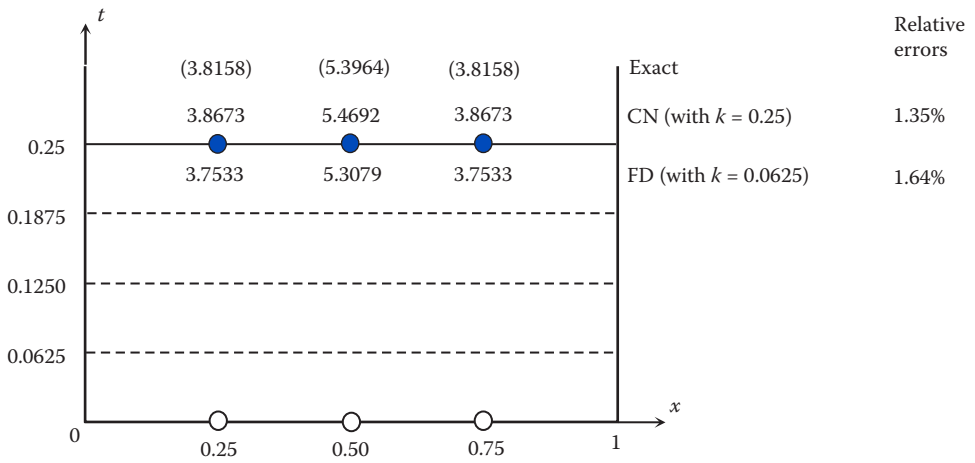


FIGURE 10.14

Accuracy and efficiency of Crank–Nicolson versus finite-difference.

10.4 Hyperbolic Partial Differential Equations

The 1D wave equation, $u_{tt} = \alpha^2 u_{xx}$ ($\alpha = \text{const} > 0$), is the simplest model of a physical system involving a hyperbolic PDE. Specifically, consider an elastic string of length L and fixed at both ends. Assuming the string is driven by initial displacement $f(x)$ and initial velocity $g(x)$ only, the free vibration of the string is governed by the boundary-initial-value problem

$$u_{tt} = \alpha^2 u_{xx} \quad (\alpha = \text{const} > 0), \quad 0 \leq x \leq L, \quad t \geq 0 \quad (10.34)$$

$$u(0, t) = 0 = u(L, t) \quad (10.35)$$

$$u(x, 0) = f(x), \quad u_t(x, 0) = g(x) \quad (10.36)$$

Figure 10.15 shows that a grid is constructed using a mesh size of h in the x -direction and a mesh size of k in the t -direction. The terms u_{xx} and u_{tt} in Equation 10.34 will be replaced by three-point central-difference approximations. Consequently, Equation 10.34 yields

$$\frac{1}{k^2} (u_{i,j-1} - 2u_{ij} + u_{i,j+1}) = \frac{\alpha^2}{h^2} (u_{i-1,j} - 2u_{i,j} + u_{i+1,j}) \quad (10.37)$$

Multiplying by k^2 , letting, $\tilde{r} = (k\alpha/h)^2$ and solving for $u_{i,j+1}$, we find

$$u_{i,j+1} = -u_{i,j-1} + 2(1 - \tilde{r})u_{ij} + \tilde{r}(u_{i-1,j} + u_{i+1,j}), \quad \tilde{r} = \left(\frac{k\alpha}{h}\right)^2 \quad (10.38)$$

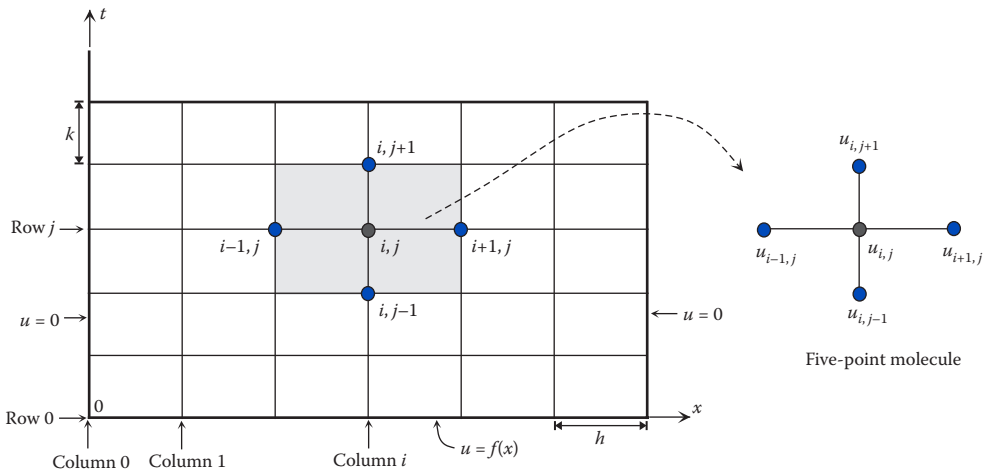


FIGURE 10.15

Region and grid used for solving the 1D wave equation.

which is known as the difference equation for the 1D wave equation using the finite-difference method. It can be shown that the numerical method described by Equation 10.38 is stable and convergent if $\tilde{r} \leq 1$.

10.4.1 Starting the Procedure

Applying Equation 10.38 along the $j = 0$ level, we have

$$u_{i,1} = -u_{i,-1} + 2(1 - \tilde{r})u_{i,0} + \tilde{r}(u_{i-1,0} + u_{i+1,0}) \quad (10.39)$$

The quantities $u_{i,0}$, $u_{i-1,0}$, and $u_{i+1,0}$ are available from the initial displacement, but $u_{i,-1}$ is not yet known. To find $u_{i,-1}$, we use the information on the initial velocity $u_t(x, 0) = g(x)$. Let $x_i = ih$ and $g_i = g(x_i)$. Using the central-difference formula for $u_t(x_i, 0)$,

$$\frac{u_{i,1} - u_{i,-1}}{2k} = g_i \quad \xRightarrow{\text{Solve for } u_{i,-1}} \quad u_{i,-1} = u_{i,1} - 2kg_i$$

Inserting this into Equation 10.39,

$$u_{i,1} = (1 - \tilde{r})u_{i,0} + \frac{1}{2}\tilde{r}(u_{i-1,0} + u_{i+1,0}) + kg_i \quad (10.40)$$

In summary, the finite-difference approximation for the 1D wave equation is implemented as follows. First apply Equation 10.40 using the knowledge of initial displacement and velocity. This gives the values of u along the first time step, $j = 1$. From this point on, apply 10.38 to find u at the mesh points on the higher time levels.

The user-defined function `Wave1DFD` uses the finite-difference approach to solve the 1D wave equation subject to zero boundary conditions and prescribed initial displacement and velocity. The function returns a failure message if $r = (k\alpha/h)^2 > 1$. The function returns

the approximate solution at the interior mesh points, as well as the values at the boundary points in a pattern that resembles the gridded region.

```
function u = Wave1DFD(t,x,u0,ut0,alpha)
%
% Wave1DFD numerically solves the one-dimensional wave equation, with zero
% boundary conditions, using the finite-difference method.
%
% u = Wave1DFD(t,x,u0,ut0,alpha), where
%
% t is the row vector of times,
% x is the row vector of x positions,
% u0 is the row vector of initial displacements for x positions,
% ut0 is the row vector of initial velocities for x positions,
% alpha is a given parameter of the wave equation,
%
% u is the approximate solution at the mesh points.
%
u = u0(:); ut = ut0(:); % u and ut must be column vectors
k = t(2)-t(1); h = x(2)-x(1); r = (k*alpha/h)^2;
if r > 1,
    warning('Method is unstable and divergent. Results will be inaccurate.')
end
i = 2:length(x)-1;
u(i,2) = (1-r)*u(i,1) + r/2*(u(i-1,1) + u(i+1,1)) + k*ut(i);
for j = 2:length(t)-1,
    u(i,j+1) = -u(i,j-1) + 2*(1-r)*u(i,j) + r*(u(i-1,j) + u(i+1,j));
end
u = flipud(u');
```

EXAMPLE 10.9: FREE VIBRATION OF AN ELASTIC STRING

Consider an elastic string of length $L = 2$ with $\alpha = 1$, fixed at both ends. Suppose the string is subject to an initial displacement $f(x) = 5 \sin(\pi x/2)$ and zero initial velocity, $g(x) = 0$. Using $h = 0.4 = k$, find the displacement $u(x, t)$ of the string for $0 \leq x \leq L$ and $0 \leq t \leq 2$. All parameters are in consistent physical units. The exact solution is given by

$$u(x, t) = 5 \sin \frac{\pi x}{2} \cos \frac{\pi t}{2}$$

Solution

We first calculate $\tilde{r} = (k\alpha/h)^2 = 1$. To find the values of u at the $j = 1$ level ($t = 0.4$), we apply Equation 10.40. But since $g_j = 0$ and $\tilde{r} = 1$, Equation 10.40 simplifies to

$$u_{i,1} = \frac{1}{2}(u_{i-1,0} + u_{i+1,0}), \quad i = 1, 2, 3, 4$$

so that

$$u_{11} = \frac{1}{2}(u_{00} + u_{20}), \quad u_{21} = \frac{1}{2}(u_{10} + u_{30}), \quad u_{31} = \frac{1}{2}(u_{20} + u_{40}), \quad u_{41} = \frac{1}{2}(u_{30} + u_{50})$$

This way, the estimates at all four interior mesh points along $j=1$ are determined. For higher time level, we use 10.38 which simplifies to

$$u_{i,j+1} = -u_{i,j-1} + u_{i-1,j} + u_{i+1,j}, \quad i, j = 1, 2, 3, 4 \quad (10.41)$$

To find the four estimates on the $j=2$ level, for example, we fix $j=1$ in Equation 10.41, vary $i=1,2,3,4$, and use the boundary values, as well as those obtained previously on the $j=1$ level. As a result, we find u_{12} , u_{22} , u_{32} and u_{42} . Continuing in this manner, estimates at all desired mesh points will be calculated; see Figure 10.16. These results can be readily verified by executing the user-defined function Wave1DFD.

```
>> t = 0:0.4:2; x = 0:0.4:2;
>> u0 = 5.*sin(pi*x/2); ut0 = zeros(length(x),1);
>> u = Wave1DFD(t,x,u0,ut0,1)
```

```
u =
    0   -2.9389   -4.7553   -4.7553   -2.9389    0
    0   -2.3776   -3.8471   -3.8471   -2.3776    0
    0   -0.9082   -1.4695   -1.4695   -0.9082    0
    0    0.9082    1.4695    1.4695    0.9082    0
    0    2.3776    3.8471    3.8471    2.3776    0
    0    2.9389    4.7553    4.7553    2.9389    0
```

We can also use Wave1DFD to plot the variations of u versus x for fixed values of time. For plotting purposes, we will use a smaller increment of 0.1 for x , and only plot the displacement curves for $t=0,0.3,0.6,0.9$. Note that the increments h and k are constrained by the condition $(k\alpha/h)^2 \leq 1$.

```
>> t = 0:0.3:0.9; x = 0:0.1:2;
>> u0 = 5.*sin(pi*x/2); ut0 = zeros(length(x),1);
>> u = Wave1DFD(t,x,u0,ut0,1); plot(x,u) % Figure 10.17
```

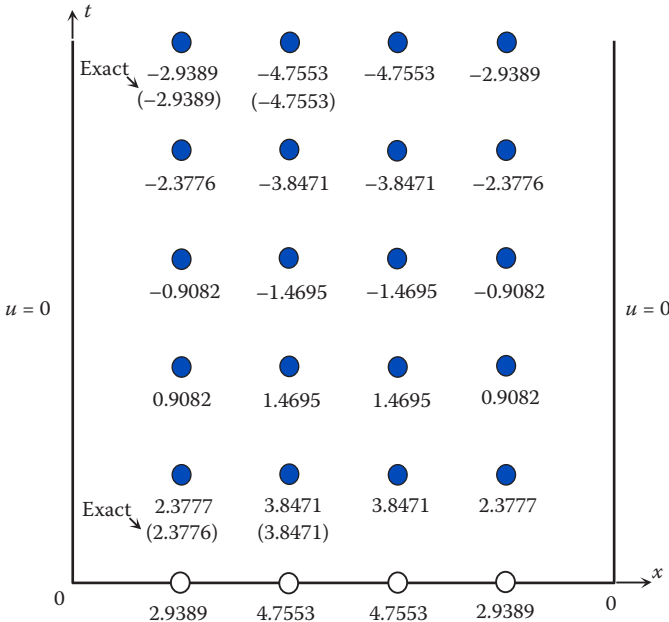


FIGURE 10.16

The grid and numerical results in Example 10.9.

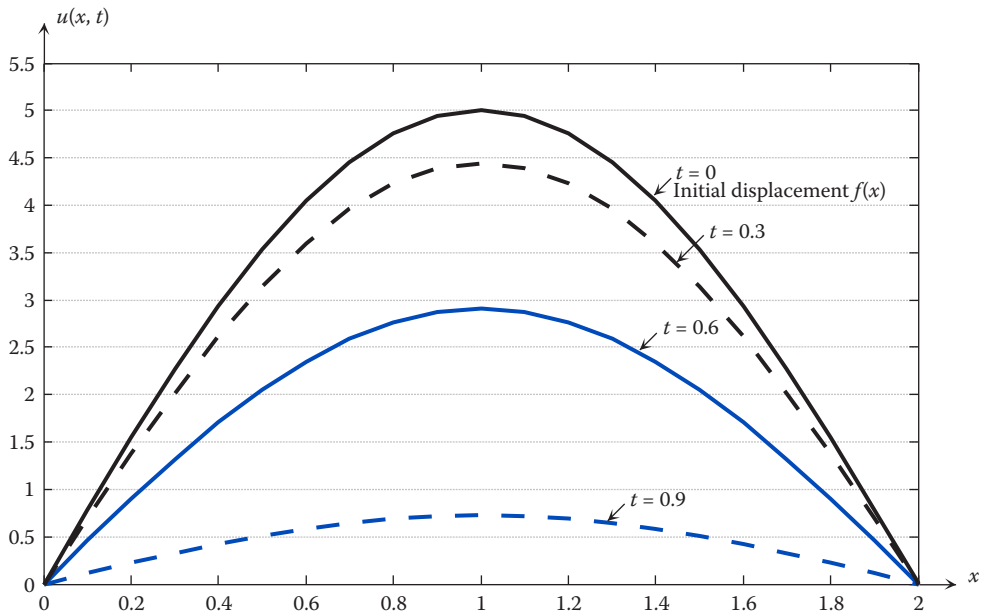


FIGURE 10.17




Displacement of the elastic string for various fixed values of time.

PROBLEM SET (CHAPTER 10)

Elliptic Partial Differential Equations (Section 10.2)

Dirichlet Problem (Laplace's Equation)

In Problems 1 through 4,

-  Solve the Dirichlet problem described in [Figure 10.18](#) with the specified boundary conditions.
-  Confirm the results of (a) by executing `DirichletPDE`. Suppress the plot.
-  Reduce the mesh size by half, resolve the problem using `DirichletPDE`, and find the approximate solutions at the original three interior mesh points.

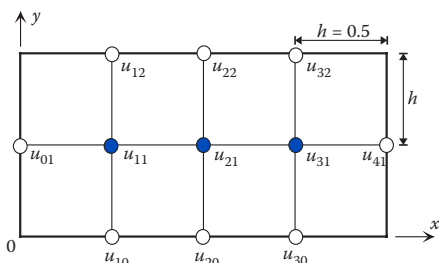
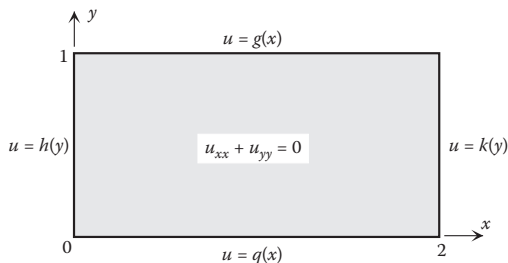


FIGURE 10.18

The Dirichlet problem in Problems 1 through 4.

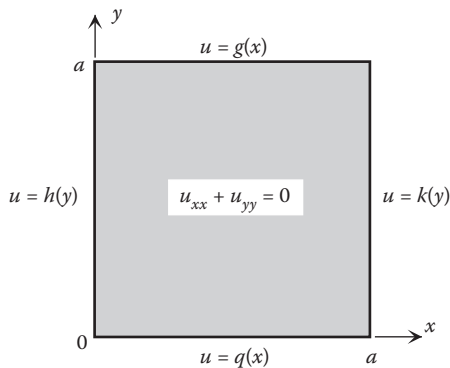





FIGURE 10.19

Region and grid used in Problems 5 through 8.




1. $q(x) = 120x(2-x)$, $g(x) = 0$, $h(y) = 0$, $k(y) = 0$
2. $q(x) = 4.7 \sin(\pi x/2)$, $g(x) = 3.8x(2-x)$, $h(y) = 0$, $k(y) = 0$
3. $q(x) = 3x(2+x)$, $g(x) = 0$, $h(y) = 0$, $k(y) = 24(1-y)$
4. $q(x) = 9$, $g(x) = 12$, $h(y) = 5y^2 - 2y + 9$, $k(y) = 2y^2 + y + 9$

In Problems 5 through 8,

- a.  Solve the Dirichlet problem described in [Figure 10.19](#) with the specified boundary conditions.
 - b.  Confirm the results of (a) by executing `DirichletPDE`. Suppress the plot.
 - c.  Reduce the mesh size by half, resolve the problem using `DirichletPDE`, and find the approximate solutions at the original four interior mesh points.
5. $a = 3$, $h = 1$, $q(x) = x$, $g(x) = 3$, $h(y) = y$, $k(y) = 3$
 6. $a = 9$, $h = 3$, $q(x) = 50 \sin^2(\pi x/9)$, $g(x) = x^2 + x$, $h(y) = 0$, $k(y) = y(y+1)$
 7. $a = 3$, $h = 1$, $q(x) = 80$, $g(x) = 158 - 17x - x^3$, $h(y) = 3y^3 - y + 80$, $k(y) = 80$
 8. $a = 9$, $h = 3$, $q(x) = 0$, $g(x) = 18 - 2x$, $h(y) = 18 \sin(\pi y/18)$, $k(y) = 20[\sin(\pi y/9) + 3 \sin(\pi y/3)]$

Dirichlet Problem (Poisson's Equation)

In Problems 9 through 14,

- a.  Solve Poisson's equation $u_{xx} + u_{yy} = f(x, y)$ in the specified region subject to the given boundary conditions.
 - b.  Confirm the results of (a) by executing `DirichletPDE`. Suppress the plot.
 - c.  Reduce the mesh size by half, resolve the problem using `DirichletPDE`, and find the approximate solutions at the original four interior mesh points.
9. $f(x, y) = (x-3)(2y-1)^2$, same region, grid, and boundary conditions as in Problem 5.

10. $f(x, y) = 10x^2 - 25y^3$, same region, grid, and boundary conditions as in Problem 7.
11. $f(x, y) = 0.1y \sin(\pi x/9)$ same region, grid, and boundary conditions as in Problem 6.
12. $f(x, y) = x + y$, same region, grid, and boundary conditions as in Problem 8.
13. $f(x, y) = y - x + 1$, same region and grid as in Figure 10.19 with $a = 1.2$ and boundary conditions described by




$$q(x) = x^2, \quad g(x) = 1.2 + 1.4x, \quad h(y) = y, \quad k(y) = 1.44 + y^2$$

14. $f(x, y) = \sin \pi x$, same region and grid as in Figure 10.19 with $a = 1.8$ and boundary conditions described by



$$q(x) = x, \quad g(x) = 1.8, \quad h(y) = y, \quad k(y) = 1.8$$


PRADI Method

15. Referring to Example 10.2,

- a.  Perform the second iteration to calculate the estimates $u_{11}^{(2)}, u_{21}^{(2)}, u_{12}^{(2)}, u_{22}^{(2)}$. Compare the corresponding relative errors with those at the completion of one iteration step.
- b.  Verify the results of (a) by executing the user-defined function PRADI.
- c.  Solve the Dirichlet problem by executing PRADI with default values for `tol` and `kmax`.

16. Consider the Dirichlet problem described in Figure 10.20.

- a.  Perform one full iteration step of the PRADI method using starting values of 0.4 for the interior mesh points. How would varying the starting values impact the estimates at the interior grid points?
- b.  Verify the numerical results of (a) by executing PRADI.

 In Problems 17 through 22, perform one complete iteration step of the PRADI method using the given starting values for the interior mesh points in the specified region and subject to the given boundary conditions.

17. $u_{xx} + u_{yy} = 0$ in the region, and with the boundary conditions, described in Problem 5; starting values of 5.

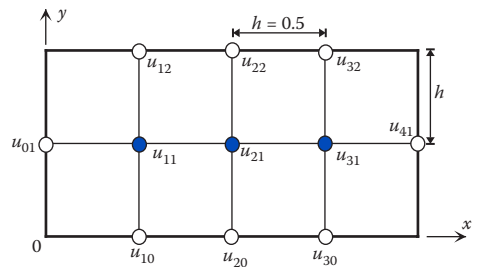
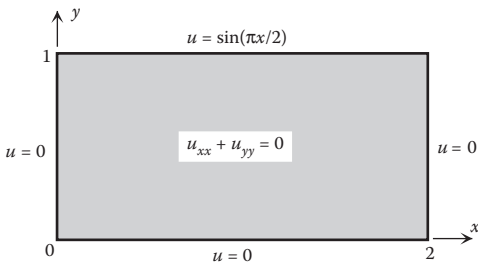



FIGURE 10.20

Region and grid in Problem 16.

18. $u_{xx} + u_{yy} = 0$ in the region, and with the boundary conditions, described in Problem 7; starting values of 80.
19. $u_{xx} + u_{yy} = 0$ in the region, and with the boundary conditions, described in Problem 6; starting values of 10.
20. $u_{xx} + u_{yy} = x + y$ in the region, and with the boundary conditions, described in Problem 8; starting values of 0.
21. Problem 13; starting values of 2.8.
22. Problem 14; starting values of 4.

Mixed Problem

 In Problems 23 through 28, solve the mixed problem in the region shown in [Figure 10.21](#) subject to the boundary conditions provided. In all cases, use a mesh size of $h = 1$.

23. $a = 3, b = 2$, Top BC : $\partial u / \partial y = 2x + 1$, Bottom BC : $u = 0$, Left BC : $u = y^2$, Right BC : $u = y^2 + 1$
24. $a = 3, b = 2$, Top BC : $u = x - 1$, Bottom BC : $u = x$, Left BC : $\partial u / \partial x = 0$, Right BC : $\partial u / \partial x = 2$
25. $a = 3, b = 2$, Top BC : $\partial u / \partial y = x$, Bottom BC : $u = 0$, Left BC : $u = 0$, Right BC : $u = 1.5y^2$
26. $a = 3, b = 2$, Top BC : $\partial u / \partial y = x^2$, Bottom BC : $u = 0$, Left BC : $u = 0$, Right BC : $u = y^3$
27. $a = 3, b = 3$, Top BC : $u = (x - 2)^2 + 1$, Bottom BC : $u = 5 - x$, Left BC : $u = 5$, Right BC : $\partial u / \partial x = 1$
28. $a = 3, b = 3$, Top BC : $\partial u / \partial y = 3x$, Bottom BC : $u = 3 - x$, Left BC : $u = 3$, Right BC : $u = 2y^2$

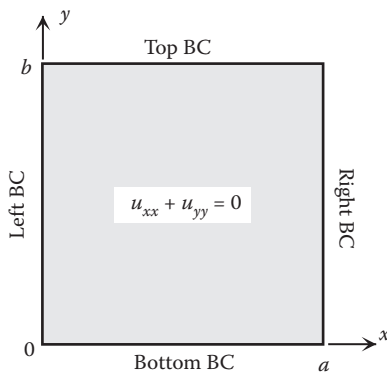


FIGURE 10.21

The region in Problems 23 through 28.

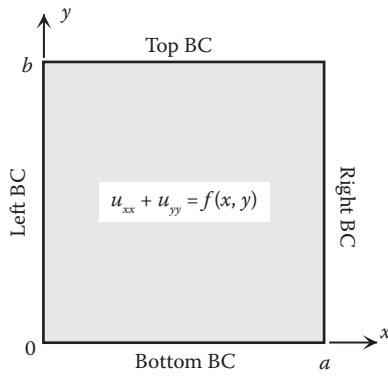



FIGURE 10.22

The region in Problems 29 and 30.

 In Problems 29 and 30, solve the mixed problem in the region shown in [Figure 10.22](#) subject to the boundary conditions provided. In both cases, use a mesh size of $h = 1$.





29. $a = 2, b = 3$, Top BC : $u = x^2 + 9$, Bottom BC : $u = 3x$, Left BC : $u = y^2$, Right BC : $\partial u / \partial x = 0.1(4 - y)^3$

$$f(x, y) = \frac{1}{3}x^2 + \frac{1}{2}y^2$$

30. $a = 3, b = 2$, Top BC : $\partial u / \partial y = 2x - 1$, Bottom BC : $u = 1$, Left BC : $u = 1$, Right BC : $\partial u / \partial x = 2y$


$$f(x, y) = 0.2xy$$

More Complex Regions

31.  Solve $u_{xx} + u_{yy} = 0$ in the region shown in [Figure 10.23](#) subject to the indicated boundary conditions. The curved portion of boundary obeys $x^2 + y^2 = 41$.
32.  Solve $u_{xx} + u_{yy} = 0$ in the region shown in [Figure 10.24](#) subject to the indicated boundary conditions. The slanted portion of boundary obeys $y + x = 8$.
33.  Solve $u_{xx} + u_{yy} = 0.5x^2y$ in the region shown in [Figure 10.24](#) subject to the indicated boundary conditions. The slanted portion of boundary obeys $y + x = 8$.
34.  Solve $u_{xx} + u_{yy} = 0$ in the region shown in [Figure 10.25](#) subject to the indicated boundary conditions. The curved portion of boundary obeys $(x - 7)^2 + (y - 9)^2 = 16$.

Parabolic Partial Differential Equations (Section 10.3)

Finite-Difference Method

35. Consider a laterally insulated wire of length $L = 2$ and $\alpha = 1$, whose ends are kept at zero temperature, subjected to initial temperature $f(x) = 10 \sin(\pi x / 2)$.
- a.  Using the finite-difference method, find the estimated values of temperature, $u(x, t)$, $0 \leq x \leq 2, 0 \leq t \leq 0.16$, at the mesh points generated by $h = 0.4$

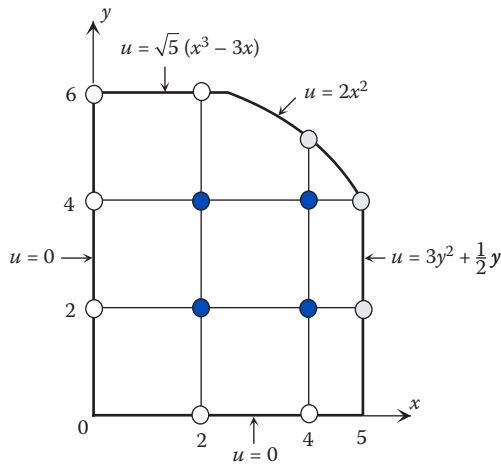


FIGURE 10.23
The region in Problem 31.

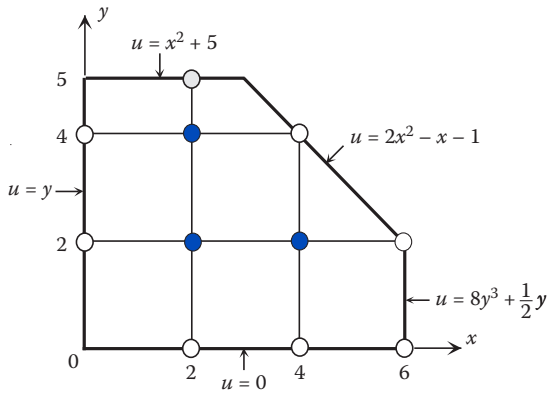



FIGURE 10.24
The region in Problem 32.

and $k = 0.04$. Also calculate the maximum % relative error at each time level. The exact solution is given by $u_{\text{Exact}}(x, t) = 10 \sin(\pi x/2) e^{-\pi^2 t/4}$.

- b.  Confirm the temperature estimates at the mesh points reported in (a) by executing Heat1DFD.

36. Consider a laterally insulated wire of length $L = 2$ and $\alpha = 1$, whose ends are kept at zero temperature, subjected to initial temperature

$$f(x) = \begin{cases} x & \text{if } 0 \leq x \leq 1 \\ 2 - x & \text{if } 1 \leq x \leq 2 \end{cases}$$

- a.  Using the finite-difference method, compute the estimated values of temperature, $u(x, t)$, $0 \leq x \leq 2$, $0 \leq t \leq 0.5$, at the mesh points generated by $h = 0.50$,

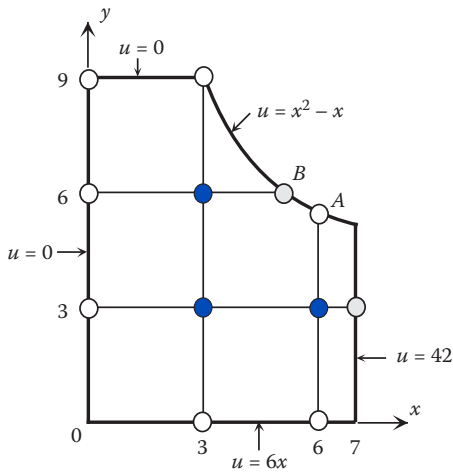



FIGURE 10.25

The region in Problem 34.



$k = 0.125$. Also calculate the maximum % relative error at each time level. The exact solution is given by $u_{\text{Exact}}(x, t) \equiv (8/\pi^2)\sin(\pi x/2) e^{-\pi^2 t/4}$.

- b. Confirm the temperature estimates at the mesh points reported in (a) by executing `Heat1DFD`.
37. Reconsider the wire in Problem 36. Assuming all information is unchanged except for $k = 0.0625$, calculate the temperature estimate $u(0.5, 0.125)$, and compare the corresponding relative error with that obtained in Problem 36.
38. Reconsider the wire in Problem 35. Assuming all information is unchanged except for $k = 0.02$, calculate the temperature estimate $u(0.4, 0.04)$, and compare the corresponding relative error with that obtained in Problem 35.
39. Consider a laterally insulated wire of length $L = 1$ and $\alpha = 1$, whose ends are kept at zero temperature, subjected to initial temperature $f(x) = x(1 - x)$.
 - a. Using the finite-difference method, find the estimated values of temperature, $u(x, t)$, $0 \leq x \leq 1$, $0 \leq t \leq 0.04$, at the mesh points generated by $h = 0.2$ and $k = 0.01$.
 - b. Confirm the numerical results of (a) using `Heat1DFD`.
40. Write a user-defined function with function call `u = Heat1DFD_gen(t, x, u0, alpha, q, g)` that uses the finite-difference method to solve the 1D heat equation subject to general boundary conditions. All arguments are as in `Heat1DFD`, while the two new parameters `q` and `g` are anonymous functions respectively representing $u(0, t)$ and $u(L, t)$. Using `Heat1DFD_gen` find the temperature estimates for a laterally insulated wire of length $L = 1$ and $\alpha = 1$, subjected to initial temperature $f(x) = x(1 - x)$ and boundary conditions $u(0, t) = 0$, $u(1, t) = t$, $0 < t < 0.05$. Construct a grid using $h = 0.2$, $k = 0.01$.
41. Using the user-defined function `Heat1DFD_gen` (see Problem 40) find the temperature estimates for a laterally insulated wire of length $L = 3$ and $\alpha = 0.8$,






subjected to initial temperature $f(x) = \sin \pi x$ and boundary conditions $u(0, t) = 1$, $u(3, t) = 1$, $0 < t < 0.3$. Construct a grid using $h = 0.3$, $k = 0.05$.




42.  Write a user-defined function with function call `u = Heat1DFD_insul_ends(t,x,u0,alpha)` that uses the finite-difference method to solve the 1D heat equation subject to insulated ends, that is, $u_x(0, t) = 0$ and $u_x(L, t) = 0$. Note that the difference equation must be applied at the boundary mesh points as well since the estimates at these points are now part of the unknown vector. Use central-difference formulns to approximate the first partial derivative at these points. All arguments are as in `Heat1DFD`. Using `Heat1DFD_insul_ends` find the temperature estimates for a laterally insulated wire of length $L = 1$ and $\alpha = 1$ subjected to initial temperature $f(x) = x(1 - x)$ and insulated ends. Construct a grid using $h = 0.2$, $k = 0.01$, and assume $0 < t < 0.05$.

Crank–Nicolson Method

43. Consider a laterally insulated wire of length $L = 1$ and $\alpha = 0.5$, whose ends are kept at zero temperature, subjected to initial temperature $f(x) = \sin \pi x + \sin 2\pi x$.
-  Using the Crank–Nicolson method, compute the estimated values of temperature, $u(x, t)$, $0 \leq x \leq 1$, $0 \leq t \leq 0.25$, at the mesh points generated by $h = 0.25$ and $k = 0.125$.
 -  Confirm the numerical results of (a) using `Heat1DCN`.
44. Consider a laterally insulated wire of length $L = 2$ and $\alpha = 1$, whose ends are kept at zero temperature, subjected to initial temperature



$$f(x) = \begin{cases} x & \text{if } 0 \leq x \leq 1 \\ 2 - x & \text{if } 1 \leq x \leq 2 \end{cases}$$

-  Using the Crank–Nicolson method, compute the estimated values of temperature, $u(x, t)$, $0 \leq x \leq 2$, $0 \leq t \leq 0.25$, at the mesh points generated by $h = 0.50$ and $k = 0.125$. Calculate the maximum % relative error at each time level, and compare with those corresponding to the finite-difference method (Problem 36). The exact solution is given in closed form as $u_{\text{Exact}}(x, t) \cong (8/\pi^2) \sin(\pi x/2) e^{-\pi^2 t/4}$.
 -  Confirm the temperature estimates at the mesh points reported in (a) by executing `Heat1DCN`.
45. Consider a laterally insulated wire of length $L = 3$ and $\alpha = 1.5$, whose ends are kept at zero temperature, subjected to initial temperature $f(x) = \sin(\pi x/3) + \sin 2\pi x$.
-  Using the Crank–Nicolson method, compute the estimated values of temperature, $u(x, t)$, $0 \leq x \leq 3$, $0 \leq t \leq 0.25$, at the mesh points generated by $h = 0.75$ and $k = 0.125$.
 -  Confirm the numerical results of (a) using `Heat1DCN`.
46. Repeat Problem 45 for an initial temperature of $f(x) = 0.5x(3 - x)$.
47.  Consider a laterally insulated wire of length $L = 1$ and $\alpha = 0.7071$, with ends kept at zero temperature, subjected to initial temperature $f(x) = 1 - \cos 2\pi x$. Let $h = 0.25$.

- a. Find the temperature estimates, $u(x, t)$, $0 \leq x \leq 1$, $0 \leq t \leq 0.5$, using `Heat1DCN` with $r = 1$.
- b. Adjust the time step size so that $r = 0.5$, and apply `Heat1DFD` to find the estimates at the mesh points. Compare the numerical results by the two methods at the time level $t = 0.125$.
48.  Consider a laterally insulated wire of length $L = 0.5$ and $\alpha = 1$, with ends kept at zero temperature, subjected to initial temperature $f(x) = x(1 - 2x)$. Let $h = 0.1$.
- a. Find the temperature estimates, $u(x, t)$, $0 \leq x \leq 0.5$, $0 \leq t \leq 0.04$, using the `Heat1DCN` with $k = 0.01$.
- b. Reduce the time step size by half and apply `Heat1DFD` to find the estimates at the mesh points. Compare the numerical results by the two methods at the time level $t = 0.01$.
49.  Consider a laterally insulated wire of length $L = 2$ and $\alpha = 1$, subjected to initial temperature $f(x) = 3x(2 - x)$ and boundary conditions $u(0, t) = 1$, $u(2, t) = t$, $0 < t < 0.25$. Construct a grid using $h = 0.5$, $k = 0.125$. Find the temperature estimates at the mesh points using the Crank–Nicolson method.
50.  Write a user-defined function with function call `u = Heat1DCN_gen(t, x, u0, alpha, q, g)` that uses the Crank–Nicolson method to solve the 1D heat equation subject to general boundary conditions. All arguments are as in `Heat1DCN`, while the two new parameters `q` and `g` are anonymous functions respectively representing $u(0, t)$ and $u(L, t)$. Using `Heat1DCN_gen` find the temperature estimates at the mesh points for the problem formulated in Problem 49.

Hyperbolic Partial Differential Equations (Section 10.4)

In Problems 51 through 56, an elastic string of length L with constant α , fixed at both ends, is considered. The string is subject to initial displacement $f(x)$ and initial velocity $g(x)$.

- a.  Using the finite-difference method with the indicated mesh sizes h and k , find the estimates for displacement $u(x, t)$ for $0 \leq x \leq L$ and the given time interval.
- b.  Confirm the numerical results of (a) using `Wave1DFD`.
51. $L = 0.5$, $\alpha = 1$, $f(x) = 10x(1 - 2x)$, $g(x) = 0$, $h = 0.1 = k$, $0 \leq t \leq 0.3$
52. $L = 1$, $\alpha = 1$, $f(x) = 6 \sin 2\pi x$, $g(x) = 0$, $h = 0.2 = k$, $0 \leq t \leq 0.4$
53. $L = 1$, $\alpha = 4$, $f(x) = 0$, $g(x) = 90x^2$, $h = 0.2$, $k = 0.05$, $0 \leq t \leq 0.15$
54. $L = 0.5$, $\alpha = 1$, $f(x) = 0$, $g(x) = -\sin(2\pi x)$, $h = 0.1 = k$, $0 \leq t \leq 0.2$
55. $L = 1$, $\alpha = 2$, $f(x) = x(1 - x)$, $g(x) = 10x$, $h = 0.2$, $k = 0.1$, $0 \leq t \leq 0.3$
56. $L = 1$, $\alpha = 2$, $f(x) = x(1 - x)$, $g(x) = \sin \pi x$, $h = 0.2$, $k = 0.1$, $0 \leq t \leq 0.3$

Index

A

- ABM, *see* Adams–Bashforth–Moulton
- ABM4PredCorr function, 328
- Absolute error, 15; *see also* Errors and approximations
- AdamsBashforth4 function, 325
- Adams–Bashforth method, 323–325
- Adams–Bashforth–Moulton (ABM), 327; *see also* Predictor–corrector methods
- Adams–Moulton method, 325–326
- ADI methods, *see* Alternating direction implicit methods
- Algorithm, 12, 19, 395–396
 - Cooley–Tukey algorithm, 216, 219–220
 - in fzero function, 61
 - and its code, 19–20
 - for Power method, 395–397
 - Sande–Tukey algorithm, 217–219
- Alternating direction implicit methods (ADI methods), 428–429; *see also* Elliptic partial differential equations
- Anonymous functions, 38

B

- BackSub function, 100, 101
- Back substitution, 103
- Best fit criterion, 163–164
- Binary and hexadecimal numbers, 13
- Bisection method, 55–60; *see also* Numerical solution of equations of single variable
- Boundary–initial–value problem, 440
- Boundary points, 424
- Boundary–value problem (BVP), 301, 367
 - boundary conditions, 367–368
 - example, 368–374, 375–378, 379–381, 384–386
 - finite-difference method, 374–378, 387–388
 - higher-order, 368
 - MATLAB built-in function `bvp4c` for, 381, 390–391
 - with mixed boundary conditions, 379
 - problem set, 386–391
 - second-order, 367, 382–384
 - shooting method, 368–374, 386–387

- Bracketing methods, 55; *see also* Numerical solution of equations of single variable philosophy of, 57
- BVP, *see* Boundary–value problem

C

- Cholesky factorization, 112; *see also* LU factorization methods
 - CholeskyFactor function, 112, 113
 - CholeskyMethod function, 113–114
 - example, 114–115
 - MATLAB built-in functions `lu` and `chol`, 115–116
 - operations count, 115
 - problem set, 151–152
- Cholesky’s method to solve linear system, 113–115
- Circ function, 48
- CN method, *see* Crank–Nicolson method
- Complex regions, 437–439
- Composite Simpson’s
 - 1/3 rule, 269, 270
 - 3/8 rule, 272–273
- Computational error, 133
 - sources, 12–13
- Condition number, 131
- Connecting line equation, 266
- Consecutive elements, 81
- Convergence, 67
 - rate of fixed-point iteration, 71–72
- Cooley–Tukey algorithm, 216, 219–220; *see also* Fourier approximation/interpolation
- Cramer’s rule, 7–8; *see also* Matrix analysis
- Crank–Nicolson method (CN method), 443; *see also* Parabolic partial differential equations
 - difference equation for 1D heat equation, 444
 - example, 445–446
 - vs. finite-difference method, 446–448
 - Heat1DCN function, 444–445
 - problem set, 459–460
 - six-point molecule used in, 444
- Cubic least-squares regression, 176–178
- Cubic splines, 198; *see also* Spline interpolation
 - clamped boundary conditions, 199
 - construction of cubic splines, 199–205

Cubic splines (*Continued*)
example, 202–204, 204–205
free boundary conditions, 199
10th-degree interpolating polynomial
and, 193
using MATLAB built-in functions, 208

Curve fitting, 161; *see also* [Interpolation](#)

D

Decimal notation, 13

Deflation methods, 403; *see also* [Matrix eigenvalue problem](#)
deflation process, 405
example, 405–407
problem set, 419–421
Wielandt’s deflation method, 403,
404–405

Derivative estimates for non-evenly spaced
data, 259

Determinant of matrix, 6

DFT, *see* [Discrete Fourier transform](#)

Difference equation for
Laplace’s equation, 425
1D heat equation, 441, 444

Differential equations, 1, 44; *see also* [Errors and approximations](#); [Iterative methods](#); [Matrix analysis](#); [Matrix eigenvalue problem](#)
example, 2, 3, 4
groups, 1
homogeneous solution, 2–3
linear, first-order ODEs, 1
method of undetermined coefficients, 3
particular solution, 3
problem set, 22
second-order ODEs with constant
coefficients, 2

Dirichlet problem, 424; *see also* [Elliptic partial differential equations](#)
boundary points, 424
difference equation for Laplace’s equation,
425
DirichletPDE function, 425–426
example, 427–428
grid for rectangular region, and five-point
molecule, 425
grid lines, 424
mesh points, 424
problem set, 452–454

Discrete Fourier transform (DFT), 215–216

Dominant eigenvalue, 395
estimation, 393

Doolittle factorization, 107; *see also* [LU factorization methods](#)
DoolittleMethod function, 110, 111
Doolittle’s method to solve linear system,
110–112
example, 108, 111–112
finding L and U, 108–110
operations count, 112

E

Editor Window, 47

Eigenvalue nearest specified value estimation,
399

Elementary row operations (EROs), 96

Elliptic partial differential equations, 424;
see also [Partial differential equation](#)
ADI methods, 428–429
complex regions, 437–439
Dirichlet problem, 424–428
mixed problem, 436–437
Neumann problem, 433–436
PRADI method, 429–433
problem set, 452–456
region with irregular boundary, 437

Equally-spaced data, 190–191

Equations with several roots, 83; *see also*
[Numerical solution of equations of single variable](#)
example, 84, 85–88
finding roots to right of specified point, 83
finding several roots in interval using
fzero, 84
Nzeros function, 83–84
problem set, 93–94

EROs, *see* [Elementary row operations](#)

Error
analysis, 305
associated with composite trapezoidal and
Simpson’s rules, 275
total, 173, 174

Error estimate
for composite Simpson’s 1/3 rule, 270
for composite Simpson’s 3/8 rule, 273
transmission from source to final result, 16–17

Errors and approximations, 12; *see also*
[Differential equations](#); [Iterative methods](#); [Matrix analysis](#); [Matrix eigenvalue problem](#)
absolute and relative errors, 15
algorithm, 12
binary and hexadecimal numbers, 13
decimal notation, 13

- error bound, 16
- example, 14–15, 15, 16, 18–19
- floating point and rounding errors, 13–14
- problem set, 24–25
- round-off, 14–15
- sources of computational error, 12–13
- subtraction of nearly equal numbers, 17–19
- transmission of error from source to final result, 16–17
- truncation error, 12

EulerODESystem function, 333

Euler's implicit method, 341–342

Euler's method, 302, 341; *see also* Numerical solution of initial-value problems

- calculation of local and global truncation errors, 305–307

- error analysis for, 305

- EulerODE function, 303

- example, 303–305, 306–307

- higher-order Taylor methods, 307–309, 351–352

- problem set, 350–352

- second-order Taylor method, 308–309

- for systems, 332–335

Explicit method, 323

Exponential function, 167

F

Fast Fourier transform (FFT), 216

FD method, *see* Finite-difference method

FFT, *see* Fast Fourier transform

Fifth-order accurate estimate, 321

Fifth-order Runge–Kutta method (RK5 method), 319–320; *see also* Runge–Kutta methods

Finite-difference formulas, 249; *see also* Numerical differentiation

- example, 253–254, 255

- for first derivative, 250

- for first, second, third, and fourth derivatives, 257–258

- for first to fourth derivatives, 256

- problem set, 286–290

- second derivative, 254

- three-point backward difference formula, 252–253, 254

- three-point central difference formula, 255

- three-point forward difference formula, 253, 254–255

- two-point backward difference formula, 250–251

- two-point central difference formula, 251–252

- two-point finite differences to approximate first derivative, 252

- two-point forward difference formula, 251

Finite-difference method (FD method), 374–378, 387–388, 440; *see also* Parabolic partial differential equations

- boundary-initial-value problem, 440

- vs. Crank–Nicolson method, 446–448

- difference equation for 1D heat equation, 441

- example, 442–443

- Heat1DFD function, 441–442

- problem set, 456–459

- region and grid used for solving 1D heat equation, 440

- stability and convergence of, 441

First-order initial-value problem, 301

Fitting interface in MATLAB, 208

FixedPoint functions, 67–68

Fixed-point iteration convergence, 66–67

Fixed-point iteration method; *see also* Numerical solution of systems of equations

- convergence of, 143

- example, 144–146

- problem set, 157–160

- for system of nonlinear equations, 143

Fixed-point method, 65; *see also* Numerical solution of equations of single variable

- convergence of fixed-point iteration, 66–67

- example, 68–71

- FixedPoint function, 67–68

- fixed point of iteration function, 65

- monotone convergence, 66

- note on convergence, 67

- oscillatory convergence, 66

- problem set, 90–92

- rate of convergence of fixed-point iteration, 71–72

- suitable iteration function selection, 66–67

Floating point and rounding errors, 13–14

Fourier approximation/interpolation, 209; *see also* Interpolation

- case study, 218–219

- Cooley–Tukey algorithm, 216, 219–220

- discrete Fourier transform, 215–216

- example, 212–215, 221–223

- fast Fourier transform, 216

- fft function, 220–223

- linear transformation of data, 210

Fourier approximation/interpolation

(Continued)

problem set, 245–248

Sande–Tukey algorithm, 216, 217–218

sinusoidal curve fitting, 209–210

TrigPoly function, 211–212

Fourth-order Runge–Kutta methods (RK4 methods), 316; *see also* [Runge–Kutta methods](#)

classical, 317

example, 318–319

increment function, 316

RK4 function, 318

Function values

at endpoints, 195

at interior knots, 196

fzero function, 60–61, 84

G

Gauss elimination method, 96; *see also* [Numerical solution of systems of equations](#)

BackSub function, 100, 101

back substitution, 103

counting number of operations, 102

elimination, 102–103

example, 97–98, 99, 102, 104

GaussPivotScale function, 100, 101

MATLAB built-in function “\”, 106

partial pivoting with row scaling, 98–99

permutation matrices, 99

problem set, 146–149

Thomas method, 104–106

ThomasMethod function, 105–106

tridiagonal systems, 103

Gaussian quadrature, 280; *see also* [Numerical integration](#)

example, 283–285

improved integral estimate, 280

integral estimate by trapezoidal rule, 280

linear transformation of data, 281

problem set, 297–299

weights and nodes used in, 283

GaussPivotScale function, 100, 101

GaussSeidel function, 127–128

Gauss–Seidel iteration method, 125–127
convergence, 127–130

General iterative method, 120

convergence, 120–121

Given’s method, 407

Grid lines, 424

H

Heat1DCN function, 444–445

Heat1DFD function, 441–442

Hessenberg matrix, 407

HeunODE function, 312–313

HeunODESystem function, 335

Heun’s method, 311; *see also* [Second-order Runge–Kutta methods](#)

graphical representation of, 312–313

HeunODE function, 312–313

Heun’s predictor–corrector method, 327

Heun’s RK3 method, 315

predictor–corrector method, 327

for systems, 335–336

Higher-order Runge–Kutta methods, 319–320

Higher-order Taylor methods, 307–309, 351–352

Householder function, 410

HouseholderQR function, 414

Householder’s tridiagonalization method, 407, 408; *see also* [Matrix eigenvalue problem](#)
determination of symmetric orthogonal \mathbf{P}_k , 409

example, 410–411

Householder function, 410

problem set, 421–422

symmetric matrices, 408–409

Hyperbolic partial differential equations, 448;
see also [Partial differential equation](#)

example, 450–452

1D wave equation, 448

problem set, 460

procedure, 449

region and grid used to solve 1D wave equation, 449

Wave1DFD function, 449, 450

I

if Command, 42–43

Ill-conditioning and error analysis, 131; *see also* [Numerical solution of systems of equations](#)

computational error, 133

condition number, 131

consequences of ill-conditioning, 135

effects of parameter change on solution, 136–138

example, 131–132, 137–138

ill-conditioning, 132

indicators of ill-conditioning, 133

problem set, 155

Implicit methods, 323

- Improper integrals, 285–286
 - problem set, 299–300
 - Improved Euler’s method, 311
 - Improved integral estimate, 280
 - Increment function, 315, 316
 - Indirect methods vs. direct methods for large systems, 130–131
 - Initial-value problem (IVP), 2, 301; *see also*
 - Numerical solution of initial-value problems
 - example, 346–347, 348–349, 349–350
 - MATLAB built-in functions, 345
 - non-stiff equations, 345
 - ODE solvers, 345, 347–348
 - problem set, 363–365
 - single first-order IVP, 345
 - stiff equations, 349–350
 - system of first-order, 348–349
 - Integral estimate by trapezoidal rule, 280
 - Integrand, 262
 - Interactive curve fitting and interpolation in MATLAB, 208
 - Interpolation, 161; *see also* Fourier approximation/interpolation; Linear regression; Polynomial interpolation; Polynomial regression; Spline interpolation
 - least-squares regression, 161
 - problem set, 223
 - using fft, 220–223
 - Inverse, 398–399
 - of matrix, 8–9
 - Iteration function selection, 66–67
 - Iterative methods, 19; *see also* Differential equations; Errors and approximations; Matrix analysis; Matrix eigenvalue problem
 - algorithms, 19
 - example, 19–20, 21, 22
 - fundamental, 20–21
 - problem set, 25
 - rate of convergence of, 21–22
 - Iterative solution of linear systems, 116; *see also*
 - Numerical solution of systems of equations
 - convergence of Jacobi iteration method, 122–125
 - example, 117–118, 118–119, 124–125, 128–130
 - GaussSeidel function, 127–128
 - Gauss–Seidel iteration method, 125–127
 - Gauss–Seidel iteration method convergence, 127–130
 - general iterative method, 120
 - general iterative method convergence, 120–121
 - indirect methods vs. direct methods for large systems, 130–131
 - Jacobi function, 123–124
 - Jacobi iteration method, 121–122
 - matrix norms, 118–119
 - problem set, 152–155
 - vector and matrix norms compatibility, 119–120
 - vector norms, 116
 - IVP, *see* Initial-value problem
- ## J
- Jacobi, 123–124
 - Jacobian matrix, 139
 - Jacobi iteration method, 121–122
 - convergence, 122–125
 - Jacobi’s method, 407
- ## K
- Knots, 194
- ## L
- Lagrange coefficient functions, 180
 - LagrangeInterp, 181
 - Lagrange interpolating polynomials, 180–183
 - Lagrange interpolation drawbacks, 183–184
 - Laplace’s equation, difference equation for, 425
 - Least-squares regression, 161
 - Linear fits with same error, 164
 - Linearization of nonlinear data, 167; *see also*
 - Interpolation
 - for curve fitting, 168
 - example, 168–172
 - exponential function, 167
 - power function, 167
 - problem set, 227–230
 - saturation function, 168
 - Linear regression, 162; *see also* Interpolation
 - best fit criterion, 163–164
 - example, 165–167
 - linear fit of data and individual errors, 162
 - linear function, 162
 - linear least-squares regression, 164
 - LinearRegression function, 165
 - problem set, 223–227
 - two linear fits with same total error, 164
 - zero total error based on criterion, 163

Linear; *see also* [Spline interpolation](#)

damping force, [249](#)

first-order ODEs, [1](#)

fit of data, [162](#)

function, [162](#)

second-order PDEs, [423](#)

splines, [194–195](#)

systems of equations, [95](#)

transformation of data, [210, 281](#)

Linspace, [30](#)

LU factorization methods, [107](#); *see also* [Cholesky](#)

[factorization](#); [Doolittle factorization](#);

[Numerical solution of systems of](#)

[equations](#)

finding L and U, [108](#)

problem set, [149–150](#)

M

MATLAB®, [27](#); *see also* [MATLAB vectors and](#)

[matrices](#); [Plotting](#); [Program flow](#)

[control](#); [Symbolic Math Toolbox](#)

built-in functions, [27](#)

differential equations, [44](#)

format options, [28](#)

formatted data, [43–44](#)

problem set, [51–54](#)

relational operators, [28](#)

rounding commands, [27, 28](#)

MATLAB built-in functions, [27, 345](#)

“\”, [106](#)

bvp4c for, [381, 390–391](#)

bvpinit, [383](#)

cond, [155](#)

cubic splines using, [208](#)

diff, [260–261, 290](#)

eig, [403](#)

fft, [220](#)

fzero, [60–61](#)

interp, [1](#) and spline, [205–207](#)

jacobian, [141](#)

lu and chol, [115–116](#)

norm, [117, 118, 119, 152](#)

Polyfit and Polyval, [178–179](#)

qr, [413](#)

quad and trapz, [273](#)

roots, [94](#)

to solve initial-value problems,

[345, 360](#)

MATLAB function, [38–39](#)

MATLAB vectors and matrices, [29](#); *see also*

[MATLAB®](#)

determinant, transpose, and inverse, [32](#)

diagonal matrices and diagonals of matrix,

[34–36](#)

element-by-element operations,

[33–34](#)

linspace function, [30](#)

matrices, [30–32](#)

slash operators, [33](#)

Matrix

determinant of, [6](#)

determination, [412](#)

diagonalization, [11](#)

diagonals of, [34–36](#)

generation, [404](#)

Hessenberg, [407](#)

inverse of, [8–9](#)

Jacobian, [139](#)

norms, [118–119](#)

special, [6](#)

symmetric, [408–409](#)

Matrix analysis, [4](#); *see also* [Differential](#)

[equations](#); [Errors and approximations](#);

[Iterative methods](#); [Matrix eigenvalue](#)

[problem](#)

Cramer’s rule, [7–8](#)

determinant of matrix, [6](#)

example, [6, 7–8, 8–9](#)

inverse of matrix, [8–9](#)

matrix operations, [5](#)

matrix transpose, [5](#)

problem set, [22–24](#)

properties of determinant, [6–7](#)

properties of inverse, [9](#)

solving linear system of equations, [9](#)

special matrices, [6](#)

Matrix eigenvalue problem, [393](#); *see also*

[Deflation methods](#); [Householder’s](#)

[tridiagonalization method](#); [Power](#)

[method](#); [QR factorization method](#);

[Shifted inverse power method](#)

eig function, [403](#)

example, [415–416](#)

Given’s method, [407](#)

Hessenberg matrix, [407](#)

HouseholderQR function, [414](#)

Jacobi’s method, [407](#)

problem set, [418–422](#)

qr function, [413](#)

terminating condition used in

[HouseholderQR function](#),

[414–416](#)

transformation to Hessenberg form,

[417–418](#)

Matrix eigenvalue problem, 9; *see also*
[Differential equations](#); [Errors and approximations](#); [Iterative methods](#); [Matrix analysis](#)

eigenvalue properties of matrices, 12

example, 10–11

matrix diagonalization, 11

problem set, 24

similarity transformation, 11

solving, 10–11

Mesh points, 424

Method of false position, *see* [Regula falsi method](#)

Method of undetermined coefficients, 3

M file functions and scripts, 47; *see also*
[MATLAB®](#)

CircFUNCTION, 48

Editor Window, 47

nargin command, 49

script file creation, 50–51

setting default values for input variables, 49–50

Mixed problem, 436–437; *see also* [Elliptic partial differential equations](#)

problem set, 455–456

Monotone convergence, 66

Multiple plots, 46–47

Multistep methods, 322; *see also* [Numerical solution of initial-value problems](#); [Predictor–corrector methods](#)

AdamsBashforth4 function, 325

Adams–Bashforth method, 323–325

Adams–Moulton method, 325–326

advantages, 322

explicit method, 323

implicit methods, 323

problem set, 355–356

N

nargin command, 49

Neumann problem, 433; *see also* [Elliptic partial differential equations](#)

example, 435–436

existence of solution for, 435

Newton–Cotes formulas, 262; *see also* [Numerical integration](#)

Newton divided-difference interpolating polynomials, 184–190

Newton forward-difference interpolating polynomials, 191–193

NewtonInterp, 187

Newton interpolation; *see also* [Interpolation](#)

problem set, 236–241

NewtonMod, 78–79

Newton–Raphson method, *see* [Newton’s method](#)

Newton’s method, 72; *see also* [Numerical solution of equations of single variable](#)

convergence of, 142

example, 74–76, 77–78, 79–80, 140–141

geometry of, 73

Jacobian matrix, 139

modified, 78–80

NewtonMod, 78–79

notes on, 77

problem set, 92–93, 157–160

rate of convergence of, 76

solving system of n nonlinear equations, 142

for system of nonlinear equations, 138–142

Non-stiff equations, 345

Not-a-knot condition, 207

Notation, 330

for partial derivatives, 423

Numerical differentiation, 249; *see also* [Finite-difference formulas](#); [Numerical integration](#); [Richardson’s extrapolation](#)

derivative estimates for non-evenly spaced data, 259

finite-difference formulas for, 249

MATLAB built-in functions diff and polyder, 260–261

problem set, 286–290

Numerical integration, 261; *see also*
[Gaussian quadrature](#); [Numerical differentiation](#); [Rectangular rule](#); [Romberg integration](#); [Simpson’s rules](#); [Trapezoidal rule](#)

of analytical functions, 275

improper integrals, 285–286

integrand, 262

Newton–Cotes formulas, 262

problem set, 290

quad, 273–274

trapz, 273–274

Numerical methods, 12; *see also* [Errors and approximations](#); [Iterative methods](#)

Numerical solution of equations of single variable, 55; *see also* [Equations with several roots](#); [Fixed-point method](#); [Newton’s method](#); [Regula falsi method](#); [Secant method](#)

bisection method, 55–60

bracketing methods, 55, 57

Numerical solution of equations of single variable (*Continued*)

classification of methods, 56

example, 59–60

fzero function, 60–61

numerical solution of equations, 55

open methods, 58

problem set, 88–94

Numerical solution of initial-value problems, 301; *see also* Euler's method; Initial-value problem; Multistep methods; Numerical stability; Runge–Kutta methods; Systems of ordinary differential equations

classification of, 302

example, 344–345

first-order initial-value problem, 301

one-step methods, 301–302

problem set, 350

stiff differential equations, 343–345

Numerical solution of linear systems, 96

Numerical solution of system of first-order ODEs, 332

Numerical solution of systems of equations, 95; *see also* Fixed-point iteration method; Gauss elimination method; Ill-conditioning and error analysis; Iterative solution of linear systems; LU factorization methods; Newton's method

classification of methods to solve linear system of equations, 96

linear systems of equations, 95

numerical solution of linear systems, 96

problem set, 146

systems of nonlinear equations, 138

Numerical stability, 340; *see also* Numerical solution of initial-value problems

Euler's implicit method, 341–342

Euler's method, 341

example, 342–343

problem set, 362–363

Nzeros, 83–84

O

ODE, *see* Ordinary differential equations

1D heat equation

difference equation for, 441, 444

solving, 440

One-dimensional heat equation, 424; *see also* Partial differential equation

One-dimensional wave equation, 424; *see also* Partial differential equation

1D wave equation, 448

solving, 449

One-step methods, 301–302

Open methods, 58; *see also* Numerical solution of equations of single variable

Operations count, 112, 115

Ordinary differential equations (ODE), 1, 345; *see also* Differential equations; Partial differential equation

with constant coefficients, 3

fourth-order, 368, 391

to handle systems of first-order initial-value, 348

linear first-order, 1

n th-order, 1, 368

solvers, 345, 347

as standard form of Equation, 2

Oscillatory convergence, 66

P

Parabolic partial differential equations, 440; *see also* Crank–Nicolson method; Finite-difference method; Partial differential equation

Partial differential equation (PDE), 1, 423; *see also* Differential equations;

Elliptic partial differential equations;

Hyperbolic partial differential equations; Ordinary differential equations; Parabolic partial

differential equations

linear, second-order, 423

notations for partial derivatives, 423

numerical solution of, 423

one-dimensional wave equation, 424

problem set, 452–460

two-dimensional Laplace's equation, 424

two-dimensional Poisson's equation, 424

Partial pivoting with row scaling, 98–99

Particular solution, 3

PDE, *see* Partial differential equation

Peaceman–Rachford alternating direction implicit method (PRADI method), 429; *see also* Elliptic partial differential equations

example, 432–433

PRADI function, 430–431

problem set, 454–455

Permutation matrices, 99
 P_k determination, symmetric orthogonal, 409
Plotting, 45; *see also* MATLAB®
analytical expressions, 46
function vs. variable, 45
multiple plots, 46–47
subplot, 45–46

Polynomial interpolation, 179; *see also*

Interpolation

divided differences table, 186
drawbacks of Lagrange interpolation, 183–184
equally-spaced data, 190–191
example, 187–190, 192–193
Lagrange coefficient functions, 180
LagrangeInterp function, 181
Lagrange interpolating polynomials, 180–183
Newton divided-difference interpolating polynomials, 184–190
Newton forward-difference interpolating polynomials, 191–193
NewtonInterp function, 187
problem set, 233–236

Polynomial regression, 172; *see also*

Interpolation

cubic least-squares regression, 176–178
example, 175–176, 177–178, 178–179
Polyfit and Polyval function, 178–179
problem set, 230–233
quadratic least-squares regression, 174–176
QuadraticRegression function, 174–175
total error, 173, 174

Power function, 167

Power method, 393; *see also* Matrix eigenvalue problem

algorithm for, 395–396
different cases of dominant eigenvalue, 395
estimation of dominant eigenvalue, 393
example, 396–397, 398–399, 401–402
inverse, 398–399
PowerMethod function, 396
problem set, 418–419
sequence of scalars, 394
shifted, 401
shifted inverse, 399–400

PRADI method, 429–433, *see* Peaceman–Rachford alternating direction implicit method

Predictor–corrector methods, 326; *see also*

Multistep methods

ABM4PredCorr function, 328
Adams–Bashforth–Moulton, 327

example, 328–330

Heun’s predictor–corrector method, 327
Program flow control, 41; *see also* MATLAB®
if Command, 42–43
for Loop, 41–42
while Loop, 43

Q

Q_k and R_k matrix determination, 412

QR factorization method, 411; *see also* Matrix eigenvalue problem

determination of Q_k and R_k matrices, 412
problem set, 421–422
structure of L_{kr} , 412–413

Quad, 273–274

Quadratic least-squares regression, 174–176

Quadratic splines, 195; *see also* Spline interpolation

example, 197–198

first derivatives at interior knots, 196
function values at endpoints, 195
function values at interior knots, 196
second derivative at left endpoint is zero, 196

R

Ralston’s method, 312; *see also* Second-order Runge–Kutta methods

Rectangular rule, 262; *see also* Numerical integration

composite, 262–264
error estimate for composite, 264
example, 265–266
problem set, 290–291

RegulaFalsi function, 62–63

Regula falsi method, 61; *see also* Numerical solution of equations of single variable

example, 63–64
modified, 64–65
problem set, 90
procedure, 62
user-defined RegulaFalsi function, 62–63

Relative errors, 15

Richardson’s extrapolation, 256, 275–278; *see also* Numerical differentiation; Romberg integration

for discrete sets of data, 259
example, 258

RK2 methods, *see* Second-order Runge–Kutta methods

RK3 methods, 315–316

RK4 method for systems, 318, 336–337, *see* [Fourth-order Runge–Kutta methods](#)
RK5 method, *see* [Fifth-order Runge–Kutta method](#)
RKF method, *see* [Runge–Kutta Fehlberg method](#)
Romberg, 278–279
Romberg integration, 275, 279; *see also* [Numerical integration](#); [Richardson’s extrapolation](#)
 errors associated with composite trapezoidal and Simpson’s rules, 275
 example, 277–278
 general formula, 278
 problem set, 297
 Richardson’s extrapolation, 275–278
 Romberg function, 278–279
 scheme, 279
Root of equation interpreted as fixed point of iteration function, 65
Rounding commands, 27, 28
Round-off, 14–15
Runge–Kutta Fehlberg method (RKF method), 320
 adjustment of step size, 321–322
 example, 322
 fifth-order accurate estimate, 321
Runge–Kutta methods, 309; *see also* [Fourth-order Runge–Kutta methods](#); [Numerical solution of initial-value problems](#); [Runge–Kutta Fehlberg method](#); [Second-order Runge–Kutta methods](#)
 higher-order Runge–Kutta methods, 319–320
 increment function, 315
 problem set, 352–355
 RK3 methods, 315–316

S

Sande–Tukey algorithm, 216, 217–218; *see also* [Fourier approximation/interpolation](#)
Saturation function, 168
Script file creation, 50–51
Secant method, 81; *see also* [Numerical solution of equations of single variable](#)
 consecutive elements, 81
 example, 82–83
 geometry of, 81
 notes on, 83
 problem set, 93
 rate of convergence of, 83
Secant function, 81–82

Second-order ODEs with constant coefficients, 2
Second-order Runge–Kutta methods (RK2 methods), 310; *see also* [Runge–Kutta methods](#)
 example, 314
 graphical representation of Heun’s method, 312–313
 Heun’s method, 311
 improved Euler’s method, 311
 Ralston’s method, 312
Second-order Taylor method, 308–309
Sequence of scalars, 394
Shifted inverse power method, 399–400; *see also* [Matrix eigenvalue problem](#)
 estimation of eigenvalue nearest specified value, 399
 notes on, 400
 ShiftInvPower function, 400
Shooting method, 368–374, 386–387
Similarity transformation, 11
Simpson’s rules, 269; *see also* [Numerical integration](#)
 error estimate for, 270, 273
 example, 271
 1/3 rule, 269
 problem set, 293–297
 3/8 rule, 271–272
Single first-order IVP, 345
Sinusoidal curve fitting, 209–210
Six-point molecule, 444
Slash operators, 33
Spline, 194
Spline interpolation, 161, 193; *see also* [Cubic splines](#); [Interpolation](#); [Quadratic splines](#)
 basic fitting interface in MATLAB, 208
 boundary conditions, 207
 example, 206
 interactive curve fitting and interpolation in MATLAB, 208
 knots, 194
 linear splines, 194–195
 MATLAB built-in functions interp, 1 and spline, 205–207
 no control over boundary conditions, 207
 not-a-knot condition, 207
 problem set, 241–245
State variables, 330
Step size adjustment, 321–322
Stiff differential equations, 343–345; *see also* [Numerical solution of initial-value problems](#)
 problem set, 363

Subtraction of nearly equal numbers, 17–19
Symbolic Math Toolbox, 36; *see also* MATLAB®
anonymous functions, 38
differentiation, 39–40
integration, 40–41
MATLAB function, 38–39
partial derivatives, 40
vpa command, 36
Symmetric matrices, 408–409
Symmetric orthogonal P_k determination, 409
Systems of nonlinear equations, 138
Systems of ordinary differential equations, 330;
see also Numerical solution of initial-value problems
classical RK4 method for systems, 336
EulerODESystem function, 333
Euler’s method for systems, 332–335
example, 331–332, 333–335, 337–340
HeunODESystem function, 335
Heun’s method for systems, 335–336
notation, 330
numerical solution of system of first-order ODEs, 332
problem set, 356–362
RK4System function, 336–337
state variables, 330
transformation into system of first-order ODEs, 330

T

10th-degree interpolating polynomial and, 193
Terminating condition used in HouseholderQR, 414–416
Thomas method, 104–106
ThomasMethod function, 105–106
Three-point
backward difference formula, 252–253, 254
central difference formula, 255
forward difference formula, 253, 254–255
Total error, 173, 174
Transformation into system of first-order ODEs, 330
Transformation to Hessenberg form, 417–418
Transmission of error from source to final result, 16–17

TrapComp, 268
Trapezoidal rule, 266; *see also* Numerical integration
composite, 267
equation of connecting line, 266
error estimate for composite, 267
example, 268
problem set, 292–293
TrapComp function, 268
Trapz, 273–274
Tridiagonal systems, 103
TrigPoly, 211–212
Truncation error, 12, 305–307; *see also* Errors and approximations
Two-dimensional Laplace’s equation, 424;
see also Partial differential equation
Two-dimensional Poisson’s equation, 424;
see also Partial differential equation
Two-point
backward difference formula, 250–251
central difference formula, 251–252
finite differences to approximate first derivative, 252
forward difference formula, 251

V

Vector and matrix norms, 119–120
Vector norms, 116
vpa command (variable precision arithmetic command), 36; *see also* Symbolic Math Toolbox

W

Wave1DFD, 449, 450
While Loop, 43
Wielandt’s deflation method, 403; *see also* Deflation methods
example, 405–406
matrix generated by, 404

Y

Zero total error based on criterion, 163