Explorations in Numerical Analysis

James V. Lambers and Amber C. Sumner

December 20, 2016

Copyright by

James V. Lambers and Amber C. Sumner

2016

Preface

This book evolved from lecture notes written by James Lambers and used in undergraduate numerical analysis courses at the University of California at Irvine, Stanford University and the University of Southern Mississippi. It is written for a year-long sequence of numerical analysis courses for either advanced undergraduate or beginning graduate students. Part II is suitable for a semester-long first course on numerical linear algebra.

The goal of this book is to introduce students to numerical analysis from both a theoretical and practical perspective, in such a way that these two perspectives reinforce each other. It is not assumed that the reader has prior programming experience. As mathematical concepts are introduced, code is used to illustrate them. As algorithms are developed from these concepts, the reader is invited to traverse the path from pseudocode to code.

Coding examples throughout the book are written in Matlab. Matlab has been a vital tool throughout the numerical analysis community since its creation thirty years ago, and its syntax that is oriented around vectors and matrices greatly accelerates the prototyping of algorithms compared to other programming environments.

The authors are indebted to the students in the authors' MAT 460/560 and 461/561 courses, taught in 2015-16, who were subjected to an early draft of this book.

J. V. Lambers
A. C. Sumner

Contents

I Preliminaries **1**

1 What is Numerical Analysis? **3**

1.1 Overview . . . 3
  1.1.1 Error Analysis . . . 4
  1.1.2 Systems of Linear Equations . . . 5
  1.1.3 Polynomial Interpolation and Approximation . . . 6
  1.1.4 Numerical Differentiation and Integration . . . 6
  1.1.5 Nonlinear Equations . . . 8
  1.1.6 Initial Value Problems . . . 8
  1.1.7 Boundary Value Problems . . . 9
1.2 Getting Started with MATLAB . . . 9
  1.2.1 Basic Mathematical Operations . . . 10
  1.2.2 Obtaining Help . . . 11
  1.2.3 Basic Matrix Operations . . . 11
  1.2.4 Storage of Variables . . . 12
  1.2.5 Complex Numbers . . . 12
  1.2.6 Creating Special Vectors and Matrices . . . 12
  1.2.7 Transpose Operators . . . 13
  1.2.8 `if` Statements . . . 14
  1.2.9 `for` Loops . . . 16
  1.2.10 `while` Loops . . . 17
  1.2.11 Function M-files . . . 18
  1.2.12 Graphics . . . 21
  1.2.13 Polynomial Functions . . . 22
  1.2.14 Number Formatting . . . 23
  1.2.15 Inline and Anonymous Functions . . . 23
  1.2.16 Saving and Loading Data . . . 24
  1.2.17 Using Octave . . . 24
1.3 How to Use This Book . . . 24
1.4 Error Analysis . . . 25
  1.4.1 Sources of Approximation . . . 25
  1.4.2 Error Measurement . . . 26
  1.4.3 Forward and Backward Error . . . 27
  1.4.4 Conditioning and Stability . . . 28

1.4.5 Convergence 30
1.5 Computer Arithmetic 34
1.5.1 Floating-Point Representation 35
1.5.2 Issues with Floating-Point Arithmetic 38
1.5.3 Loss of Precision 38

II Numerical Linear Algebra 43

2 Methods for Systems of Linear Equations 45
2.1 Triangular Systems 45
2.1.1 Upper Triangular Systems 45
2.1.2 Diagonal Systems 47
2.1.3 Lower Triangular Systems 48
2.2 Gaussian Elimination 48
2.2.1 Row Operations 48
2.2.2 The *LU* Factorization 53
2.2.3 Pivoting 61
2.3 Estimating and Improving Accuracy 64
2.3.1 The Condition Number 64
2.3.2 Iterative Refinement 66
2.3.3 Scaling and Equilibration 68
2.4 Special Matrices 69
2.4.1 Banded Matrices 69
2.4.2 Symmetric Matrices 71
2.4.3 Symmetric Positive Definite Matrices 73
2.5 Iterative Methods 78
2.5.1 Stationary Iterative Methods 79
2.5.2 Krylov Subspace Methods 83

3 Least Squares Problems 91
3.1 The Full Rank Least Squares Problem 91
3.1.1 Derivation of the Normal Equations 91
3.1.2 Solving the Normal Equations 94
3.1.3 The Condition Number of A^TA 94
3.2 The *QR* Factorization 95
3.2.1 Gram-Schmidt Orthogonalization 95
3.2.2 Classical Gram-Schmidt 96
3.2.3 Modified Gram-Schmidt 97
3.2.4 Householder Reflections 98
3.2.5 Givens Rotations 103
3.3 Rank-Deficient Least Squares 110
3.3.1 *QR* with Column Pivoting 110
3.3.2 Complete Orthogonal Decomposition 113
3.3.3 The Pseudo-Inverse 114

3.3.4 Perturbation Theory . . . 116
3.4 The Singular Value Decomposition . . . 117
3.4.1 Existence . . . 117
3.4.2 Properties . . . 118
3.4.3 Applications . . . 119
3.4.4 Minimum-norm least squares solution . . . 120
3.4.5 Closest Orthogonal Matrix . . . 120
3.4.6 Other Low-Rank Approximations . . . 121
3.5 Least Squares with Constraints . . . 121
3.5.1 Linear Constraints . . . 121
3.5.2 Quadratic Constraints . . . 123
3.6 Total Least Squares . . . 125

4 Eigenvalue Problems 127
4.1 Eigenvalues and Eigenvectors . . . 127
4.1.1 Definitions and Properties . . . 127
4.1.2 Decompositions . . . 128
4.1.3 Perturbation Theory . . . 132
4.2 Power Iterations . . . 136
4.2.1 The Power Method . . . 136
4.2.2 Orthogonal Iteration . . . 137
4.2.3 Inverse Iteration . . . 138
4.3 The QR Algorithm . . . 138
4.3.1 Hessenberg Reduction . . . 139
4.3.2 Shifted QR Iteration . . . 142
4.3.3 Computation of Eigenvectors . . . 145
4.4 The Symmetric Eigenvalue Problem . . . 145
4.4.1 Properties . . . 145
4.4.2 Perturbation Theory . . . 146
4.4.3 Rayleigh Quotient Iteration . . . 148
4.4.4 The Symmetric QR Algorithm . . . 149
4.5 The SVD Algorithm . . . 150
4.6 Jacobi Methods . . . 152
4.6.1 The Jacobi Idea . . . 152
4.6.2 The 2-by-2 Symmetric Schur Decomposition . . . 153
4.6.3 The Classical Jacobi Algorithm . . . 153
4.6.4 The Cyclic-by-Row Algorithm . . . 154
4.6.5 Error Analysis . . . 154
4.6.6 Parallel Jacobi . . . 154
4.6.7 Jacobi SVD Procedures . . . 155

III Data Fitting and Function Approximation 157

5 Polynomial Interpolation 159
- 5.1 Existence and Uniqueness . . . 159
- 5.2 Lagrange Interpolation . . . 161
- 5.3 Divided Differences . . . 164
  - 5.3.1 Newton Form . . . 166
  - 5.3.2 Computing the Newton Interpolating Polynomial . . . 167
  - 5.3.3 Equally Spaced Points . . . 174
- 5.4 Error Analysis . . . 176
  - 5.4.1 Error Estimation . . . 176
  - 5.4.2 Chebyshev Interpolation . . . 177
- 5.5 Osculatory Interpolation . . . 180
  - 5.5.1 Hermite Interpolation . . . 180
  - 5.5.2 Divided Differences . . . 181
- 5.6 Piecewise Polynomial Interpolation . . . 183
  - 5.6.1 Piecewise Linear Approximation . . . 184
  - 5.6.2 Cubic Spline Interpolation . . . 185

6 Approximation of Functions 193
- 6.1 Discrete Least Squares Approximations . . . 193
- 6.2 Continuous Least Squares Approximation . . . 201
  - 6.2.1 Orthogonal Polynomials . . . 203
  - 6.2.2 Construction of Orthogonal Polynomials . . . 204
  - 6.2.3 Legendre Polynomials . . . 206
  - 6.2.4 Chebyshev Polynomials . . . 208
  - 6.2.5 Error Analysis . . . 210
  - 6.2.6 Roots of Orthogonal Polynomials . . . 210
- 6.3 Rational Approximation . . . 211
  - 6.3.1 Continued Fraction Form . . . 214
  - 6.3.2 Chebyshev Rational Approximation . . . 215
- 6.4 Trigonometric Interpolation . . . 217
  - 6.4.1 Fourier Series . . . 217
  - 6.4.2 The Discrete Fourier Transform . . . 219
  - 6.4.3 The Fast Fourier Transform . . . 221
  - 6.4.4 Convergence and Gibbs' Phenomenon . . . 223

7 Differentiation and Integration 225
- 7.1 Numerical Differentiation . . . 225
  - 7.1.1 Taylor Series . . . 225
  - 7.1.2 Lagrange Interpolation . . . 227
  - 7.1.3 Higher-Order Derivatives . . . 230
  - 7.1.4 Sensitivity . . . 230
  - 7.1.5 Differentiation Matrices . . . 230
- 7.2 Numerical Integration . . . 232

7.2.1 Quadrature Rules 232
7.2.2 Interpolatory Quadrature 233
7.2.3 Sensitivity 234
7.3 Newton-Cotes Rules 234
7.3.1 Error Analysis 236
7.3.2 Higher-Order Rules 237
7.4 Composite Rules 238
7.4.1 Error Analysis 239
7.5 Gaussian Quadrature 241
7.5.1 Direct Construction 241
7.5.2 Orthogonal Polynomials 241
7.5.3 Error Analysis 243
7.5.4 Other Weight Functions 247
7.5.5 Prescribing Nodes 248
7.6 Extrapolation to the Limit 249
7.6.1 Richardson Extrapolation 250
7.6.2 The Euler-Maclaurin Expansion 251
7.6.3 Romberg Integration 253
7.7 Adaptive Quadrature 256
7.8 Multiple Integrals 262
7.8.1 Double Integrals 262
7.8.2 Higher Dimensions 266

IV Nonlinear and Differential Equations 269

8 Zeros of Nonlinear Functions 271
8.1 Nonlinear Equations in One Variable 271
8.1.1 Existence and Uniqueness 272
8.1.2 Sensitivity of Solutions 272
8.2 The Bisection Method 273
8.3 Fixed-Point Iteration 279
8.3.1 Successive Substitution 279
8.3.2 Convergence Analysis 282
8.3.3 Relaxation 285
8.4 Newton's Method and the Secant Method 287
8.4.1 Newton's Method 287
8.4.2 Convergence Analysis 291
8.4.3 The Secant Method 295
8.5 Convergence Acceleration 298
8.6 Systems of Nonlinear Equations 300
8.6.1 Fixed-Point Iteration 300
8.6.2 Newton's Method 303
8.6.3 Broyden's Method 305

9 Initial Value Problems 309

9.1 Existence and Uniqueness of Solutions . . . 311
9.2 One-Step Methods . . . 312
  9.2.1 Euler's Method . . . 312
  9.2.2 Solving IVPs in MATLAB . . . 314
  9.2.3 Runge-Kutta Methods . . . 315
  9.2.4 Implicit Methods . . . 317
9.3 Multistep Methods . . . 319
  9.3.1 Adams Methods . . . 319
  9.3.2 Predictor-Corrector Methods . . . 321
  9.3.3 Backward Differentiation Formulae . . . 322
9.4 Convergence Analysis . . . 322
  9.4.1 Consistency . . . 323
  9.4.2 Stability . . . 327
  9.4.3 Convergence . . . 329
  9.4.4 Stiff Differential Equations . . . 330
9.5 Adaptive Methods . . . 334
  9.5.1 Error Estimation . . . 335
  9.5.2 Adaptive Time-Stepping . . . 336
9.6 Higher-Order Equations and Systems of Differential Equations . . . 338
  9.6.1 Systems of First-Order Equations . . . 338
  9.6.2 Higher-Order Equations . . . 341

10 Two-Point Boundary Value Problems 343

10.1 The Shooting Method . . . 343
  10.1.1 Linear Problems . . . 344
  10.1.2 Nonlinear Problems . . . 344
10.2 Finite Difference Methods . . . 346
  10.2.1 Linear Problems . . . 347
  10.2.2 Nonlinear Problems . . . 349
10.3 Collocation . . . 353
10.4 The Finite Element Method . . . 358
10.5 Further Reading . . . 365

V Appendices 367

A Review of Calculus 369

A.1 Limits and Continuity . . . 369
  A.1.1 Limits . . . 369
  A.1.2 Limits of Functions of Several Variables . . . 371
  A.1.3 Limits at Infinity . . . 372
  A.1.4 Continuity . . . 372
  A.1.5 The Intermediate Value Theorem . . . 373
A.2 Derivatives . . . 374

A.2.1 Differentiability and Continuity . . . 375
A.3 Extreme Values . . . 375
A.4 Integrals . . . 377
A.5 The Mean Value Theorem . . . 378
A.5.1 The Mean Value Theorem for Integrals . . . 379
A.6 Taylor's Theorem . . . 380

B Review of Linear Algebra 385
B.1 Matrices . . . 385
B.2 Vector Spaces . . . 385
B.3 Subspaces . . . 387
B.4 Linear Independence and Bases . . . 387
B.5 Linear Transformations . . . 389
B.5.1 The Matrix of a Linear Transformation . . . 389
B.5.2 Matrix-Vector Multiplication . . . 390
B.5.3 Special Subspaces . . . 391
B.6 Matrix-Matrix Multiplication . . . 391
B.7 Other Fundamental Matrix Operations . . . 393
B.7.1 Vector Space Operations . . . 393
B.7.2 The Transpose of a Matrix . . . 393
B.7.3 Inner and Outer Products . . . 394
B.7.4 Hadamard Product . . . 395
B.7.5 Kronecker Product . . . 396
B.7.6 Partitioning . . . 396
B.8 Understanding Matrix-Matrix Multiplication . . . 396
B.8.1 The Identity Matrix . . . 397
B.8.2 The Inverse of a Matrix . . . 397
B.9 Triangular and Diagonal Matrices . . . 398
B.10 Determinants . . . 399
B.11 Vector and Matrix Norms . . . 400
B.11.1 Vector Norms . . . 400
B.11.2 Matrix Norms . . . 403
B.12 Function Spaces and Norms . . . 405
B.13 Inner Product Spaces . . . 407
B.14 Eigenvalues . . . 408
B.15 Differentiation of Matrices . . . 411

List of Figures

1.1 The dotted red curves demonstrate polynomial interpolation (left plot) and least-squares approximation (right plot) applied to $f(x) = 1/(1 + x^2)$ (blue solid curve). . 7

1.2 Screen shot of MATLAB at startup in Mac OS X 10

1.3 Figure for Exercise 1.2.5 22

5.1 The function $f(x) = 1/(1 + x^2)$ (solid curve) cannot be interpolated accurately on $[-5, 5]$ using a tenth-degree polynomial (dashed curve) with equally-spaced interpolation points. 178

5.2 Cubic spline that passing through the points $(0, 3)$, $(1/2, -4)$, $(1, 5)$, $(2, -6)$, and $(3, 7)$.190

6.1 Data points (x_i, y_i) (circles) and least-squares line (solid line) 196

6.2 Data points (x_i, y_i) (circles) and quadratic least-squares fit (solid curve) 199

6.3 Graphs of $f(x) = e^x$ (red dashed curve) and 4th-degree continuous least-squares polynomial approximation $f_4(x)$ on $[0, 5]$ (blue solid curve) 203

6.4 Graph of $\cos x$ (solid blue curve) and its continuous least-squares quadratic approximation (red dashed curve) on $(-\pi/2, \pi/2)$ 208

6.5 (a) Left plot: noisy signal (b) Right plot: discrete Fourier transform 221

6.6 Aliasing effect on noisy signal: coefficients $\hat{f}(\omega)$, for ω outside $(-63, 64)$, are added to coefficients inside this interval. 222

7.1 Graph of $f(x) = e^{3x} \sin 2x$ on $[0, \pi/4]$, with quadrature nodes from Example 7.7.2 shown on the graph and on the x-axis. 261

8.1 Left plot: Well-conditioned problem of solving $f(x) = 0$. $f'(x^*) = 24$, and an approximate solution $\hat{y} = f^{-1}(\epsilon)$ has small error relative to ϵ. Right plot: Ill-conditioned problem of solving $f(x) = 0$. $f'(x^*) = 0$, and $\hat{y}$ has large error relative to ϵ. 273

8.2 Illustrations of the Intermediate Value Theorem. Left plot: $f(x) = x - \cos x$ has a unique root on $[0, \pi/2]$. Right plot: $g(x) = e^x \cos(x^2)$ has multiple roots on $[0, \pi]$. . . 274

8.3 Because $f(\pi/4) > 0$, $f(x)$ has a root in $(0, \pi/4)$. 275

8.4 Progress of the Bisection method toward finding a root of $f(x) = x - \cos x$ on $(0, \pi/2)$277

8.5 Fixed-point Iteration applied to $g(x) = \cos x + 2$. 286

8.6 Approximating a root of $f(x) = x - \cos x$ using the tangent line of $f(x)$ at $x_0 = 1$. . 289

8.7 Newton's Method used to compute the reciprocal of 8 by solving the equation $f(x) = 8-1/x = 0$. When $x_0 = 0.1$, the tangent line of $f(x)$ at $(x_0, f(x_0))$ crosses the x-axis at $x_1 = 0.12$, which is close to the exact solution. When $x_0 = 1$, the tangent line crosses the x-axis at $x_1 = -6$, which causes searching to continue on the wrong portion of the graph, so the sequence of iterates does not converge to the correct solution. 292
8.8 Newton's Method applied to $f(x) = x^2 - 2$. The bold curve is the graph of f. The initial iterate x_0 is chosen to be 1. The tangent line of $f(x)$ at the point $(x_0, f(x_0))$ is used to approximate $f(x)$, and it crosses the x-axis at $x_1 = 1.5$, which is much closer to the exact solution than x_0. Then, the tangent line at $(x_1, f(x_1))$ is used to approximate $f(x)$, and it crosses the x-axis at $x_2 = 1.41\bar{6}$, which is already very close to the exact solution. 293

9.1 Solutions of $y' = -2ty$, $y(0) = 1$ on $[0, 1]$, computed using Euler's method and the fourth-order Runge-Kutta method 318

10.1 Left plot: exact (solid curve) and approximate (dashed curve with circles) solutions of the BVP (10.8) computed using finite differences. Right plot: error in the approximate solution. 349
10.2 Exact (solid curve) and approximate (dashed curve with circles) solutions of the BVP (10.11) from Example 10.2.2. 352
10.3 Exact (blue curve) and approximate (dashed curve) solutions of (10.18), (10.19) from Example 10.3.1. 356
10.4 Piecewise linear basis functions $\phi_j(x)$, as defined in (10.25), for $j = 1, 2, 3, 4$, with $N = 4$ 360
10.5 Exact (solid curve) and approximate (dashed curve) solutions of (10.22), (10.23) with $f(x) = x$ and $N = 4$ 364

List of Tables

6.1 Data points (x_i, y_i), for $i = 1, 2, \ldots, 10$, to be fit by a linear function 195
6.2 Data points (x_i, y_i), for $i = 1, 2, \ldots, 10$, to be fit by a quadratic function 198
6.3 Data points (x_i, y_i), for $i = 1, 2, \ldots, 5$, to be fit by an exponential function 200

Part I

Preliminaries

Chapter 1

What is Numerical Analysis?

1.1 Overview

This book provides a comprehensive introduction to the subject of *numerical analysis*, which is the study of the design, analysis, and implementation of numerical algorithms for solving mathematical problems that arise in science and engineering. These numerical algorithms differ from the analytical methods that are presented in other mathematics courses, in that they rely exclusively on the four basic arithmetic operations, addition, subtraction, multiplication and division, so that they can be implemented on a computer.

The goal in numerical analysis is to develop numerical methods that are effective, in terms of the following criteria:

- *A numerical method must be accurate.* While this seems like common sense, careful consideration must be given to the notion of accuracy. For a given problem, what level of accuracy is considered sufficient? As will be discussed in Section 1.4, there are many sources of error. As such, it is important to question whether it is prudent to expend resources to reduce one type of error, when another type of error is already more significant. This will be illustrated in Section 7.1.

- *A numerical method must be efficient.* Although computing power has been rapidly increasing in recent decades, this has resulted in expectations of solving larger-scale problems. Therefore, it is essential that numerical methods produce approximate solutions with as few arithmetic operations or data movements as possible. Efficiency is not only important in terms of time; memory is still a finite resource and therefore algorithms must also aim to minimize data storage needs.

- *A numerical method must be robust.* A method that is highly accurate and efficient for some (or even most) problems, but performs poorly on others, is unreliable and therefore not likely to be used in applications, even if any alternative is not as accurate and efficient. The user of a numerical method needs to know that the result produced can be trusted.

These criteria should be balanced according to the requirements of the application. For example, if less accuracy is acceptable, then greater efficiency can be achieved. This can be the case, for example, if there is so much uncertainty in the underlying mathematical model that there is no point in obtaining high accuracy.

1.1.1 Error Analysis

Numerical analysis is employed to develop algorithms for solving problems that arise in other areas of mathematics, such as calculus, linear algebra, or differential equations. Of course, these areas already include algorithms for solving such problems, but these algorithms are *analytical* methods. Examples of analytical methods are:

- Applying the Fundamental Theorem of Calculus to evaluate a definite integral,
- Using Gaussian elimination, with exact arithmetic, to solve a system of linear equations, and
- Using the method of undetermined coefficients to solve an inhomogeneous ordinary differential equation.

Such analytical methods have the benefit that they yield exact solutions, but the drawback is that they can only be applied to a limited range of problems. Numerical methods, on the other hand, can be applied to a much wider range of problems, but only yield approximate solutions. Fortunately, in many applications, one does not necessarily need very high accuracy, and even when such accuracy is required, it can still be obtained, if one is willing to expend the extra computational effort (or, really, have a computer do so).

Because solutions produced by numerical algorithms are not exact, we will begin our exploration of numerical analysis with one of its most fundamental concepts, which is *error analysis*. Numerical algorithms must not only be *efficient*, but they must also be *accurate*, and *robust*. In other words, the solutions they produce are at best *approximate* solutions because an exact solution cannot be computed by analytical techniques. Furthermore, these computed solutions should not be too sensitive to the input data, because if they are, any error in the input can result in a solution that is essentially useless. Such error can arise from many sources, such as

- neglecting components of a mathematical model or making simplifying assumptions in the model,
- discretization error, which arises from approximating continuous functions by sets of discrete data points,
- convergence error, which arises from truncating a sequence of approximations that is meant to converge to the exact solution, to make computation possible, and
- roundoff error, which is due to the fact that computers represent real numbers approximately, in a fixed amount of storage in memory.

We will see that in some cases, these errors can be surprisingly large, so one must be careful when designing and implementing numerical algorithms. Section 1.4 will introduce fundamental concepts of error analysis that will be used throughout this book, and Section 1.5 will discuss computer arithmetic and roundoff error in detail.

1.1.2 Systems of Linear Equations

Next, we will learn about how to solve a system of linear equations

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

which can be more conveniently written in matrix-vector form

$$A\mathbf{x} = \mathbf{b},$$

where A is an $n \times n$ matrix, because the system has n equations (corresponding to rows of A) and n unknowns (corresponding to columns).

To solve a general system with n equations and unknowns, we can use *Gaussian elimination* to reduce the system to *upper-triangular form*, which is easy to solve. In some cases, this process requires *pivoting*, which entails interchanging of rows or columns of the matrix A. Gaussian elimination with pivoting can be used not only to solve a system of equations, but also to compute the *inverse* of a matrix, even though this is not normally practical. It can also be used to efficiently compute the determinant of a matrix.

Gaussian elimination with pivoting can be viewed as a process of factorizing the matrix A. Specifically, it achieves the decomposition

$$PA = LU,$$

where P is a *permutation matrix* that describes any row interchanges, L is a *lower-triangular* matrix, and U is an upper-triangular matrix. This decomposition, called the *LU decomposition*, is particularly useful for solving $A\mathbf{x} = \mathbf{b}$ when the right-hand side vector $\mathbf{b}$ varies. We will see that for certain special types of matrices, such as those that arise in the normal equations, variations of the general approach to solving $A\mathbf{x} = \mathbf{b}$ can lead to improved efficiency.

Gaussian elimination and related methods are called *direct methods* for solving $A\mathbf{x} = \mathbf{b}$, because they compute the exact solution (up to roundoff error, which can be significant in some cases) in a fixed number of operations that depends on n. However, such methods are often not practical, especially when A is very large, or when it is *sparse*, meaning that most of its entries are equal to zero. Therefore, we also consider *iterative methods*. Two general classes of iterative methods are:

- *stationary iterative methods*, which can be viewed as fixed-point iterations, and rely primarily on *splittings* of A to obtain a system of equations that can be solved rapidly in each iteration, and

- *non-stationary methods*, which tend to rely on matrix-vector multiplication in each iteration and a judicious choice of *search direction* and *linesearch* to compute each iterate from the previous one.

We will also consider systems of equations, for which the number of equations, m, is greater than the number of unknowns, n. This is the *least-squares* problem, which is reduced to a system with n equations and unknowns,

$$A^T A\mathbf{x} = A^T\mathbf{b},$$

called the *normal equations*. While this system can be solved directly using methods discussed above, this can be problematic due to sensitivity to roundoff error. We therefore explore other approaches based on *orthogonalization* of the columns of A.

Another fundamental problem from linear algebra is the solution of the *eigenvalue problem*

$$A\mathbf{x} = \lambda\mathbf{x},$$

where the scalar λ is called an *eigenvalue* and the nonzero vector $\mathbf{x}$ is called an *eigenvector*. This problem has many applications throughout applied mathematics, including the solution of differential equations and statistics. We will see that the tools developed for efficient and robust solution of least squares problems are useful for the eigenvalue problem as well.

1.1.3 Polynomial Interpolation and Approximation

Polynomials are among the easiest functions to work with, because it is possible to evaluate them, as well as perform operations from calculus on them, with great efficiency. For this reason, more complicated functions, or functions that are represented only by values on a discrete set of points in their domain, are often approximated by polynomials.

Such an approximation can be computed in various ways. We first consider *interpolation*, in which we construct a polynomial that agrees with the given data at selected points. While interpolation methods are efficient, they must be used carefully, because it is not necessarily true that a polynomial that agrees with a given function at certain points is a good approximation to the function elsewhere in its domain.

One remedy for this is to use *piecewise* polynomial interpolation, in which a low-degree polynomial, typically linear or cubic, is used to approximate data only on a given subdomain, and these polynomial “pieces” are “glued” together to obtain a piecewise polynomial approximation. This approach is also efficient, and tends to be more robust than standard polynomial interpolation, but there are disadvantages, such as the fact that a piecewise polynomial only has very few derivatives.

An alternative to polynomial interpolation, whether piecewise or not, is polynomial approximation, in which the goal is to find a polynomial that, in some sense, best fits given data. For example, it is not possible to exactly fit a large number of points with a low-degree polynomial, but an approximate fit can be more useful than a polynomial that can fit the given data exactly but still fail to capture the overall behavior of the data. This is illustrated in Figure 1.1.

1.1.4 Numerical Differentiation and Integration

It is often necessary to approximate derivatives or integrals of functions that are represented only by values at a discrete set of points, thus making differentiation or integration rules impossible to use directly. Even when this is not the case, derivatives or integrals produced by differentiation or integration rules can often be very complicated functions, making their computation and evaluation computationally expensive.

While there are many software tools, such as Mathematica or Maple, that can compute derivatives of integrals symbolically using such rules, they are inherently unreliable because they require detection of patterns in whatever data structure is used to represent the function being differentiated or integrated, and it is not as easy to implement software that performs this kind of task effectively as it is for a person to learn how to do so through observation and intuition.

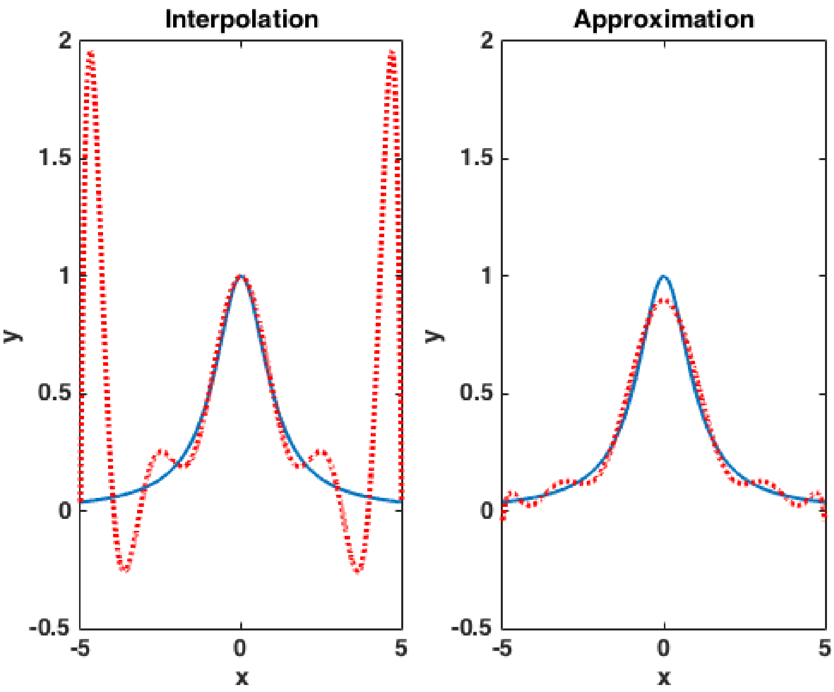


Figure 1.1: The dotted red curves demonstrate polynomial interpolation (left plot) and least-squares approximation (right plot) applied to $f(x) = 1/(1+x^2)$ (blue solid curve).

Therefore, it is important to have methods for evaluating derivatives and integrals that are insensitive to the complexity of the function being acted upon. Numerical techniques for these operations make use of polynomial interpolation by (implicitly) constructing a polynomial interpolant that fits the given data, and then applying differentiation or integration rules to the polynomial. We will see that by choosing the method of polynomial approximation judiciously, accurate results can be obtain with far greater efficiency than one might expect.

As an example, consider the definite integral

$$\int_0^1 \frac{1}{x^2 - 5x + 6}\,dx.$$

Evaluating this integral exactly entails factoring the denominator, which is simple in this case but not so in general, and then applying partial fraction decomposition to obtain an antiderivative, which is then evaluated at the limits. Alternatively, simply computing

$$\frac{1}{12}[f(0) + 4f(1/4) + 2f(1/2) + 4f(3/4) + f(1)],$$

where $f(x)$ is the integrand, yields an approximation with 0.01% error (that is, the error is 10^{-4}). While the former approach is less tedious to carry out by hand, at least if one has a calculator, clearly the latter approach is the far more practical use of computational resources.

1.1.5 Nonlinear Equations

The vast majority of equations, especially nonlinear equations, cannot be solved using analytical techniques such as algebraic manipulations or knowledge of trigonometric functions. For example, while the equations

$$x^2 - 5x + 6 = 0, \quad \cos x = \frac{1}{2}$$

can easily be solved to obtain exact solutions, these slightly different equations

$$x^2 - 5xe^x + 6 = 0, \quad x\cos x = \frac{1}{2}$$

cannot be solved using analytical methods.

Therefore, *iterative* methods must instead be used to obtain an approximate solution. We will study a variety of such methods, which have distinct advantages and disadvantages. For example, some methods are guaranteed to produce a solution under reasonable assumptions, but they might do so slowly. On the other hand, other methods may produce a sequence of iterates that quickly converges to the solution, but may be unreliable for some problems.

After learning how to solve nonlinear equations of the form $f(x) = 0$ using iterative methods such as Newton's method, we will learn how to generalize such methods to solve systems of nonlinear equations of the form $f(\mathbf{x}) = \mathbf{0}$, where $f : \mathbb{R}^n \to \mathbb{R}^n$. In particular, for Newton's method, computing $x_{n+1} - x_n = -f(x_n)/f'(x_n)$ in the single-variable case is generalized to solving the system of equations $J_f(\mathbf{x}_n)\mathbf{s}_n = -f(\mathbf{x}_n)$, where $J_f(\mathbf{x}_n)$ is the *Jacobian* of f evaluated at $\mathbf{x}_n$, and $\mathbf{s}_n = \mathbf{x}_{n+1} - \mathbf{x}_n$ is the *step* from each iterate to the next.

1.1.6 Initial Value Problems

Next, we study various algorithms for solving an *initial value problem*, which consists of an *ordinary differential equation*

$$\frac{dy}{dt} = f(t, y), \quad a \leq t \leq b,$$

an an *initial condition*

$$y(t_0) = \alpha.$$

Unlike analytical methods for solving such problems, that are used to find the exact solution in the form of a function $y(t)$, numerical methods typically compute values $y_1, y_2, y_3, \ldots$ that approximate $y(t)$ at discrete time values $t_1, t_2, t_3, \ldots$. At each time t_{n+1}, for $n > 0$, value of the solution is approximated using its values at previous times.

We will learn about two general classes of methods: *one-step* methods, which are derived using Taylor series and compute y_{n+1} only from y_n, and *multistep* methods, which are based on polynomial interpolation and compute y_{n+1} from $y_n, y_{n-1}, \ldots, y_{n-s+1}$, where s is the number of steps in the method. Either type of method can be *explicit*, in which y_{n+1} can be described in terms of an explicit formula, or *implicit*, in which y_{n+1} is described implicitly using an equation, normally nonlinear, that must be solved during each time step.

The difference between consecutive times t_n and t_{n+1}, called the *time step*, need not be uniform; we will learn about how it can be varied to achieve a desired level of accuracy as efficiently as possible. We will also learn about how the methods used for the first-order initial-value problem described above can be generalized to solve higher-order equations, as well as systems of equations.

One key issue with time-stepping methods is *stability*. If the time step is not chosen to be sufficiently small, the computed solution can grow without bound, even if the exact solution is bounded. Generally, the need for stability imposes a more severe restriction on the size of the time step for explicit methods, which is why implicit methods are commonly used, even though they tend to require more computational effort per time step. Certain systems of differential equations can require an extraordinarily small time step to be solved by explicit methods; such systems are said to be *stiff*.

1.1.7 Boundary Value Problems

We conclude with a discussion of solution methods for the *two-point boundary value problem*

$$y'' = f(x, y, y'), \quad a \leq x \leq b,$$

with *boundary conditions*

$$y(a) = \alpha, \quad y(b) = \beta.$$

One approach, called the *shooting method*, transforms this boundary-value problem into an initial-value problem so that methods for such problems can then be used. However, it is necessary to find the correct initial values so that the boundary condition at $x = b$ is satisfied. An alternative approach is to discretize y'' and y' using *finite differences*, the approximation schemes covered last semester, to obtain a system of equations to solve for an approximation of $y(x)$; this system can be linear or nonlinear. We conclude with the *Rayleigh-Ritz method*, which treats the boundary value problem as a continuous least-squares problem.

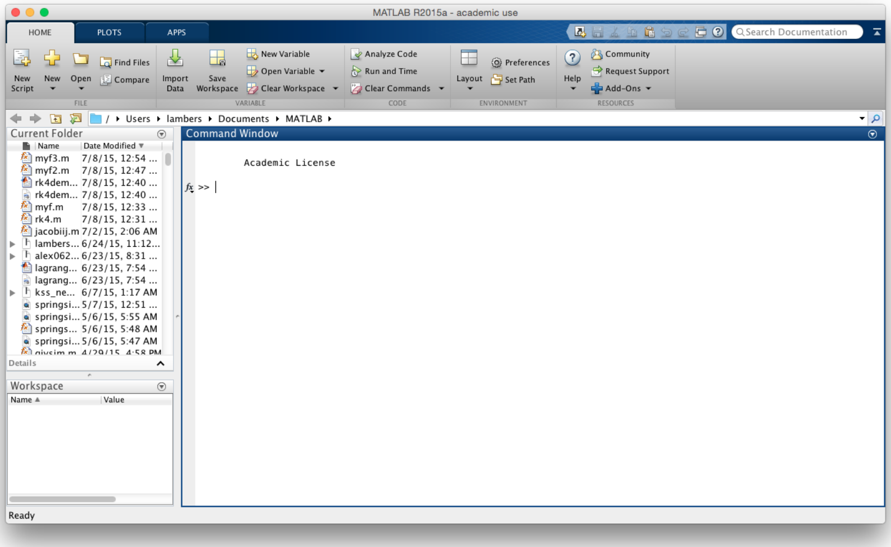
1.2 Getting Started with MATLAB

Matlab is commercial software, originally developed by Cleve Moler in 1982 [24] and currently sold by The Mathworks. It can be purchased and downloaded from `mathworks.com`. As of this writing, the student version can be obtained for $50, whereas academic and industrial licenses are much more expensive. For any license, “toolboxes” can be purchased in order to obtain additional functionality, but for the tasks performed in this book, the core product will be sufficient.

As an alternative, one can instead use Octave, a free application which uses the same programming language as Matlab, with only minor differences. It can be obtained from `gnu.org`. Its user interface is not as “friendly” as that of Matlab, but it has improved significantly in its most recent versions. In this book, examples will feature only Matlab, but the code will also work in Octave, without modification except where noted.

Figure 1.2 shows Matlab when it is launched. The large window on the right is the *command window*, in which commands can be entered interactively at the “$\gg$” prompt. On the left, the top window lists the files in the *current working directory*. By default, this directory is a subdirectory of your `Documents` (Mac OS X) or `My Documents` (Windows) directory that is named `MATLAB`. It is important to keep track of your current working directory, because that is the first place that Matlab looks for *M-files*, or files that contain Matlab code. These will be discussed in detail later in this section.

We will now present basic commands, operations and programming constructs in Matlab. Before we begin, we will use the `diary` command to save all subsequent commands, and their


MATLAB R2015a - academic use
HOME
PLOTS
APPS
Search Documentation
New Script
New
Open
Find Files
Compare
FILE
Import Data
Save Workspace
New Variable
Open Variable
Clear Workspace
VARIABLE
Analyze Code
Run and Time
Clear Commands
CODE
Layout
Preferences
Set Path
ENVIRONMENT
Help
Community
Request Support
Add-Ons
RESOURCES
/ ▸ Users ▸ lambers ▸ Documents ▸ MATLAB ▸
Current Folder
Details
Workspace
Command Window
Ready

Name	Date Modified
myf3.m	7/8/15, 12:54 ...
myf2.m	7/8/15, 12:47 ...
rk4dem...	7/8/15, 12:40 ...
rk4dem...	7/8/15, 12:40 ...
myf.m	7/8/15, 12:33 ...
rk4.m	7/8/15, 12:31 ...
jacobiij.m	7/2/15, 2:06 AM
lambers...	6/24/15, 11:12...
alex062...	6/23/15, 8:31 ...
lagrang...	6/23/15, 7:54 ...
lagrang...	6/23/15, 7:54 ...
kss_ne...	6/7/15, 1:17 AM
springsi...	5/7/15, 12:51 ...
springsi...	5/6/15, 5:55 AM
springs...	5/6/15, 5:48 AM
springsi...	5/6/15, 5:47 AM

Name	Value

```
        Academic License

fx >> |
```

Figure 1.2: Screen shot of MATLAB at startup in Mac OS X

output, to a text file. By default, this output is saved to a file that is named `diary` in the current working directory, but we will supply our own filename as an argument, to make the saved file easier to open later in a text editor.

```
>> diary tutorial.txt
```

1.2.1 Basic Mathematical Operations

Next, we will perform basic mathematical operations. Try out these commands, and observe the corresponding output. Once a command is typed at the prompt, simply hit the Enter key to execute the command.

```
>> a=3+4

a =

     7

>> b=sqrt(a)

b =

   2.645751311064591

>> c=exp(a)
```

```
c =

    1.096633158428459e+003
```

As can be seen from these statements, arithmetic operations and standard mathematical functions can readily be performed, so Matlab could be used as a "desk calculator" in which results of expressions can be stored in *variables*, such as `a`, `b` and `c` in the preceding example. Also, note that once a command is executed, the output displayed is the variable name, followed by its value. This is typical behavior in Matlab, so for the rest of this tutorial, the output will not be displayed in the text.

1.2.2 Obtaining Help

Naturally, when learning new software, it is important to be able to obtain help. Matlab includes the `help` command for this purpose. Try the following commands. You will observe that Matlab offers help on various categories of commands, functions or operators (hereafter referred to simply as "commands"), and also help pages on individual commands, such as `lu`.

```
>> help
>> help ops
>> help lu
```

1.2.3 Basic Matrix Operations

Matlab, which is short for "matrix laboratory", is particularly easy to use with vectors and matrices. We will now see this for ourselves by constructing and working with some simple matrices. Try the following commands.

```
>> A=[ 1 0; 0 2 ]
>> B=[ 5 7; 9 10 ]
>> A+B
>> 2*ans
>> C=A+B
>> 4*A
>> C=A-B
>> C=A*B
>> w=[ 4; 5; 6 ]
```

As we can see, matrix arithmetic is easily performed. However, what happens if we attempt an operation that, mathematically, does not make sense? Consider the following example, in which a 2×2 matrix is multiplied by a 3×1 vector.

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

Since this operation is invalid, Matlab does not attempt to perform the operation and instead displays an error message. The function name `mtimes` refers to the function that implements the matrix multiplication operator that is represented in the above command by `*`.

1.2.4 Storage of Variables

Let's examine the variables that we have created so far. The `whos` command is useful for this purpose.

```
>> whos
  Name      Size            Bytes  Class     Attributes

  A         2x2                32  double
  B         2x2                32  double
  C         2x2                32  double
  a         1x1                 8  double
  ans       2x2                32  double
  b         1x1                 8  double
  c         1x1                 8  double
  w         3x1                24  double
```

Note that each number, such as `a`, or each entry of a matrix, occupies 8 bytes of storage, which is the amount of memory allocated to a double-precision floating-point number. This system of representing real numbers will be discussed further in Section 1.5. Also, note the variable `ans`. It was not explicitly created by any of the commands that we have entered. It is a special variable that is assigned the most recent expression that is not already assigned to a variable. In this case, the value of `ans` is the output of the operation `4*A`, since that was not assigned to any variable.

1.2.5 Complex Numbers

MATLAB can also work with complex numbers. The following command creates a vector with one real element and one complex element.

```
>> z=[ 6; 3+4i ]
```

Now run the `whos` command again. Note that it states that `z` occupies 32 bytes, even though it has only two elements. This is because each element of `z` has a real part and an imaginary part, and each part occupies 8 bytes. It is important to note that if a single element of a vector or matrix is complex, then the entire vector or matrix is considered complex. This can result in wasted storage if imaginary parts are supposed to be zero, but in fact are small, nonzero numbers due to roundoff error (which will be discussed in Section 1.5).

The `real` and `imag` commands can be used to extract the real and imaginary parts, respectively, of a complex scalar, vector or matrix. The output of these commands are stored as real numbers.

```
>> y=real(z)
>> y=imag(z)
```

1.2.6 Creating Special Vectors and Matrices

It can be very tedious to create matrices entry-by-entry, as we have done so far. Fortunately, MATLAB has several functions that can be used to easily create certain matrices of interest. Try the following commands to learn what these functions do. In particular, note the behavior when only one argument is given, instead of two.

```
>> E=ones(6,5)
>> E=ones(3)
>> R=rand(3,2)
```

As the name suggests, `rand` creates a matrix with random entries. More precisely, the entries are random numbers that are uniformly distributed on $[0, 1]$.

Exercise 1.2.1 *What if we want the entries distributed within a different interval, such as* $[-1, 1]$*? Create such a matrix, of size* 3×2*, using matrix arithmetic that we have seen, and the* `ones` *function.*

In many situations, it is helpful to have a vector of equally spaced values. For example, if we want a vector consisting of the integers from 1 to 10, inclusive, we can create it using the statement

```
>> z=[ 1 2 3 4 5 6 7 8 9 10 ]
```

However, this can be very tedious if a vector with many more entries is needed. Imagine creating a vector with all of the integers from 1 to 1000! Fortunately, this can easily be accomplished using the *colon operator*. Try the following commands to see how this operator behaves.

```
>> z=1:10
>> z=1:2:10
>> z=10:-2:1
>> z=1:-2:10
```

It should be noted that the second argument, that determines spacing between entries, need not be an integer.

Exercise 1.2.2 *Use the colon operator to create a vector of real numbers between 0 and 1, inclusive, with spacing 0.01.*

1.2.7 Transpose Operators

We now know how to create *row* vectors with equally spaced values, but what if we would rather have a column vector? This is just one of many instances in which we need to be able to compute the *transpose* of a matrix in MATLAB. Fortunately, this is easily accomplished, using the single quote as an operator. For example, this statement

```
>> z=(0:0.1:1)'
```

has the desired effect. However, one should not simply conclude that the single quote is the transpose operator, or they could be in for an unpleasant surprise when working with complex-valued matrices. Try these commands to see why:

```
>> z=[ 6; 3+4i ]
>> z'
>> z.'
```

We can see that the single quote is an operator that takes the *Hermitian transpose* of a matrix A, commonly denoted by A^H: it is the transpose *and* complex conjugate of A. That is, $A^H = \overline{A^T}$. Meanwhile, the dot followed by the single quote is the transpose operator.

Either operator can be used to take the transpose for matrices with real entries, but one must be more careful when working with complex entries. That said, why is the "default" behavior, represented by the simpler single quote operator, the Hermitian transpose rather than the transpose? This is because in general, results or techniques established for real matrices, that make use of the transpose, do not generalize to the complex case unless the Hermitian transpose is used instead.

1.2.8 `if` Statements

Now, we will learn some essential programming constructs, that MATLAB shares with many other programming languages. The first is an `if` statement, which is used to perform a different task based on the result of a given conditional expression, that is either true or false.

At this point, we will also learn how to write a *script* in MATLAB. Scripts are very useful for the following reasons:

- Some MATLAB statements, such as the programming constructs we are about to discuss, are quite complicated and span several lines. Typing them at the command window prompt can be very tedious, and if a mistake is made, then the entire construct must be retyped.
- It frequently occurs that a sequence of commands needs to be executed several times, with no or minor changes. It can be very tedious and inefficient to repeatedly type out such command sequences several times, even if MATLAB's history features (such as using the arrow keys to scroll through past commands) are used.

A script can be written in a plain text file, called an *M-file*, which is a file that has a `.m` extension. An M-file can be written in any text editor, or in MATLAB's own built-in editor. To create a new M-file or edit an existing M-file, one can use the `edit` command at the prompt:

```
>> edit entermonthyear
```

If no extension is given, a `.m` extension is assumed. If the file does not exist in the current working directory, MATLAB will ask if the file should be created.

In the editor, type in the following code, that computes the number of days in a given month, while taking leap years into account.

```
% entermonthyear - script that asks the user to provide a month and year,
% and displays the number of days in that month

% Prompt user for month and year
month=input('Enter the month (1-12): ');
year=input('Enter the 4-digit year: ');
% For displaying the month by day, we construct an array of strings
% containing the names of all the months, in numerical order.
% This is a 12-by-9 matrix, since the longest month name (September)
% contains 9 letters.  Each row must contain the same number of columns, so
% other month names must be padded to 9 characters.
```

```
months=[ 'January  '; 'February '; 'March    '; 'April    '; 'May      '; 'June     '; 'July
% extract the name of the month indicated by the user
monthname=months(month,:);
% remove trailing blanks
monthname=deblank(monthname);
if month==2
    % month is February
    if rem(year,4)==0
        % leap year
        days=29;
    else
        % non-leap year
        days=28;
    end
elseif month==4 || month==6 || month==9 || month==11
    % "30 days hath April, June, September and November..."
    days=30;
else
    % "...and all the rest have 31"
    days=31;
end
% display number of days in the given month
disp([ monthname ', ' num2str(year) ' has ' num2str(days) ' days.' ])
```

Can you figure out how an `if` statement works, based on your knowledge of what the result should be?

Note that this M-file includes *comments*, which are preceded by a percent sign (%). Once a % is entered on a line, the rest of that line is ignored. This is very useful for documenting code so that a reader can understand what it is doing. The importance of documenting one's code cannot be overstated. In fact, it is good practice to write the documentation *before* the code, so that the process of writing code is informed with a clearer idea of the task at hand.

As this example demonstrates, `if` statements can be nested within one another. Also note the use of the keywords `else` and `elseif`. These are used to provide alternative conditions under which different code can be executed, if the original condition in the `if` statement turns out to be false. If any conditions paired with the `elseif` keyword also turn out to be false, then the code following the `else` keyword, if any, is executed.

This script features some new functions that can be useful in many situations:

- `deblank(s)`: returns a new string variable that is the same as `s`, except that any "white space" (spaces, tabs, or newlines) at the end of `s` is removed
- `rem(a,b)`: returns the remainder after dividing `a` by `b`
- `num2str(x)`: returns a string variable based on formatting the number `x` as text

To execute a script M-file, simply type the name of the file (without the `.m` extension) at the prompt.

```
>> entermonthyear
Enter the month (1-12): 5
Enter the 4-digit year: 2001
May, 2001 has 31 days.
```

Note that in the above script M-file, most of the statements are terminated with semicolons. Unlike in programming languages such as C++, the semicolon is not required. If it is omitted from a statement, then the value of any expression that is computed in that statement is displayed, along with its variable name (or `ans`, if there is no variable associated with the expression). Including the semicolon suppresses printing. In most cases, this is the desired behavior, because excessive output can make software less user-friendly. However, omitting semicolons can be useful when writing and debugging new code, because seeing intermediate results of a computation can expose bugs. Once the code is working, then semicolons can be added to suppress superfluous output.

1.2.9 `for` Loops

The next programming construct is the `for` loop, which is used to carry out an iterative process that runs for a predetermined number of iterations. To illustrate for loops, we examine the script file `gausselim.m`:

```
% gausselim - script that performs Gaussian elimination on a random 40-by-40
% matrix
m=40;
n=40;
% generate random matrix
A=rand(m,n);
% display it
disp('Before elimination:')
disp(A)
for j=1:n-1,
    % use elementary row operations to zero all elements in column j below
    % the diagonal
    for i=j+1:m,
        mult=A(i,j)/A(j,j);
        % subtract mult * row j from row i
        for k=j:n,
            A(i,k)=A(i,k)-mult*A(j,k);
        end
        % equivalent code:
        %A(i,j:n)=A(i,j:n)-mult*A(j,j:n);
    end
end
% display updated matrix
disp('After elimination:')
disp(A)
```

Note the syntax for a `for` statement: the keyword `for` is followed by a *loop variable*, such as `i`, `j` or `k` in this example, and that variable is assigned a value. Then the body of the loop is given, followed by the keyword `end`.

What does this loop actually do? During the nth iteration, the loop variable is set equal to the nth column of the expression that is assigned to it by the `for` statement. Then, the loop variable retains this value throughout the body of the loop (unless the loop variable is changed within the body of the loop, which is ill-advised, and sometimes done by mistake!), until the iteration is completed. Then, the loop variable is assigned the next column for the next iteration. In most cases, the loop variable is simply used as a counter, in which case assigning to it a row vector of values, created using the colon operator, yields the desired behavior.

Now run this script, just like in the previous example. The script displays a randomly generated matrix A, then performs Gaussian elimination on A to obtain an upper triangular matrix U, and then displays the final result U.

Exercise 1.2.3 *An upper trianglar matrix U has the property that $u_{ij} = 0$ whenever $i > j$; that is, the entire "lower triangle" of U, consisting of all entries below the main diagonal, must be zero. Examine the matrix U produced by the script* `gausselim` *above. Why are some subdiagonal entries nonzero?*

1.2.10 `while` Loops

Next, we introduce the `while` loop, which, like a `for` loop, is also used to implement an iterative process, but is controlled by a conditional expression rather than a predetermined set of values such as `1:n`. The following script, saved in the file `newtonsqrt.m`, illustrates the use of a `while` loop.

```
% newtonsqrt - script that uses Newton's method to compute the square root
% of 2

% choose initial iterate
x=1;
% announce what we are doing
disp('Computing the square root of 2...')
% iterate until convergence.  we will test for convergence inside the loop
% and use the break keyword to exit, so we can use a loop condition that's
% always true
while true
    % save previous iterate for convergence test
    oldx=x;
    % compute new iterate
    x=x/2+1/x;
    % display new iterate
    disp(x)
    % if relative difference in iterates is less than machine precision,
    % exit the loop
    if abs(x-oldx)<eps*abs(x)
        break;
    end
end
% display result and verify that it really is the square root of 2
disp('The square root of 2 is:')
x
disp('x^2 is:')
disp(x^2)
```

Note the use of the expression `true` in the `while` statement. The value of the predefined variable `true` is 1, while the value of `false` is 0, following the convention used in many programming languages that a nonzero number is interpreted as the boolean value “true”, while zero is interpreted as “false”.

A `while` loop runs as long as the condition in the `while` statement is true. It follows that this particular `while` statement is an infinite loop, since the value of `true` will never be false. However, this loop will exit when the condition in the enclosed `if` statement is true, due to the `break` statement. A `break` statement causes the enclosing `for` or `while` loop to immediately exit.

This particular `while` loop computes an approximation to $\sqrt{2}$ such that the *relative* difference between each new iterate `x` and the previous iterate `oldx` is less than the value of the predefined variable `eps`, which is informally known as “unit roundoff” or “machine precision”. This concept will be discussed further in Section 1.5.1. The actual process used to obtain this approximation to $\sqrt{2}$ is obtained using *Newton's method*, which will be discussed in Section 8.4.

Go ahead and run this script by typing its name, `newtonsqrt`, at the prompt. The code will display each iteration as the approximation is improved until it is sufficiently accurate. Note that convergence to $\sqrt{2}$ is quite rapid! This effect will be explored further in Chapter 8.

1.2.11 Function M-files

In addition to script M-files, MATLAB also uses *function M-files.* A function M-file is also a text file with a `.m` extension, but unlike a script M-file, that simply includes a sequence of commands that could have instead been typed at the prompt, the purpose of a function M-file is to extend the capability of MATLAB by defining a new function, that can then be used by other code.

The following function M-file, called `fahtocel.m`, illustrates function definition.

```
function tc=fahtocel(tf)
% converts function input 'tf' of temperatures in Fahrenheit to
% function output 'tc' of temperatures in Celsius
temp=tf-32;
tc=temp*5/9;
```

Note that a function definition begins with the keyword `function`; this is how MATLAB distinguishes a script M-file from a function M-file (though in a function M-file, comments can still precede `function`).

After the keyword `function`, the *output arguments* of the function are specified. In this function, there is only one output argument, `tc`, which represents the Celsius temperature. If there were more than one, then they would be enclosed in square brackets, and in a comma-separated list.

After the output arguments, there is a `=` sign, then the function name, which should match the name of the M-file aside from the `.m` extension. Finally, if there are any *input arguments*, then they are listed after the function name, separated by commas and enclosed in parentheses.

After this first line, all subsequent code is considered the body of the function–the statements that are executed when the function is called. The only exception is that other functions can be defined within a function M-file, but they are "helper" functions, that can only be called by code within the same M-file. Helper functions must appear *after* the function after which the M-file is named.

Type in the above code for `fahtocel` into a file `fahtocel.m` in the current working directory. Then, it can be executed as follows:

```
>> tc=fahtocel(212)
tc =
   100
```

If `tc=` had been omitted, then the output value 100 would have been assigned to the special variable `ans`, described in §1.2.4.

Note that the definition of `fahtocel` uses a variable `temp`. Here, it should be emphasized that all variables defined within a function, including input and output arguments, are only defined within the function itself. If a variable inside a function, such as `temp`, happens to have the same name as another variable defined in the top-level workspace (the memory used by variables defined outside of any function), or in another function, then this other variable is completely independent of the one that is internal to the function. Consider the following example:

```
>> temp=32
temp =
    32
>> tfreeze=fahtocel(temp)
tfreeze =
     0
>> temp
temp =
    32
```

Inside `fahtocel`, `temp` is set equal to zero by the subtraction of 32, but the `temp` in the top-level workspace retains its value of 32.

Comments included at the top of an M-file (whether script or function) are assumed by MATLAB to provide documentation of the M-file. As such, these comments are displayed by the `help` command, as applied to that function. Try the following commands:

```
>> help fahtocel
>> help newtonsqrt
```

We now illustrate other important aspects of functions, using the following M-file, which is called `vecangle2.m`:

```
% vecangle2 - function that computes the angle between two given vectors
% in both degrees and radians.  We use the formula
% x'*y = ||x||_2 ||y||_2 cos(theta), where theta is the angle between x and
% y
function [anglerad,angledeg]=vecangle2(x,y)
n=length(x);
if n~=length(y)
    error('vector lengths must agree')
end
% compute needed quantities for above formula
dotprod=x'*y;
xnorm=norm(x);
ynorm=norm(y);
% obtain cos(angle)
cosangle=dotprod/(xnorm*ynorm);
% use inverse cosine to obtain angle in radians
anglerad=acos(cosangle);
% if angle in degrees is desired (that is, two output arguments are
% specified), then convert to degrees.  Otherwise, don't bother
if nargout==2,
    angledeg=anglerad*180/pi;
end
```

As described in the comments, the purpose of this function is to compute the angle between two vectors in n-dimensional space, in both radians and degrees. Note that this function accepts multiple input arguments and multiple output arguments. The way in which this function is called is similar to how it is defined. For example, try this command:

```
>> [arad,adeg]=vecangle2(rand(5,1),rand(5,1))
```

It is important that code include error-checking, if it might be used by other people. To that end, the first task performed in this function is to check whether the input arguments `x` and `y` have the same length, using the `length` function that returns the number of elements of a vector or matrix. If they do not have the same length, then the `error` function is used to immediately exit the function `vecangle2` and display an informative error message.

Note the use of the variable `nargout` at the end of the function definition. The function `vecangle2` is defined to have two output arguments, but `nargout` is the number of output arguments that are actually specified when the function is called. Similarly, `nargin` is the number of input arguments that are specified.

These variables allow functions to behave more flexibly and more efficiently. In this case, the angle between the vectors is only converted to degrees if the user specified both output arguments, thus making `nargout` equal to 2. Otherwise, it is assumed that the user only wanted the angle in radians, so the conversion is never performed. MATLAB typically provides several interfaces to its functions, and uses `nargin` and `nargout` to determine which interface is being used. These multiple interfaces are described in the help pages for such functions.

Exercise 1.2.4 *Try calling* `vecangle2` *in various ways, with different numbers of input and output arguments, and with vectors of either the same or different lengths. Observe the behavior of* MATLAB *in each case.*

1.2.12 Graphics

Next, we learn some basic graphics commands. We begin by plotting the graph of the function $y = x^2$ on $[-1, 1]$. Start by creating a vector `x`, of equally spaced values between -1 and 1, using the colon operator. Then, create a vector `y` that contains the squares of the values in `x`. Make sure to use the correct approach to squaring the elements of a vector!

The `plot` command, in its simplest form, takes two input arguments that are vectors, that must have the same length. The first input argument contains x-values, and the second input argument contains y-values. The command `plot(x,y)` creates a new figure window (if none already exists) and plots `y` as a function of `x` in a set of axes contained in the figure window. Try plotting the graph of the function $y = x^2$ on $[-1, 1]$ using this command.

Note that by default, `plot` produces a solid blue curve. In reality, it is not a curve; it simply "connects the dots" using solid blue line segments, but if the segments are small enough, the resulting piecewise linear function resembles a smooth curve. But what if we want to plot curves using different colors, different line styles, or different symbols at each point?

Use the `help` command to view the help page for `plot`, which lists the specifications for different colors, line styles, and marker styles (which are used at each point that is plotted). The optional third argument to the `plot` command is used to specify these colors and styles. They can be mixed together; for example, the third argument `'r--'` plots a dashed red curve. Experiment with these different colors and styles, and with different functions.

Matlab provides several commands that can be used to produce more sophisticated plots. It is recommended that you view the help pages for these commands, and also experiment with their usage.

- `hold` is used to specify that subsequent `plot` commands should be superimposed on the same set of axes, rather than the default behavior in which the current axes are cleared with each new `plot` command.
- `subplot` is used to divide a figure window into an $m \times n$ matrix of axes, and specify which set of axes should be used for subsequent `plot` commands.
- `xlabel` and `ylabel` are used to label the horizontal and vertical axes, respectively, with given text.
- `title` is used to place given text at the top of a set of axes.
- `legend` is used to place a legend within a set of axes, so that the curves displayed on the axes can be labeled with given text.
- `gtext` is used to place given text at an arbitrary point within a figure window, indicated by clicking the mouse at the desired point.

Exercise 1.2.5 *Reproduce the plot shown in Figure 1.3 using the commands discussed in this section.*

Finally, it is essential to be able to save a figure so that it can be printed or included in a document. In the figure window, go to the File menu and choose "Save" or "Save As". You will

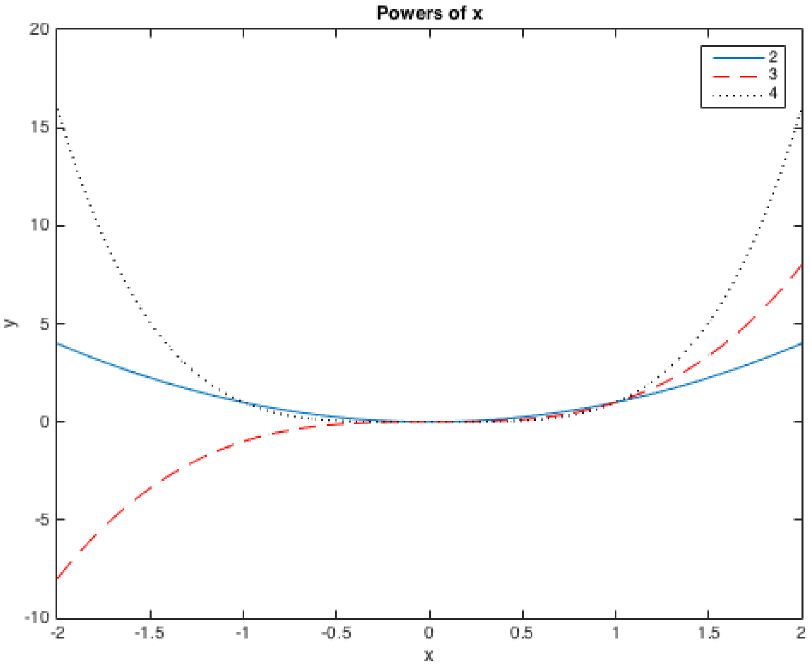


Figure 1.3: Figure for Exercise 1.2.5

see that the figure can be saved in a variety of standard image formats, such as JPEG or Windows bitmap (BMP).

Another format is “Matlab Figure (*.fig)”. It is highly recommended that you save your figure in this format, as well as the desired image format. Then, if you need to go back and change something about the figure after you have already closed the figure window, you can simply use the `open` command, with the `.fig` filename as its input argument, to reopen the figure window. Otherwise, you would have to recreate the entire figure from scratch.

1.2.13 Polynomial Functions

Finally, MATLAB provides several functions for working with polynomials. A polynomial is represented within MATLAB using a row vector of coefficients, with the highest degree coefficient listed first. For example, $f(x) = x^4 - 3x^2 + 2x + 5$ is represented by the vector `[ 1 0 -3 2 5 ]`. Note that the zero for the second element is necessary, or this vector would be interpreted as $x^3 - 3x^2 + 2x + 5$. The following functions work with polynomials in this format:

- `r=roots(p)` returns a column vector `r` consisting of the roots of the polynomial represented by `p`
- `p=poly(r)` is, in a sense, an inverse of `roots`. This function produces a row vector `p` that represents the *monic* polynomial (that is, with leading coefficient 1) whose roots are the entries of the vector `r`.

- `y=polyval(p,x)` evaluates the polynomial represented by `p` at all of the entries of `x` (which can be a scalar, vector or matrix) and returns the resulting values in `y`.
- `q=polyder(p)` computes the coefficients of the polynomial `q` that is the derivative of the polynomial `p`.
- `q=polyint(p)` computes the coefficients of the polynomial `q` that is the antiderivative, or indefinite integral, of the polynomial `p`. A constant of integration of zero is assumed.
- `r=conv(p,q)` computes the coefficients of the polynomial `r` that is the product of the polynomials `p` and `q`.

It is recommended that you experiment with these functions in order to get used to working with them.

1.2.14 Number Formatting

By default, numbers are displayed in "short" format, which uses 5 decimal places. The `format` command, with one of MATLAB's predefined formats, can be used to change how numbers are displayed. For example, type `format long` at the prompt, and afterwards, numbers are displayed using 15 decimal digits. Here is how 1/3 is displayed using various formats:

`short`	`0.33333`
`short e`	`3.33333e-01`
`long`	`0.333333333333333`
`long e`	`3.33333333333333e-01`
`bank`	`0.33`
`hex`	`3fd5555555555555`
`rat`	`1/3`

1.2.15 Inline and Anonymous Functions

Often, it is desirable to use functions in MATLAB that compute relatively simple expressions, but it is tedious to make a single small function M-file for each such function. Instead, very simple functions can be defined using the `inline` command:

```
>> f=inline('exp(sin(2*x))');
>> f(pi/4)
ans =
         2.7183
```

If an inline function takes more than one argument, it is important to specify which argument is first, which is second, and so on. For example, to construct an inline function for $f(x,y) = \sqrt{x^2 + y^2}$, it is best to proceed as follows:

```
>> f=inline('sqrt(x^2+y^2)','x','y');
>> f(2,1)
ans =
         2.2361
```

Alternatively, one can define an *anonymous* function as follows:

```
>> f=@(x)exp(sin(2*x));
>> f(0)
ans =
           1
```

Anonymous functions are helpful when it is necessary to pass a function `f` as a *parameter* to another function `g`, but `g` assumes that `f` has fewer input parameters than it actually accepts. An anonymous function can be used to fill in values for the extra parameters of `f` before it is passed to `g`.

This will be particularly useful when using MATLAB's functions for solving ordinary differential equations, as they expect the time derivative f in the ODE $dy/dt = f(t, y)$ to be a function of only two arguments. If the function `f` that computes the time derivative has additional input arguments, one can use, for example, the anonymous function `@(t,y)f(t,y,a,b,c)` where `a`, `b` and `c` are the additional input arguments, whose values must be known.

1.2.16 Saving and Loading Data

It is natural to have to end a MATLAB session, and then start a new session in which the data from the previous session needs to be re-used. Also, one might wish to send their data to another user. Fortunately, MATLAB supports saving variables to a file and loading them into a workspace.

The `save` command saves all variables in the current workspace (whether it is the top-level workspace, or the scope of a function) into a file. By default, this file is given a `.mat` extension. For example, `save mydata` saves all variables in a file called `mydata.mat` in the current working directory. Similarly, the `load` command loads data from a given file.

1.2.17 Using Octave

In Octave, by default output is *paginated*, meaning that output stops after each full page and waits for user input to continue. You can use the space bar to advance by a full page, or the enter key to advance one line at a time. Type `q` to immediately return to the prompt. To turn off pagination, type `more off` at the prompt. This makes the output behave like in MATLAB, which does not paginate output.

1.3 How to Use This Book

In order to get the most out of this book, it is recommended that you not simply read through it. As you have already seen, there are frequent “interruptions” in the text at which you are asked to perform some coding task. The purposes of these coding tasks are to get you accustomed to programming, particularly in MATLAB and Python, but also to reinforce the concepts of numerical analysis that are presented throughout the book.

Reading about the design, analysis and implementation of the many algorithms covered in this book is not sufficient to be able to fully understand these aspects or be able to efficiently and effectively work with these algorithms in code, especially as part of a larger application. A "hands-on" approach is needed to achieve this level of proficiency, and this book is written with this necessity in mind.

1.4 Error Analysis

Mathematical problems arising from scientific applications present a wide variety of difficulties that prevent us from solving them exactly. This has led to an equally wide variety of techniques for computing approximations to quantities occurring in such problems in order to obtain approximate solutions. In this chapter, we will describe the types of approximations that can be made, and learn some basic techniques for analyzing the accuracy of these approximations.

1.4.1 Sources of Approximation

Suppose that we are attempting to solve a particular instance of a problem arising from a mathematical model of a scientific application. We say that such a problem is *well-posed* if it meets the following criteria:

- A solution of the problem exists.
- The solution is unique.
- A small perturbation in the problem data results in a small perturbation in the solution; i.e., the solution *depends continuously* on the data.

By the first condition, the process of solving a well-posed problem can be seen to be equivalent to the evaluation of some function f at some known value x, where x represents the problem data. Since, in many cases, knowledge of the function f is limited, the task of computing $f(x)$ can be viewed, at least conceptually, as the execution of some (possibly infinite) sequence of steps that solves the underlying problem for the data x. The goal in numerical analysis is to develop a finite sequence of steps, i.e., an algorithm, for computing an approximation to the value $f(x)$.

There are two general types of error that occur in the process of computing this approximation to $f(x)$:

1. *data error* is the error in the data x. In reality, numerical analysis involves solving a problem with *approximate* data $\hat{x}$. The exact data is often unavailable because it must be obtained by measurements or other computations that fail to be exact due to limited precision. In addition, data may be altered in order to simplify the solution process.
2. *computational error* refers to the error that occurs when attempting to compute $f(\hat{x})$. Effectively, we must approximate $f(\hat{x})$ by the quantity $\hat{f}(\hat{x})$, where $\hat{f}$ is a function that approximates f. This approximation may be the result of *truncation*, which occurs when it is not possible to evaluate f exactly using a finite sequence of steps, and therefore a finite sequence that evaluates f approximately must be used instead. This particular source of computational error will be discussed in this chapter. Another source of computational error is roundoff error, which will discussed in Section 1.5.

Exercise 1.4.1 *Consider the process of computing* $\cos(\pi/4)$ *using a calculator or computer. Indicate sources of data error and computational error, including both truncation and roundoff error.*

1.4.2 Error Measurement

Now that we have been introduced to some specific errors that can occur during computation, we introduce useful terminology for discussing such errors. Suppose that a real number $\hat{y}$ is an approximation to some real number y. For instance, $\hat{y}$ may be the closest number to y that can be represented using finite precision, or $\hat{y}$ may be the result of a sequence of arithmetic operations performed using finite-precision arithmetic, where y is the result of the same operations performed using exact arithmetic.

Definition 1.4.1 (Absolute Error, Relative Error) *Let $\hat{y}$ be a real number that is an approximation to the real number y. The* **absolute error** *in $\hat{y}$ is*

$$E_{abs} = \hat{y} - y.$$

The **relative error** *in $\hat{y}$ is*

$$E_{rel} = \frac{\hat{y} - y}{y},$$

provided that y is nonzero.

The absolute error is the most natural measure of the accuracy of an approximation, but it can be misleading. Even if the absolute error is small in magnitude, the approximation may still be grossly inaccurate if the exact value y is even smaller in magnitude. For this reason, it is preferable to measure accuracy in terms of the relative error.

The magnitude of the relative error in $\hat{y}$ can be interpreted as a percentage of $|y|$. For example, if the relative error is greater than 1 in magnitude, then $\hat{y}$ can be considered completely erroneous, since the error is larger in magnitude as the exact value. Another useful interpretation of the relative error concerns *significant digits*, which are all digits excluding leading zeros. Specifically, if the relative error is at most β^{-p}, where β is an integer greater than 1, then the representation of $\hat{y}$ in base β has at least p correct significant digits.

It should be noted that the absolute error and relative error are often defined using absolute value; that is,

$$E_{abs} = |\hat{y} - y|, \quad E_{rel} = \left|\frac{\hat{y} - y}{y}\right|.$$

This definition is preferable when one is only interested in the magnitude of the error, which is often the case. If the sign, or direction, of the error is also of interest, then the first definition must be used.

Example 1.4.2 *If we add the numbers* 0.4567×10^0 *and* 0.8580×10^{-2}*, we obtain the exact result*

$$x = 0.4567 \times 10^0 + 0.008530 \times 10^0 = 0.46523 \times 10^0,$$

which is rounded to

$$\hat{x} = 0.4652 \times 10^0.$$

The absolute error in this computation is

$$E_{abs} = \hat{x} - x = 0.4652 - 0.46523 = -0.00003,$$

while the relative error is

$$E_{rel} = \frac{\hat{x} - x}{x} = \frac{0.4652 - 0.46523}{0.46523} \approx -0.000064484.$$

Now, suppose that we multiply 0.4567×10^4 *and* 0.8530×10^{-2}*. The exact result is*

$$x = (0.4567 \times 10^4) \times (0.8530 \times 10^{-2}) = 0.3895651 \times 10^2 = 38.95651,$$

which is rounded to

$$\hat{x} = 0.3896 \times 10^2 = 38.96.$$

The absolute error in this computation is

$$E_{abs} = \hat{x} - x = 38.96 - 38.95651 = 0.00349,$$

while the relative error is

$$E_{rel} = \frac{\hat{x} - x}{x} = \frac{38.96 - 38.95651}{38.95651} \approx 0.000089587.$$

We see that in this case, the relative error is smaller than the absolute error, because the exact result is larger than 1, whereas in the previous operation, the relative error was larger in magnitude, because the exact result is smaller than 1. □

Example 1.4.3 *Suppose that the exact value of a computation is supposed to be* 10^{-16}*, and an approximation of* 2×10^{-16} *is obtained. Then the absolute error in this approximation is*

$$E_{abs} = 2 \times 10^{-16} - 10^{-16} = 10^{-16},$$

which suggests the computation is accurate because this error is small. However, the relative error is

$$E_{rel} = \frac{2 \times 10^{-16} - 10^{-16}}{10^{-16}} = 1,$$

which suggests that the computation is completely erroneous, because by this measure, the error is equal in magnitude to the exact value; that is, the error is 100%. It follows that an approximation of zero would be just as accurate. This example, although an extreme case, illustrates why the absolute error can be a misleading measure of error. □

1.4.3 Forward and Backward Error

Suppose that we compute an approximation $\hat{y} = \hat{f}(x)$ of the value $y = f(x)$ for a given function f and given problem data x. Before we can analyze the accuracy of this approximation, we must have a precisely defined notion of error in such an approximation. We now provide this precise definition.

Definition 1.4.4 (Forward Error) *Let* x *be a real number and let* $f : \mathbb{R} \to \mathbb{R}$ *be a function. If* $\hat{y}$ *is a real number that is an approximation to* $y = f(x)$*, then the* **forward error** *in* $\hat{y}$ *is the difference* $\Delta y = \hat{y} - y$*. If* $y \neq 0$*, then the* **relative forward error** *in* $\hat{y}$ *is defined by*

$$\frac{\Delta y}{y} = \frac{\hat{y} - y}{y}.$$

Clearly, our primary goal in error analysis is to obtain an estimate of the forward error Δy. Unfortunately, it can be difficult to obtain this estimate directly.

An alternative approach is to instead view the computed value $\hat{y}$ as the *exact* solution of a problem with modified data; i.e., $\hat{y} = f(\hat{x})$ where $\hat{x}$ is a perturbation of x.

Definition 1.4.5 (Backward Error) *Let x be a real number and let $f : \mathbb{R} \to \mathbb{R}$ be a function. Suppose that the real number $\hat{y}$ is an approximation to $y = f(x)$, and that $\hat{y}$ is in the range of f; that is, $\hat{y} = f(\hat{x})$ for some real number $\hat{x}$. Then, the quantity $\Delta x = \hat{x} - x$ is the* **backward error** *in $\hat{y}$. If $x \neq 0$, then the* **relative forward error** *in $\hat{y}$ is defined by*

$$\frac{\Delta x}{x} = \frac{\hat{x} - x}{x}.$$

The process of estimating Δx is known as *backward error analysis*. As we will see, this estimate of the backward error, in conjunction with knowledge of f, can be used to estimate the forward error.

As will be discussed in Section 1.5, floating-point arithmetic does not follow the laws of real arithmetic. This tends to make forward error analysis difficult. In backward error analysis, however, real arithmetic is employed, since it is assumed that the computed result is the exact solution to a modified problem. This is one reason why backward error analysis is sometimes preferred.

Exercise 1.4.2 *Let $x_0 = 1$, and $f(x) = e^x$. If the magnitude of the forward error in computing $f(x_0)$, given by $|\hat{f}(x_0) - f(x_0)|$, is 0.01, then determine a bound on the magnitude of the backward error.*

Exercise 1.4.3 *For a general function $f(x)$, explain when the magnitude of the forward error is greater than, or less than, that of the backward error. Assume f is differentiable near x and use calculus to explain your reasoning.*

1.4.4 Conditioning and Stability

In most cases, the goal of error analysis is to obtain an estimate of the forward relative error $(f(\hat{x}) - f(x))/f(x)$, but it is often easier to instead estimate the relative backward error $(\hat{x} - x)/x$. Therefore, it is necessary to be able to estimate the forward error in terms of the backward error. The following definition addresses this need.

Definition 1.4.6 (Condition Number) *Let x be a real number and let $f : \mathbb{R} \to \mathbb{R}$ be a function. The* **absolute condition number***, denoted by κ_{abs}, is the ratio of the magnitude of the forward error to the magnitude of the backward error,*

$$\kappa_{abs} = \frac{|f(\hat{x}) - f(x)|}{|\hat{x} - x|}.$$

If $f(x) \neq 0$, then the **relative condition number** *of the problem of computing $y = f(x)$, denoted by κ_{rel}, is the ratio of the magnitude of the relative forward error to the magnitude of the relative backward error,*

$$\kappa_{rel} = \frac{|(f(\hat{x}) - f(x))/f(x)|}{|(\hat{x} - x)/x|} = \frac{|\Delta y/y|}{|\Delta x/x|}.$$

Intuitively, either condition number is a measure of the change in the solution due to a change in the data. Since the relative condition number tends to be a more reliable measure of this change, it is sometimes referred to as simply the *condition number*.

If the condition number is large, e.g. much greater than 1, then a small change in the data can cause a disproportionately large change in the solution, and the problem is said to be *ill-conditioned* or *sensitive*. If the condition number is small, then the problem is said to be *well-conditioned* or *insensitive*.

Since the condition number, as defined above, depends on knowledge of the exact solution $f(x)$, it is necessary to estimate the condition number in order to estimate the relative forward error. To that end, we assume, for simplicity, that $f : \mathbb{R} \to \mathbb{R}$ is differentiable and obtain

$$\begin{aligned}\kappa_{rel} &= \frac{|x\Delta y|}{|y\Delta x|} \\ &= \frac{|x(f(x+\Delta x) - f(x))|}{|f(x)\Delta x|} \\ &\approx \frac{|xf'(x)\Delta x|}{|f(x)\Delta x|} \\ &\approx \left|\frac{xf'(x)}{f(x)}\right|.\end{aligned}$$

Therefore, if we can estimate the backward error Δx, and if we can bound f and f' near x, we can then bound the relative condition number and obtain an estimate of the relative forward error. Of course, the relative condition number is undefined if the exact value $f(x)$ is zero. In this case, we can instead use the absolute condition number. Using the same approach as before, the absolute condition number can be estimated using the derivative of f. Specifically, we have $\kappa_{abs} \approx |f'(x)|$.

Exercise 1.4.4 *Let* $f(x) = e^x$, $g(x) = e^{-x}$, *and* $x_0 = 2$. *Suppose that the relative backward error in* x_0 *satisfies* $|\Delta x_0/x_0| = |\hat{x}_0 - x_0|/|x_0| \leq 10^{-2}$. *What is an upper bound on the relative forward error in* $f(x_0)$ *and* $g(x_0)$? *Use* MATLAB *or a calculator to experimentally confirm that this bound is valid, by evaluating* $f(x)$ *and* $g(x)$ *at selected points and comparing values.*

The condition number of a function f depends on, among other things, the absolute forward error $f(\hat{x}) - f(x)$. However, an algorithm for evaluating $f(x)$ actually evaluates a function $\hat{f}$ that approximates f, producing an approximation $\hat{y} = \hat{f}(x)$ to the exact solution $y = f(x)$. In our definition of backward error, we have assumed that $\hat{f}(x) = f(\hat{x})$ for some $\hat{x}$ that is close to x; i.e., our approximate solution to the original problem is the exact solution to a "nearby" problem. This assumption has allowed us to define the condition number of f independently of any approximation $\hat{f}$. This independence is necessary, because the sensitivity of a problem depends solely on the problem itself and not any algorithm that may be used to approximately solve it.

Is it always reasonable to assume that any approximate solution is the exact solution to a nearby problem? Unfortunately, it is not. It is possible that an algorithm that yields an accurate approximation for given data may be unreasonably sensitive to perturbations in that data. This leads to the concept of a *stable algorithm*: an algorithm applied to a given problem with given data x is said to be *stable* if it computes an approximate solution that is the exact solution to the same problem with data $\hat{x}$, where $\hat{x}$ is a small perturbation of x.

It can be shown that if a problem is well-conditioned, and if we have a stable algorithm for solving it, then the computed solution can be considered accurate, in the sense that the relative error in the computed solution is small. On the other hand, a stable algorithm applied to an ill-conditioned problem cannot be expected to produce an accurate solution.

Example 1.4.7 *This example will illustrate the last point made above. To solve a system of linear equations* $A\mathbf{x} = \mathbf{b}$ *in* MATLAB*, we can use the* \ *operator:*

$$\texttt{x} = \texttt{A}\backslash\texttt{b}$$

Enter the following matrix and column vectors in MATLAB*, as shown. Recall that a semicolon (;) separates rows.*

```
>> A=[ 0.6169 0.4798; 0.4925 0.3830 ]
>> b1=[ 0.7815; 0.6239 ]
>> b2=[ 0.7753; 0.6317 ]
```

Then, solve the systems `A*x1` = `b1` *and* `A*x2` = `b2`*. Note that* `b1` *and* `b2` *are not very different, but what about the solutions* `x1` *and* `x2`*? The algorithm implemented by the* \ *operator is stable, but what can be said about the* conditioning *of the problem of solving* `Ax` = `b` *for this matrix* `A`*? The conditioning of systems of linear equations will be studied in depth in Chapter 2.* □

Exercise 1.4.5 *Let* $f(x)$ *be a function that is one-to-one. Then, solving the equation* $f(x) = c$ *for some* c *in the range of* f *is equivalent to computing* $x = f^{-1}(c)$*. What is the condition number of the problem of solving* $f(x) = c$*?*

1.4.5 Convergence

Many algorithms in numerical analysis are *iterative methods* that produce a sequence $\{\alpha_n\}$ of approximate solutions which, ideally, converges to a limit α that is the exact solution as n approaches ∞. Because we can only perform a finite number of iterations, we cannot obtain the exact solution, and we have introduced computational error.

If our iterative method is properly designed, then this computational error will approach zero as n approaches ∞. However, it is important that we obtain a sufficiently accurate approximate solution using as few computations as possible. Therefore, it is not practical to simply perform enough iterations so that the computational error is determined to be sufficiently small, because it is possible that another method may yield comparable accuracy with less computational effort.

The total computational effort of an iterative method depends on both the effort per iteration and the number of iterations performed. Therefore, in order to determine the amount of computation that is needed to attain a given accuracy, we must be able to measure the error in α_n as a function of n. The more rapidly this function approaches zero as n approaches ∞, the more rapidly the sequence of approximations $\{\alpha_n\}$ converges to the exact solution α, and as a result, fewer iterations are needed to achieve a desired accuracy. We now introduce some terminology that will aid in the discussion of the convergence behavior of iterative methods.

Definition 1.4.8 (Big-O Notation) *Let f and g be two functions defined on a domain $D \subseteq \mathbb{R}$ that is not bounded above. We write that $f(n) = O(g(n))$ if there exists a positive constant c such that*

$$|f(n)| \leq c|g(n)|, \quad n \geq n_0,$$

for some $n_0 \in D$.

As sequences are functions defined on $\mathbb{N}$, the domain of the natural numbers, we can apply big-O notation to sequences. Therefore, this notation is useful to describe the rate at which a sequence of computations converges to a limit.

Definition 1.4.9 (Rate of Convergence) *Let $\{\alpha_n\}_{n=1}^{\infty}$ and $\{\beta_n\}_{n=1}^{\infty}$ be sequences that satisfy*

$$\lim_{n\to\infty} \alpha_n = \alpha, \quad \lim_{n\to\infty} \beta_n = 0,$$

where α is a real number. We say that $\{\alpha_n\}$ converges to α with **rate of convergence** *$O(\beta_n)$ if $\alpha_n - \alpha = O(\beta_n)$.*

We say that an iterative method converges rapidly, in some sense, if it produces a sequence of approximate solutions whose rate of convergence is $O(\beta_n)$, where the terms of the sequence β_n approach zero rapidly as n approaches ∞. Intuitively, if two iterative methods for solving the same problem perform a comparable amount of computation during each iteration, but one method exhibits a faster rate of convergence, then that method should be used because it will require less overall computational effort to obtain an approximate solution that is sufficiently accurate.

Example 1.4.10 *Consider the sequence $\{\alpha_n\}_{n=1}^{\infty}$ defined by*

$$\alpha_n = \frac{n+1}{n+2}, \quad n = 1, 2, \ldots$$

Then, we have

$$\begin{aligned} \lim_{n\to\infty} \alpha_n &= \lim_{n\to\infty} \frac{n+1}{n+2}\frac{1/n}{1/n} \\ &= \lim_{n\to\infty} \frac{1+1/n}{1+2/n} \\ &= \frac{\lim_{n\to\infty}(1+1/n)}{\lim_{n\to\infty}(1+2/n)} \\ &= \frac{1+\lim_{n\to\infty} 1/n}{1+\lim_{n\to\infty} 2/n} \\ &= \frac{1+0}{1+0} \\ &= 1. \end{aligned}$$

That is, the sequence $\{\alpha_n\}$ converges to $\alpha = 1$. To determine the rate of convergence, we note that

$$\alpha_n - \alpha = \frac{n+1}{n+2} - 1 = \frac{n+1}{n+2} - \frac{n+2}{n+2} = \frac{-1}{n+2},$$

and since

$$\left|\frac{-1}{n+2}\right| \leq \left|\frac{1}{n}\right|$$

for any positive integer n*, it follows that*

$$\alpha_n = \alpha + O\left(\frac{1}{n}\right).$$

On the other hand, consider the sequence $\{\alpha_n\}_{n=1}^\infty$ *defined by*

$$\alpha_n = \frac{2n^2+4n}{n^2+2n+1}, \quad n = 1, 2, \ldots$$

Then, we have

$$\begin{aligned}
\lim_{n\to\infty} \alpha_n &= \lim_{n\to\infty} \frac{2n^2+4n}{n^2+2n+1}\frac{1/n^2}{1/n^2} \\
&= \lim_{n\to\infty} \frac{2+4/n}{1+2/n+1/n^2} \\
&= \frac{\lim_{n\to\infty}(2+4/n)}{\lim_{n\to\infty}(1+2/n+1/n^2)} \\
&= \frac{2+\lim_{n\to\infty} 4/n}{1+\lim_{n\to\infty}(2/n+1/n^2)} \\
&= 2.
\end{aligned}$$

That is, the sequence $\{\alpha_n\}$ *converges to* $\alpha = 2$*. To determine the rate of convergence, we note that*

$$\alpha_n - \alpha = \frac{2n^2+4n}{n^2+2n+1} - 2 = \frac{2n^2+4n}{n^2+2n+1} - \frac{2n^2+4n+2}{n^2+2n+1} = \frac{-2}{n^2+2n+1},$$

and since

$$\left|\frac{-2}{n^2+2n+1}\right| = \left|\frac{2}{(n+1)^2}\right| \leq \left|\frac{2}{n^2}\right|$$

for any positive integer n*, it follows that*

$$\alpha_n = \alpha + O\left(\frac{1}{n^2}\right).$$

□

We can also use big-O notation to describe the rate of convergence of a *function* to a limit.

Example 1.4.11 *Consider the function* $f(h) = 1 + 2h$*. Since this function is continuous for all* h*, we have*

$$\lim_{h\to 0} f(h) = f(0) = 1.$$

It follows that

$$f(h) - f_0 = (1 + 2h) - 1 = 2h = O(h),$$

so we can conclude that as $h \to 0$*,* $1 + 2h$ *converges to 1 of order* $O(h)$*.* □

Example 1.4.12 *Consider the function* $f(h) = 1 + 4h + 2h^2$*. Since this function is continuous for all* h*, we have*

$$\lim_{h\to 0} f(h) = f(0) = 1.$$

It follows that

$$f(h) - f_0 = (1 + 4h + 2h^2) - 1 = 4h + 2h^2.$$

To determine the rate of convergence as $h \to 0$*, we consider* h *in the interval* $[-1, 1]$*. In this interval,* $|h^2| \leq |h|$*. It follows that*

$$\begin{aligned} |4h + 2h^2| &\leq |4h| + |2h^2| \\ &\leq |4h| + |2h| \\ &\leq 6|h|. \end{aligned}$$

Since there exists a constant C *(namely, 6) such that* $|4h + 2h^2| \leq C|h|$ *for* h *satisfying* $|h| \leq h_0$ *for some* h_0 *(namely, 1), we can conclude that as* $h \to 0$*,* $1 + 4h + 2h^2$ *converges to 1 of order* $O(h)$*.* □

In general, when $f(h)$ denotes an approximation that depends on h, and

$$f_0 = \lim_{h\to 0} f(h)$$

denotes the exact value, $f(h) - f_0$ represents the absolute error in the approximation $f(h)$. When this error is a polynomial in h, as in this example and the previous example, the rate of convergence is $O(h^k)$ where k is the smallest exponent of h in the error. This is because as $h \to 0$, the smallest power of h approaches zero more slowly than higher powers, thereby making the dominant contribution to the error.

By contrast, when determining the rate of convergence of a sequence $\{\alpha_n\}$ as $n \to \infty$, the *highest* power of n determines the rate of convergence. As powers of n are negative if convergence occurs at all as $n \to \infty$, and powers of h are positive if convergence occurs at all as $h \to 0$, it can be said that for either types of convergence, it is the exponent that is closest to zero that determines the rate of convergence.

Example 1.4.13 *Consider the function* $f(h) = \cos h$*. Since this function is continuous for all* h*, we have*

$$\lim_{h\to 0} f(h) = f(0) = 1.$$

Using Taylor's Theorem, with center $h_0 = 0$*, we obtain*

$$f(h) = f(0) + f'(0)h + \frac{f''(\xi(h))}{2}h^2,$$

where $\xi(h)$ *is between 0 and* h*. Substituting* $f(h) = \cos h$ *into the above, we obtain*

$$\cos h = 1 - (\sin 0)h + \frac{-\cos\xi(h)}{2}h^2,$$

or

$$\cos h = 1 - \frac{\cos\xi(h)}{2}h^2.$$

Because $|\cos x| \leq 1$ *for all* x*, we have*

$$|\cos h - 1| = \left|-\frac{\cos\xi(h)}{2}h^2\right| \leq \frac{1}{2}h^2,$$

so we can conclude that as $h \to 0$*,* $\cos h$ *converges to 1 of order* $O(h^2)$*.* □

Exercise 1.4.6 *Determine the rate of convergence of*

$$\lim_{h\to 0} e^h - h - \frac{1}{2}h^2 = 1.$$

Example 1.4.14 *The following approximation to the derivative is based on the definition:*

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0)}{h}, \quad \textit{error} = -\frac{h}{2}f''(\xi).$$

An alternative approximation is

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0-h)}{2h}, \quad \textit{error} = -\frac{h^2}{6}f'''(\xi).$$

Taylor series can be used to obtain the error terms in both cases.

Exercise 1.4.7 *Use Taylor's Theorem to derive the error terms in both formulas.*

Exercise 1.4.8 *Try both of these approximations with* $f(x) = \sin x$*,* $x_0 = 1$*, and* $h = 10^{-1}, 10^{-2}, 10^{-3}$*, and then* $h = 10^{-14}$*. What happens, and can you explain* why*?*

If you can't explain what happens for the smallest value of h*, fortunately this will be addressed in Section 1.5.* □

1.5 Computer Arithmetic

In computing the solution to any mathematical problem, there are many sources of error that can impair the accuracy of the computed solution. The study of these sources of error is called *error analysis*, which will be discussed later in this section. First, we will focus on one type of error that occurs in all computation, whether performed by hand or on a computer: *roundoff error.*

This error is due to the fact that in computation, real numbers can only be represented using a finite number of digits. In general, it is not possible to represent real numbers exactly with this limitation, and therefore they must be approximated by real numbers that *can* be represented using a fixed number of digits, which is called the *precision.* Furthermore, as we shall see, arithmetic operations applied to numbers that can be represented exactly using a given precision do not necessarily produce a result that can be represented using the same precision. It follows that if a fixed precision is used, then *every* arithmetic operation introduces error into a computation.

Given that scientific computations can have several sources of error, one would think that it would be foolish to compound the problem by performing arithmetic using fixed precision. However, using a fixed precision is actually far more practical than other options, and, as long as computations are performed carefully, sufficient accuracy can still be achieved.

1.5.1 Floating-Point Representation

We now describe a typical system for representing real numbers on a computer.

Definition 1.5.1 (Floating-point Number System) *Given integers* $\beta > 1$, $p \geq 1$, L, *and* $U \geq L$, *a* **floating-point number system** $\mathbb{F}$ *is defined to be the set of all real numbers of the form*

$$x = \pm m\beta^E.$$

The number m *is the* **mantissa** *of* x, *and has the form*

$$m = \left(\sum_{j=0}^{p-1} d_j \beta^{-j}\right),$$

where each digit d_j, $j = 0, \ldots, p-1$ *is an integer satisfying* $0 \leq d_i \leq \beta - 1$. *The number* E *is called the* **exponent** *or the* **characteristic** *of* x, *and it is an integer satisfying* $L \leq E \leq U$. *The integer* p *is called the* **precision** *of* $\mathbb{F}$, *and* β *is called the* **base** *of* $\mathbb{F}$.

The term “floating-point” comes from the fact that as a number $x \in \mathbb{F}$ is multiplied by or divided by a power of β, the mantissa does not change, only the exponent. As a result, the decimal point shifts, or “floats,” to account for the changing exponent. Nearly all computers use a binary floating-point system, in which $\beta = 2$.

Example 1.5.2 *Let* $x = -117$. *Then, in a floating-point number system with base* $\beta = 10$, x *is represented as*

$$x = -(1.17)10^2,$$

where 1.17 *is the mantissa and 2 is the exponent. If the base* $\beta = 2$, *then we have*

$$x = -(1.110101)2^6,$$

where 1.110101 *is the mantissa and 6 is the exponent. The mantissa should be interpreted as a string of binary digits, rather than decimal digits; that is,*

$$\begin{aligned} 1.110101 &= 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 1 \cdot 2^{-6} \\ &= 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{64} \\ &= \frac{117}{64} \\ &= \frac{117}{2^6}. \end{aligned}$$

□

Exercise 1.5.1 *Express* $x = \pi$ *as accurately as possible in a floating-point number system with base* $\beta = 2$ *and precision* $p = 12$.

1.5.1.1 Overflow and Underflow

A floating-point system $\mathbb{F}$ can only represent a finite subset of the real numbers. As such, it is important to know how large in magnitude a number can be and still be represented, at least approximately, by a number in $\mathbb{F}$. Similarly, it is important to know how small in magnitude a number can be and still be represented by a *nonzero* number in $\mathbb{F}$; if its magnitude is too small, then it is most accurately represented by zero.

Definition 1.5.3 (Underflow, Overflow) *Let $\mathbb{F}$ be a floating-point number system. The smallest positive number in $\mathbb{F}$ is called the* **underflow level***, and it has the value*

$$UFL = m_{\min}\beta^L,$$

where L is the smallest valid exponent and $m_{\min}$ is the smallest mantissa. The largest positive number in $\mathbb{F}$ is called the overflow level*, and it has the value*

$$OFL = \beta^{U+1}(1-\beta^{-p}).$$

The value of $m_{\min}$ depends on whether floating-point numbers are *normalized* in $\mathbb{F}$; this point will be discussed later. The overflow level is the value obtained by setting each digit in the mantissa to $\beta - 1$ and using the largest possible value, U, for the exponent.

Exercise 1.5.2 *Determine the value of OFL for a floating-point system with base $\beta = 2$, precision $p = 53$, and largest exponent $U = 1023$.*

It is important to note that the real numbers that can be represented in $\mathbb{F}$ are not equally spaced along the real line. Numbers having the same exponent are equally spaced, and the spacing between numbers in $\mathbb{F}$ decreases as their magnitude decreases.

1.5.1.2 Normalization

It is common to *normalize* floating-point numbers by specifying that the leading digit d_0 of the mantissa be nonzero. In a binary system, with $\beta = 2$, this implies that the leading digit is equal to 1, and therefore need not be stored. In addition to the benefit of gaining one additional bit of precision, normalization also ensures that each floating-point number has a unique representation.

One drawback of normalization is that fewer numbers near zero can be represented exactly than if normalization is not used. One workaround is a practice called *gradual underflow*, in which the leading digit of the mantissa is allowed to be zero when the exponent is equal to L, thus allowing smaller values of the mantissa. In such a system, the number UFL is equal to β^{L-p+1}, whereas in a normalized system, UFL $= \beta^L$.

Exercise 1.5.3 *Determine the value of UFL for a floating-point system with base $\beta = 2$, precision $p = 53$, and smallest exponent $L = -1022$, both with and without normalization.*

1.5.1.3 Rounding

A number that can be represented exactly in a floating-point system is called a *machine number*. Since only finitely many real numbers are machine numbers, it is necessary to determine how non-machine numbers are to be approximated by machine numbers. The process of choosing a machine number to approximate a non-machine number is called *rounding*, and the error introduced by such an approximation is called *roundoff error*. Given a real number x, the machine number obtained by rounding x is denoted by $\mathrm{fl}(x)$.

In most floating-point systems, rounding is achieved by one of two strategies:

- *chopping*, or *rounding to zero*, is the simplest strategy, in which the base-β expansion of a number is truncated after the first p digits. As a result, $\mathrm{fl}(x)$ is the unique machine number between 0 and x that is nearest to x.

- *rounding to nearest* sets $\mathrm{fl}(x)$ to be the machine number that is closest to x in absolute value; if two numbers satisfy this property, then an appropriate tie-breaking rule must be used, such as setting $\mathrm{fl}(x)$ equal to the choice whose last digit is even.

Example 1.5.4 *Suppose we are using a floating-point system with $\beta = 10$ (decimal), with $p = 4$ significant digits. Then, if we use chopping, or rounding to zero, we have fl$(2/3) = 0.6666$, whereas if we use rounding to nearest, then we have fl$(2/3) = 0.6667$.* □

Example 1.5.5 *When rounding to even in decimal, 88.5 is rounded to 88, not 89, so that the last digit is even, while 89.5 is rounded to 90, again to make the last digit even.* □

1.5.1.4 Machine Precision

In error analysis, it is necessary to estimate error incurred in each step of a computation. As such, it is desirable to know an upper bound for the relative error introduced by rounding. This leads to the following definition.

Definition 1.5.6 (Machine Precision) *Let $\mathbb{F}$ be a floating-point number system. The* **unit roundoff** *or* **machine precision**, *denoted by* $\mathbf{u}$, *is the real number that satisfies*

$$\left|\frac{\mathrm{fl}(x) - x}{x}\right| \leq \mathbf{u}$$

for any real number x such that $\mathrm{UFL} < x < \mathrm{OFL}$.

An intuitive definition of $\mathbf{u}$ is that it is the smallest positive number such that

$$\mathrm{fl}\,(1 + \mathbf{u}) > 1.$$

The value of $\mathbf{u}$ depends on the rounding strategy that is used. If rounding toward zero is used, then $\mathbf{u} = \beta^{1-p}$, whereas if rounding to nearest is used, $\mathbf{u} = \frac{1}{2}\beta^{1-p}$.

It is important to avoid confusing $\mathbf{u}$ with the underflow level UFL. The unit roundoff is determined by the number of digits in the mantissa, whereas the underflow level is determined by the range of allowed exponents. However, we do have the relation that $0 < \mathrm{UFL} < \mathbf{u}$.

In analysis of roundoff error, it is assumed that $\mathrm{fl}(x \;\mathtt{op}\; y) = (x \;\mathtt{op}\; y)(1 + \delta)$, where `op` is an arithmetic operation and δ is an unknown constant satisfying $|\delta| \leq \mathbf{u}$. From this assumption, it can be seen that the relative error in $\mathrm{fl}(x \;\mathtt{op}\; y)$ is $|\delta|$. In the case of addition, the relative backward error in each operand is also $|\delta|$.

1.5.1.5 The IEEE Floating-Point Standard

In fact, most computers conform to the *IEEE standard* for floating-point arithmetic. The standard specifies, among other things, how floating-point numbers are to be represented in memory. Two representations are given, one for *single-precision* and one for *double-precision.*

Under the standard, single-precision floating-point numbers occupy 4 bytes in memory, with 23 bits used for the mantissa, 8 for the exponent, and one for the sign. IEEE double-precision floating-point numbers occupy eight bytes in memory, with 52 bits used for the mantissa, 11 for the exponent, and one for the sign. That is, in the IEEE floating-point standard, $p = 24$ for single precision, and $p = 53$ for double precision, even though only 23 and 52 bits, respectively, are used to store mantissas.

Example 1.5.7 *The following table summarizes the main aspects of a general floating-point system and a double-precision floating-point system that uses a 52-bit mantissa and 11-bit exponent. For both systems, we assume that rounding to nearest is used, and that normalization is used.* □

	General	*Double Precision*
Form of machine number	$\pm m\beta^E$	$\pm 1.d_1 d_2 \cdots d_{52} 2^E$
Precision	p	*53*
Exponent range	$L \leq E \leq U$	$-1023 \leq E \leq 1024$
UFL (Underflow Level)	β^L	2^{-1023}
OFL (Overflow Level)	$\beta^{U+1}(1-\beta^{-p})$	$2^{1025}(1-2^{-53})$
$\mathbf{u}$	$\frac{1}{2}\beta^{1-p}$	2^{-53}

Exercise 1.5.4 *Are the values for UFL and OFL given in the table above the actual values used in the IEEE double-precision floating point system? Experiment with powers of 2 in* Matlab *to find out. What are the largest and smallest positive numbers you can represent? Can you explain any discrepancies between these values and the ones in the table?*

1.5.2 Issues with Floating-Point Arithmetic

We now discuss the various issues that arise when performing *floating-point arithmetic*, or *finite-precision arithmetic*, which approximates arithmetic operations on real numbers.

1.5.3 Loss of Precision

When adding or subtracting floating-point numbers, it is necessary to shift one of the operands so that both operands have the same exponent, before adding or subtracting the mantissas. As a result, digits of precision are lost in the operand that is smaller in magnitude, and the result of the operation cannot be represented using a machine number. In fact, if x is the smaller operand and y is the larger operand, and $|x| < |y|\mathbf{u}$, then the result of the operation will simply be y (or $-y$, if y is to be subtracted from x), since the entire value of x is lost in rounding the result.

Exercise 1.5.5 *Consider the evaluation of the summation*

$$\sum_{i=1}^{n} x_i,$$

where each term x_i is positive. Will the sum be computed more accurately in floating-point arithmetic if the numbers are added in order from smallest to largest, or largest to smallest? Justify your answer.

In multiplication or division, the operands need not be shifted, but the mantissas, when multiplied or divided, cannot necessarily be represented using only p digits of precision. The product of two mantissas requires $2p$ digits to be represented exactly, while the quotient of two mantissas could conceivably require infinitely many digits.

1.5.3.1 Violation of Arithmetic Rules

Because floating-point arithmetic operations are not exact, they do not follow all of the laws of real arithmetic. In particular, floating-point arithmetic is not associative; i.e., $x + (y + z) \neq (x + y) + z$ in floating-point arithmetic.

Exercise 1.5.6 *In* MATLAB*, generate three random numbers x, y and z, and compute $x + (y + z)$ and $(x + y) + z$. Do they agree? Try this a few times with different random numbers and observe what happens.*

1.5.3.2 Overflow and Underflow

Furthermore, overflow or underflow may occur depending on the exponents of the operands, since their sum or difference may lie outside of the interval $[L, U]$.

Exercise 1.5.7 *Consider the formula $z = \sqrt{x^2 + y^2}$. Explain how overflow can occur in computing z, even if x, y and z all have magnitudes that can be represented. How can this formula be rewritten so that overflow does not occur?*

1.5.3.3 Cancellation

Subtraction of floating-point numbers presents a unique difficulty, in addition to the rounding error previously discussed. If the operands, after shifting exponents as needed, have leading digits in common, then these digits cancel and the first digit in which the operands do not match becomes the leading digit. However, since each operand is represented using only p digits, it follows that the result contains only $p - m$ correct digits, where m is the number of leading digits that cancel.

In an extreme case, if the two operands differ by less than $\mathbf{u}$, then the result contains no correct digits; it consists entirely of roundoff error from previous computations. This phenomenon is known as *catastrophic cancellation.* Because of the highly detrimental effect of this cancellation, it is important to ensure that no steps in a computation compute small values from relatively large operands. Often, computations can be rearranged to avoid this risky practice.

Example 1.5.8 *When performing the subtraction*

$$\begin{array}{r} 0.345769258233 \\ -0.345769258174 \\ \hline 0.000000000059 \end{array}$$

only two significant digits can be included in the result, even though each of the operands includes twelve, because ten digits of precision are lost to catastrophic cancellation. □

Example 1.5.9 *Consider the quadratic equation*

$$ax^2 + bx + c = 0,$$

which has the solutions

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

Suppose that $b > 0$*. Then, in computing* x_1*, we encounter catastrophic cancellation if* b *is much larger than* a *and* c*, because this implies that* $\sqrt{b^2 - 4ac} \approx b$ *and as a result we are subtracting two numbers that are nearly equal in computing the numerator. On the other hand, if* $b < 0$*, we encounter this same difficulty in computing* x_2*.*

Suppose that we use 4-digit rounding arithmetic to compute the roots of the equation

$$x^2 + 10,000x + 1 = 0.$$

Then, we obtain $x_1 = 0$ *and* $x_2 = -10,000$*. Clearly,* x_1 *is incorrect because if we substitute* $x = 0$ *into the equation then we obtain the contradiction* $1 = 0$*. In fact, if we use 7-digit rounding arithmetic then we obtain the same result. Only if we use at least 8 digits of precision do we obtain roots that are reasonably correct,*

$$x_1 \approx -1 \times 10^{-4}, \quad x_2 \approx -9.9999999 \times 10^3.$$

A similar result is obtained if we use 4-digit rounding arithmetic but compute x_1 *using the formula*

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

which can be obtained by multiplying and dividing the original formula for x_1 *by the* conjugate *of the numerator,* $-b - \sqrt{b^2 - 4ac}$*. The resulting formula is not susceptible to catastrophic cancellation, because an addition is performed instead of a subtraction.* □

Exercise 1.5.8 *Use the* MATLAB *function* `randn` *to generate 1000 normally distributed random numbers with mean 1000 and standard deviation 0.1. Then, use these formulas to compute the variance:*

1. $\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$
2. $\left(\frac{1}{n}\sum_{i=1}^{n} x_i^2\right) - \bar{x}^2$

where $\bar{x}$ *is the mean. How do the results differ? Which formula is more susceptible to issues with floating-point arithmetic, and why?*

Exercise 1.5.9 *Recall Exercise 1.4.8, in which two approximations of the derivative were tested using various values of the spacing* h*. In light of the discussion in this chapter, explain the behavior for the case of* $h = 10^{-14}$.

Part II

Numerical Linear Algebra

Chapter 2

Methods for Systems of Linear Equations

In this chapter we discuss methods for solving the problem $A\mathbf{x} = \mathbf{b}$, where A is a square invertible matrix of size n, $\mathbf{x}$ is an unknown n-vector, and $\mathbf{b}$ is a vector of size n. Solving a system of linear equations comes up in a number of applications such as solving a linear system of ordinary differential equations (ODEs).

2.1 Triangular Systems

The basic idea behind methods for solving a system of linear equations is to reduce them to linear equations involving a single unknown, because such equations are trivial to solve. These types of equations make up a system that we call *triangular systems of equations.* There are three types of triangular systems:

1. upper triangular, where $a_{ij} = 0$ for $i > j$,
2. diagonal, where $a_{ij} = 0$ for $i \neq j$,
3. and lower triangular, where $a_{ij} = 0$ for $i < j$.

2.1.1 Upper Triangular Systems

In this section we will look at how each system can be solved efficiently, and analyze the cost of each method. First we will study the upper triangular system, which has the form

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1j} \\ & a_{21} & a_{22} & \cdots & a_{2j} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & \vdots \\ 0 & & & & a_{ij} \end{bmatrix}.$$

To illustrate this process, let us take a look at a sytem of equations in upper triangular form where $n = 3$.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{33}x_3 &= b_3 \end{aligned}$$

From the above augmented matrix, it is easy to see that we can now simply solve

$$\begin{aligned} a_{33}x_3 &= b_3 \\ x_3 &= b_3/a_{33} \end{aligned}$$

Now that we have found x_3, we substitute it in to the previous equation to solve for the unknown in that equation, which is

$$\begin{aligned} a_{22}x_2 + a_{23}x_3 &= b_2 \\ x_2 &= (b_2 - a_{23})/a_{22}. \end{aligned}$$

Similarly as before, now substitute x_2, and x_3 into the first equation, and again we have a linear equation with only one unknown.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \end{aligned}$$

The process we have previously described is known as *back subsitution.* Just by looking at the above 3×3 case you can see a pattern emerging, and this pattern can be described in terms of an algorithm that can be used to solve these types of triangular systems.

Exercise 2.1.1 *(a) Write a* MATLAB *function to solve the following triangular system:*

$$\begin{aligned} 2x + y - 3z &= -10 \\ -2y + z &= -2 \\ z &= 6 \end{aligned}$$

(b) How many floating point operations does this function perform?

In the algorithm, we assume that U is the upper triangular matrix containing the coefficients of the system, and $\mathbf{y}$ is the vector containing the right-hand sides of the equations.

```
for $i = n, n-1, \ldots, 1$ do
    $x_i = y_i$
    for $j = i+1, i+2, \ldots, n$ do
        $x_i = x_i - u_{ij}x_j$
    end
    $x_i = x_i/u_{ii}$
end
```

Upper triangular systems are the goal we are trying to achieve through Gaussian elimnation, which we discuss in the next section. If we take a look at the number of level of loops in the above algorithm, we can see that there are two levels of nesting involved, therefore it takes n^2 floating point operations to solve these systems.

2.1.2 Diagonal Systems

Now we switch our focus to a system that is even simpler to solve. This system as the form

$$A = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}.$$

From looking at the augmented matrix, $[A \quad \mathbf{b}]$,

$$A = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 & | & b_1 \\ 0 & a_{22} & 0 & \cdots & 0 & | & b_2 \\ 0 & 0 & a_{33} & \cdots & 0 & | & b_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & | & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} & | & b_n \end{bmatrix}$$

it can be seen that in the system, each linear equation only has one unknown, so we can solve each equation independently of the other equations. It does not matter which equation we start with in solving this system, since they can all be solved independently of each other. But for the purpose of this book, we will start at the top. The equations we need to solve are as follows:

$$\begin{aligned} a_{11}x_1 &= b_1 \\ a_{22}x_2 &= b_2 \\ a_{33}x_3 &= b_3 \\ &\vdots \\ a_{nn}x_n &= b_n. \end{aligned}$$

From the above we can see that each solution is found by $x_n = b_n/a_{nn}$.

Exercise 2.1.2 *(a) Write a* Matlab *function to solve the following diagonal system of equations:*

$$\begin{aligned} 3x \quad\quad\quad &= 4 \\ 2y \quad\quad &= 8 \\ 7z &= 21 \end{aligned}$$

(b) How many floating point operations does this function perform?

We can also look at this in terms of an algorithm:

```
for $i = 1, 2, \ldots, n$ do
    $x_i = b_i/a_{ii}$
end
```

We can see that the above algorithm only has 1 level of nested loops, therefore the number of floating point operations required to solve a system of this type is only n^1 or just n!

2.1.3 Lower Triangular Systems

The next type of system we will look at are lower triangular systems, which has the form

$$A = \begin{bmatrix} a_{11} & & & & 0 \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & & & \\ \vdots & \vdots & \ddots & & \\ a_{i1} & a_{i2} & \cdots & a_{i,j-1} & a_{ij} \end{bmatrix}.$$

These types of systems can be solved similarly to the upper triangular systems using a method called *forward substitution*. That will be discussed later in this chapter.

2.2 Gaussian Elimination

2.2.1 Row Operations

Now that we have looked at the simplest cases, we want to switch our focus to solving a general square invertible system. An approach that we use is to first reduce our original system to one of the types of systems discussed in Section 2.1, since we now know an efficient way to solve these systems. Such a reduction is achieved by manipulating the equations in the system in such a way that the solution does not change, but unknowns are eliminated from selected equations until, finally, we obtain an equation involving only a single unknown. These manipulations are called *elementary row operations*, and they are defined as follows:

- Multiplying both sides of an equation by a scalar ($R_i \to sR_i$)

- Reordering the equations by interchanging both sides of the ith and jth equation in the system $(R_i \to R_j)$
- Replacing equation i by the sum of equation i and a multiple of both sides of equation j $(R_j \to R_j - sR_i)$

Exercise 2.2.1 *Prove that the following row operations do not change the solution set.*

- *Multiplying both sides of an equation by a scalar.*
- *Replacing equation i by the sum of equation i and a multiple of both sides of equation.*

The third operation is by far the most useful. We will now demonstrate how it can be used to reduce a system of equations to a form in which it can easily be solved.

Example Consider the system of linear equations

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 5, \\ 3x_1 + 2x_2 + 4x_3 &= 17, \\ 4x_1 + 4x_2 + 3x_3 &= 26. \end{aligned}$$

First, we eliminate x_1 from the second equation by subtracting 3 times the first equation from the second. This yields the equivalent system

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 5, \\ -4x_2 + x_3 &= 2, \\ 4x_1 + 4x_2 + 3x_3 &= 26. \end{aligned}$$

Next, we subtract 4 times the first equation from the third, to eliminate x_1 from the third equation as well:

$$\begin{aligned} x_2 + 2x_2 + x_3 &= 5, \\ -4x_2 + x_3 &= 2, \\ -4x_2 - x_3 &= 6. \end{aligned}$$

Then, we eliminate x_2 from the third equation by subtracting the second equation from it, which yields the system

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 5, \\ -4x_2 + x_3 &= 2, \\ -2x_3 &= 4. \end{aligned}$$

This system is in *upper-triangular form*, because the third equation depends only on x_3, and the second equation depends on x_2 and x_3.

Because the third equation is a linear equation in x_3, it can easily be solved to obtain $x_3 = -2$.

$$\begin{array}{rcl} x_1 + 2x_2 + x_3 &=& 5, \\ -4x_2 + x_3 &=& 2, \\ -2x_3 &=& 4. \end{array} \quad \rightarrow \quad \begin{array}{rcl} x_1 + 2x_2 + x_3 &=& 5, \\ -4x_2 + x_3 &=& 2, \\ x_3 &=& -2. \end{array}$$

Then, we can substitute this value into the second equation, which yields $-4x_2 = 4$.

$$\begin{array}{rcl} x_1 + 2x_2 + x_3 &=& 5, \\ -4x_2 + -2 &=& 2, \\ x_3 &=& -2. \end{array} \quad \rightarrow \quad \begin{array}{rcl} x_1 + 2x_2 + x_3 &=& 5, \\ x_2 &=& -1, \\ x_3 &=& -2. \end{array}$$

This equation only depends on x_2, so we can easily solve it to obtain $x_2 = -1$. Finally, we substitute the values of x_2 and x_3 into the first equation to obtain $x_1 = 9$.

$$\begin{array}{rcl} x_1 + 2(-1) + -2 &=& 5, \\ x_2 &=& -1, \\ x_3 &=& -2. \end{array} \quad \rightarrow \quad \begin{array}{rcl} x_1 &=& 9, \\ x_2 &=& -1, \\ x_3 &=& -2. \end{array}$$

This process of computing the unknowns from a system that is in upper-triangular form is called *back substitution.* □

Exercise 2.2.2 *Write a* MATLAB *function to solve the following upper triangular system using back substitution.*

$$\begin{array}{rcl} 3x_1 + 2x_2 + x_3 - x_4 &=& 0 \\ -x_2 + 2x_3 + x_4 &=& 0 \\ x_3 + x_4 &=& 1 \\ -x_4 &=& 2 \end{array}$$

Your function should return the solution, and should take the corresponding matrix and right hand side vector as input.

In general, a system of n linear equations in n unknowns is in upper-triangular form if the ith equation depends only on the unknowns x_j for $i < j \leq n$.

Now, performing row operations on the system $A\mathbf{x} = \mathbf{b}$ can be accomplished by performing them on the *augmented matrix*

$$\left[\begin{array}{cc} A & \mathbf{b} \end{array}\right] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & & & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right].$$

By working with the augmented matrix instead of the original system, there is no need to continually rewrite the unknowns or arithmetic operators. Once the augmented matrix is reduced to upper triangular form, the corresponding system of linear equations can be solved by back substitution, as before.

The process of eliminating variables from the equations, or, equivalently, zeroing entries of the corresponding matrix, in order to reduce the system to upper-triangular form is called *Gaussian elimination*. We will now step through an example as we discuss the steps of the Gaussian Elimination algorithm. The algorithm is as follows:

```
for $j = 1, 2, \ldots, n-1$ do
    for $i = j+1, j+2, \ldots, n$ do
        $m_{ij} = a_{ij}/a_{jj}$
        for $k = j+1, j+2, \ldots, n$ do
            $a_{ik} = a_{ik} - m_{ij}a_{jk}$
        end
        $b_i = b_i - m_{ij}b_j$
    end
end
```

This algorithm requires approximately $\frac{2}{3}n^3$ arithmetic operations, so it can be quite expensive if n is large. Later, we will discuss alternative approaches that are more efficient for certain kinds of systems, but Gaussian elimination remains the most generally applicable method of solving systems of linear equations.

The number m_{ij} is called a *multiplier*. It is the number by which row j is multiplied before adding it to row i, in order to eliminate the unknown x_j from the ith equation. Note that this algorithm is applied to the augmented matrix, as the elements of the vector $\mathbf{b}$ are updated by the row operations as well.

It should be noted that in the above description of Gaussian elimination, each entry below the main diagonal is never explicitly zeroed, because that computation is unnecessary. It is only necessary to update entries of the matrix that are involved in subsequent row operations or the solution of the resulting upper triangular system. We will see that when solving systems of equations in which the right-hand side vector $\mathbf{b}$ is changing, but the coefficient matrix A remains fixed, it is quite practical to apply Gaussian elimination to A only once, and then repeatedly apply it to each $\mathbf{b}$, along with back substitution, because the latter two steps are much less expensive.

We now illustrate the use of both these algorithms with an example.

Example Consider the system of linear equations

$$\begin{aligned} x_1 + 2x_2 + x_3 - x_4 &= 5 \\ 3x_1 + 2x_2 + 4x_3 + 4x_4 &= 16 \\ 4x_1 + 4x_2 + 3x_3 + 4x_4 &= 22 \\ 2x_1 + x_3 + 5x_4 &= 15. \end{aligned}$$

This system can be represented by the coefficient matrix A and right-hand side vector $\mathbf{b}$, as follows:

$$A = \begin{bmatrix} 1 & 2 & 1 & -1 \\ 3 & 2 & 4 & 4 \\ 4 & 4 & 3 & 4 \\ 2 & 0 & 1 & 5 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 5 \\ 16 \\ 22 \\ 15 \end{bmatrix}.$$

To perform row operations to reduce this system to upper triangular form, we define the augmented matrix

$$\tilde{A} = \left[\begin{array}{cc} A & \mathbf{b} \end{array}\right] = \left[\begin{array}{rrrrr} 1 & 2 & 1 & -1 & 5 \\ 3 & 2 & 4 & 4 & 16 \\ 4 & 4 & 3 & 4 & 22 \\ 2 & 0 & 1 & 5 & 15 \end{array}\right].$$

We first define $\tilde{A}^{(1)} = \tilde{A}$ to be the original augmented matrix. Then, we denote by $\tilde{A}^{(2)}$ the result of the first elementary row operation, which entails subtracting 3 times the first row from the second in order to eliminate x_1 from the second equation:

$$\tilde{A}^{(2)} = \left[\begin{array}{rrrrr} 1 & 2 & 1 & -1 & 5 \\ 0 & -4 & 1 & 7 & 1 \\ 4 & 4 & 3 & 4 & 22 \\ 2 & 0 & 1 & 5 & 15 \end{array}\right].$$

Next, we eliminate x_1 from the third equation by subtracting 4 times the first row from the third:

$$\tilde{A}^{(3)} = \left[\begin{array}{rrrrr} 1 & 2 & 1 & -1 & 5 \\ 0 & -4 & 1 & 7 & 1 \\ 0 & -4 & -1 & 8 & 2 \\ 2 & 0 & 1 & 5 & 15 \end{array}\right].$$

Then, we complete the elimination of x_1 by subtracting 2 times the first row from the fourth:

$$\tilde{A}^{(4)} = \left[\begin{array}{rrrrr} 1 & 2 & 1 & -1 & 5 \\ 0 & -4 & 1 & 7 & 1 \\ 0 & -4 & -1 & 8 & 2 \\ 0 & -4 & -1 & 7 & 5 \end{array}\right].$$

We now need to eliminate x_2 from the third and fourth equations. This is accomplished by subtracting the second row from the third, which yields

$$\tilde{A}^{(5)} = \left[\begin{array}{rrrrr} 1 & 2 & 1 & -1 & 5 \\ 0 & -4 & 1 & 7 & 1 \\ 0 & 0 & -2 & 1 & 1 \\ 0 & -4 & -1 & 7 & 5 \end{array}\right],$$

and the fourth, which yields

$$\tilde{A}^{(6)} = \left[\begin{array}{rrrrr} 1 & 2 & 1 & -1 & 5 \\ 0 & -4 & 1 & 7 & 1 \\ 0 & 0 & -2 & 1 & 1 \\ 0 & 0 & -2 & 0 & 4 \end{array}\right].$$

Finally, we subtract the third row from the fourth to obtain the augmented matrix of an upper-triangular system,

$$\tilde{A}^{(7)} = \left[\begin{array}{rrrrr} 1 & 2 & 1 & -1 & 5 \\ 0 & -4 & 1 & 7 & 1 \\ 0 & 0 & -2 & 1 & 1 \\ 0 & 0 & 0 & -1 & 3 \end{array}\right].$$

Note that in a matrix for such a system, all entries below the *main diagonal* (the entries where the row index is equal to the column index) are equal to zero. That is, $a_{ij} = 0$ for $i > j$.

From this, we see that we need to examine all columns, $i = 1 \dots n-1$. Likewise, we need to examine rows where $j = i+1, i+2, \dots, n$. Now we are getting ready to build our algorithm.

Now, we can perform back substitution on the corresponding system,

$$\begin{aligned} x_1 + 2x_2 + x_3 - x_4 &= 5, \\ -4x_2 + x_3 + 7x_4 &= 1, \\ -2x_3 + x_4 &= 1, \\ -x_4 &= 3, \end{aligned}$$

to obtain the solution, which yields $x_4 = -3$, $x_3 = -2$, $x_2 = -6$, and $x_1 = 16$. □

2.2.2 The *LU* Factorization

We have learned how to solve a system of linear equations $A\mathbf{x} = \mathbf{b}$ by applying Gaussian elimination to the augmented matrix $\tilde{A} = \left[\begin{array}{cc} A & \mathbf{b} \end{array}\right]$, and then performing back substitution on the resulting upper-triangular matrix. However, this approach is not practical if the right-hand side $\mathbf{b}$ of the system is changed, while A is not. This is due to the fact that the choice of $\mathbf{b}$ has no effect on the row operations needed to reduce A to upper-triangular form. Therefore, it is desirable to instead apply these row operations to A only once, and then "store" them in some way in order to apply them to any number of right-hand sides.

To accomplish this, we first note that subtracting m_{ij} times row j from row i to eliminate a_{ij} is equivalent to multiplying A by the matrix

$$M_{ij} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & 0 & & & & & 0 \\ \vdots & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & -m_{ij} & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & & & 0 & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{bmatrix},$$

where the entry $-m_{ij}$ is in row i, column j. Each such matrix M_{ij} is an example of an *elementary row matrix*, which is a matrix that results from applying any elementary row operation to the identity matrix.

More generally, if we let $A^{(1)} = A$ and let $A^{(k+1)}$ be the matrix obtained by eliminating elements of column k in $A^{(k)}$, then we have, for $k = 1, 2, \dots, n-1$,

$$A^{(k+1)} = M^{(k)} A^{(k)}$$

where

$$M^{(k)} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & 0 & & & & & & 0 \\ \vdots & \ddots & \ddots & \ddots & & & & & \vdots \\ \vdots & & 0 & \ddots & \ddots & & & & \vdots \\ \vdots & & \vdots & -m_{k+1,k} & \ddots & \ddots & & & \vdots \\ \vdots & & \vdots & \vdots & 0 & \ddots & \ddots & & \vdots \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \vdots & \vdots & \vdots & & \ddots & 1 & 0 \\ 0 & \cdots & 0 & -m_{nk} & 0 & \cdots & \cdots & 0 & 1 \end{bmatrix},$$

with the elements $-m_{k+1,k}, \ldots, -m_{nk}$ occupying column k. It follows that the matrix

$$U = A^{(n)} = M^{(n-1)}A^{(n-1)} = M^{(n-1)}M^{(n-2)} \cdots M^{(1)}A$$

is upper triangular, and the vector

$$\mathbf{y} = M^{(n-1)}M^{(n-2)} \cdots M^{(1)}\mathbf{b},$$

being the result of applying the same row operations to $\mathbf{b}$, is the right-hand side for the upper-triangular system that is to be solved by back subtitution.

Exercise 2.2.3 *(a) Write a* MATLAB *function that computes the matrix U by using the above description. Start in the first column, and accumulate all of the multipliers in a matrix. Once you have done this for each $n-1$ columns, you can multiply them together in the manner described to accumulate the matrix $U = M^{(n-1)}M^{(n-2)} \ldots M^1 A$.*

(b) Your function should store the values of m_{ij} in the appropriate entry of a matrix we will call L. This matrix will be lower unit triangular, and is discussed in the next section.

2.2.2.1 Unit Lower Triangular Matrices

We have previously learned about *upper triangular* matrices that result from Gaussian elimination. Recall that an $m \times n$ matrix A is upper triangular if $a_{ij} = 0$ whenever $i > j$. This means that all entries below the *main diagonal*, which consists of the entries a_{11}, a_{22}, $\ldots$, are equal to zero. A system of linear equations of the form $U\mathbf{x} = \mathbf{y}$, where U is an $n \times n$ nonsingular upper triangular matrix, can be solved by back substitution.

Exercise 2.2.4 *Prove that such a matrix, as in U described above, is nonsingular if and only if all of its diagonal entries are nonzero.*

Similarly, a matrix L is *lower triangular* if all of its entries above the main diagonal, that is, entries ℓ_{ij} for which $i < j$, are equal to zero. We will see that a system of equations of the form $L\mathbf{y} = \mathbf{b}$, where L is an $n \times n$ nonsingular lower triangular matrix, can be solved using a process similar to back substitution, called *forward substitution.* As with upper triangular matrices, a lower triangular matrix is nonsingular if and only if all of its diagonal entries are nonzero.

Exercise 2.2.5 *Prove the following useful properties for triangular matrices. Triangular matrices have the following useful properties:*

(a) The product of two upper/lower triangular matrices is upper/lower triangular.

(b) The inverse of a nonsingular upper/lower triangular matrix is upper/lower triangular.

Now that you have proven these properties, we can say that matrix multiplication and inversion preserve triangularity.

Now, we note that each matrix $M^{(k)}$, $k = 1, 2, \ldots, n-1$, is not only a lower-triangular matrix, but a *unit lower triangular* matrix, because all of its diagonal entries are equal to 1. Next, we note two important properties of unit lower/upper triangular matrices:

- The product of two unit lower/upper triangular matrices is unit lower/upper triangular.
- A unit lower/upper triangular matrix is nonsingular, and its inverse is unit lower/upper triangular.

In fact, the inverse of each $M^{(k)}$ is easily computed. We have

$$L^{(k)} = [M^{(k)}]^{-1} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & 0 & & & & & & 0 \\ \vdots & \ddots & \ddots & \ddots & & & & & \vdots \\ \vdots & & 0 & \ddots & \ddots & & & & \vdots \\ \vdots & & \vdots & m_{k+1,k} & \ddots & \ddots & & & \vdots \\ \vdots & & \vdots & \vdots & 0 & \ddots & \ddots & & \vdots \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \vdots & \vdots & \vdots & & \ddots & 1 & 0 \\ 0 & \cdots & 0 & m_{nk} & 0 & \cdots & \cdots & 0 & 1 \end{bmatrix}.$$

It follows that if we define $M = M^{(n-1)} \cdots M^{(1)}$, then M is unit lower triangular, and $MA = U$, where U is upper triangular. It follows that $A = M^{-1}U = LU$, where

$$L = L^{(1)} \cdots L^{(n-1)} = [M^{(1)}]^{-1} \cdots [M^{(n-1)}]^{-1}$$

is also unit lower triangular. Furthermore, from the structure of each matrix $L^{(k)}$, it can readily be determined that

$$L = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ m_{21} & 1 & 0 & & \vdots \\ \vdots & m_{32} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & 1 & 0 \\ m_{n1} & m_{n2} & \cdots & m_{n,n-1} & 1 \end{bmatrix}.$$

That is, L stores all of the multipliers used during Gaussian elimination. The factorization of A that we have obtained,

$$A = LU,$$

is called the *LU decomposition*, or *LU factorization*, of A.

Exercise 2.2.6 *Write a* MATLAB *function to compute the matrices L and U for a randomly generated matrix. Check your accuracy by multiplying them together to see if you get $A = LU$.*

2.2.2.2 Solution of $A\mathbf{x} = \mathbf{b}$

Once the LU decomposition $A = LU$ has been computed, we can solve the system $A\mathbf{x} = \mathbf{b}$ by first noting that if $\mathbf{x}$ is the solution, then

$$A\mathbf{x} = LU\mathbf{x} = \mathbf{b}.$$

Therefore, we can obtain $\mathbf{x}$ by first solving the system

$$L\mathbf{y} = \mathbf{b},$$

and then solving

$$U\mathbf{x} = \mathbf{y}.$$

Then, if $\mathbf{b}$ should change, then only these last two systems need to be solved in order to obtain the new solution; the LU decomposition does not need to be recomputed.

The system $U\mathbf{x} = \mathbf{y}$ can be solved by back substitution, since U is upper-triangular. To solve $L\mathbf{y} = \mathbf{b}$, we can use *forward substitution*, since L is unit lower triangular.

```
for $i = 1, 2, \ldots, n$ do
    $y_i = b_i$
    for $j = 1, 2, \ldots, i-1$ do
        $y_i = y_i - \ell_{ij} y_j$
    end
end
```

Exercise 2.2.7 *Implement a function for forward substitution in* MATLAB*. Try your function on the following lower unit triangular matrix.*

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 3 & 1 & 0 \\ 4 & 6 & 4 & 1 \end{bmatrix}$$

Like back substitution, this algorithm requires $O(n^2)$ floating-point operations. Unlike back substitution, there is no division of the ith component of the solution by a diagonal element of the matrix, but this is only because in this context, L is unit lower triangular, so $\ell_{ii} = 1$. When applying forward substitution to a general lower triangular matrix, such a division is required.

Example The matrix

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

can be reduced to the upper-triangular matrix

$$U = \begin{bmatrix} 1 & 2 & 1 & -1 \\ 0 & -4 & 1 & 7 \\ 0 & 0 & -2 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

by performing the following row operations. These row operations are represented below in the *elementary row matrices* $E_1, \dots E_6$.

- Subtracting three times the first row from the second
- Subtracting four times the first row from the third
- Subtracting two times the first row from the fourth
- Subtracting the second row from the third
- Subtracting the second row from the fourth
- Subtracting the third row from the fourth

$$E_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad E_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$E_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -2 & 0 & 0 & 1 \end{bmatrix}, \quad E_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$E_5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}, \quad E_6 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}.$$

Exercise 2.2.8 *(a) How might these elementary row matrices be used as a part of the matrix factorization? Consider how you would apply each one of them individually to the matrix* A *to get the intermediate result,* $U = L^{-1}A$.

(b) Check your results by multiplying the matrices in the order that you have discovered to get U *as a result.*

(c) Explain why this works, what is this matrix multiplication equivalent to?

Now that you have discovered what we call pre-multiplying the matrix A by L^{-1}, we know that

$$L = E_1^{-1}E_2^{-1}E_3^{-1}E_4^{-1}E_5^{-1}E_6^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 4 & 1 & 1 & 0 \\ 2 & 1 & 1 & 1 \end{bmatrix},$$

and we have the factorization $A = LU$.

We see that L is a unit lower-triangular matrix, with the subdiagonal entries equal to the multipliers. That is, if m_{ij} times row j is subtracted from row i, where $i > j$, then $\ell_{ij} = m_{ij}$.

Let $\mathbf{b} = \begin{bmatrix} 5 & 1 & 22 & 15 \end{bmatrix}^T$. Applying the same row operations to $\mathbf{b}$, which is equivalent to pre-multiplying $\mathbf{b}$ by L^{-1}, or solving the system $L\mathbf{y} = \mathbf{b}$, yields the modified right-hand side

$$\mathbf{y} = E_6E_5E_4E_3E_2E_1\mathbf{b} = \begin{bmatrix} 5 \\ 1 \\ 1 \\ 3 \end{bmatrix}.$$

We then use back substitution to solve the system $U\mathbf{x} = \mathbf{y}$:

$$\begin{aligned} x_4 &= y_4/u_{44} = 3/(-1) = -3, \\ x_3 &= (y_3 - u_{34}x_4)/u_{33} = (1 - 1(-3))/(-2) = -2, \\ x_2 &= (y_2 - u_{23}x_3 - u_{24}x_4)/u_{22} = (1 - 1(-2) - 7(-3))/(-4) = -6, \\ x_1 &= (y_1 - u_{12}x_2 - u_{13}x_3 - u_{14}x_4) \\ &= (5 - 2(-6) - 1(-2) + 1(-3))/1 = 16. \end{aligned}$$

□

2.2.2.3 Implementation Details

Because both forward and back substituion require only $O(n^2)$ operations, whereas Gaussian elimination requires $O(n^3)$ operations, changes in the right-hand side $\mathbf{b}$ can be handled quite efficiently by computing the factors L and U once, and storing them. This can be accomplished quite efficiently, because L is unit lower triangular. It follows from this that L and U can be stored in a single $n \times n$ matrix by storing U in the upper triangular part, and the multipliers m_{ij} in the lower triangular part. Once the factorization $A = LU$ has been found, there are two systems to solve once the substitution $A = LU$ has been made into $A\mathbf{x} = \mathbf{b}$ to get $LU\mathbf{x} = \mathbf{b}$.

$$\begin{aligned} L \quad U\mathbf{x} &= \mathbf{b} && \text{Let } U\mathbf{x} = \mathbf{y} \\ L\mathbf{y} &= \mathbf{b} && \text{Solving requires } O(n^2) \\ U\mathbf{x} &= \mathbf{y} && \text{Solving requires } O(n^2) \end{aligned}$$

2.2.2.4 Existence and Uniqueness

Not every nonsingular $n \times n$ matrix A has an LU decomposition. For example, if $a_{11} = 0$, then the multipliers m_{i1}, for $i = 2, 3, \ldots, n$, are not defined, so no multiple of the first row can be added to the other rows to eliminate subdiagonal elements in the first column. That is, Gaussian elimination can break down. Even if $a_{11} \neq 0$, it can happen that the (j, j) element of $A^{(j)}$ is zero, in which case a similar breakdown occurs. When this is the case, the *LU* decomposition of A does not exist. This will be addressed by *pivoting*, resulting in a modification of the *LU* decomposition.

It can be shown that the *LU* decomposition of an $n \times n$ matrix A *does* exist if the *leading principal submatrices* of A, defined by

$$[A]_{1:k,1:k} = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{k1} & \cdots & a_{kk} \end{bmatrix}, \quad k = 1, 2, \ldots, n,$$

are all nonsingular. Furthermore, when the *LU* decomposition exists, it is unique.

Exercise 2.2.9 *Prove, by contradiction, that the LU factorization is unique. Start by supposing that two distinct LU factorizations exist.*

2.2.2.5 Practical Computation of Determinants

Computing the determinant of an $n \times n$ matrix A using its definition requires a number of arithmetic operations that is exponential in n. However, more practical methods for computing the determinant can be obtained by using its properties:

- If $\tilde{A}$ is obtained from A by adding a multiple of a row of A to another row, then $\det(\tilde{A}) = \det(A)$.
- If B is an $n \times n$ matrix, then $\det(AB) = \det(A)\det(B)$.
- If A is a triangular matrix (either upper or lower), then $\det(A) = \prod_{i=1}^{n} a_{ii}$.

It follows from these properties that if Gaussian elimination is used to reduce A to an upper-triangular matrix U, then $\det(A) = \det(U)$, where U is the resulting upper-triangular matrix, because the elementary row operations needed to reduce A to U do not change the determinant. Because U is upper triangular, $\det(U)$, being the product of its diagonal entries, can be computed in $n - 1$ multiplications. It follows that the determinant of any matrix can be computed in $O(n^3)$ operations.

It can also be seen that $\det(A) = \det(U)$ by noting that if $A = LU$, then $\det(A) = \det(L)\det(U)$, by one of the abovementioned properties, but $\det(L) = 1$, because L is a unit lower triangular matrix. It follows from the fact that L is lower triangular that $\det(L)$ is the product of its diagonal entries, and it follows from the fact that L is *unit* lower triangular that all of its diagonal entries are equal to 1.

2.2.2.6 Perturbations and the Inverse

Using what we have learned about matrix norms and convergence of sequences of matrices, we can quantify the change in the inverse of a matrix A in terms of the change in A. Suppose that an $n \times n$ matrix F satisfies $\|F\|_p < 1$, for some p. Then, from our previous discussion, $\lim_{k\to\infty} F^k = 0$. It follows that, by convergence of telescoping series,

$$\left(\lim_{k\to\infty} \sum_{i=0}^{k} F^i \right) (I - F) = \lim_{k\to\infty} I - F^{k+1} = I,$$

and therefore $(I-F)$ is nonsingular, with inverse $(I-F)^{-1} = \sum_{i=0}^{\infty} F^i$. By the properties of matrix norms, and convergence of geometric series, we then obtain

$$\|(I-F)^{-1}\|_p \leq \sum_{i=0}^{\infty} \|F\|_p^i = \frac{1}{1-\|F\|_p}.$$

Now, let A be a nonsingular matrix, and let E be a perturbation of A such that $r \equiv \|A^{-1}E\|_p < 1$. Because $A+E = A(I-F)$ where $F = -A^{-1}E$, with $\|F\|_p = r < 1$, $I-F$ is nonsingular, and therefore so is $A+E$. We then have

$$\begin{aligned}
\|(A+E)^{-1} - A^{-1}\|_p &= \|-A^{-1}E(A+E)^{-1}\|_p \\
&= \|-A^{-1}E(I-F)^{-1}A^{-1}\|_p \\
&\leq \|A^{-1}\|_p^2 \|E\|_p \|(I-F)^{-1}\|_p \\
&\leq \frac{\|A^{-1}\|_p^2 \|E\|_p}{1-r},
\end{aligned}$$

from the formula for the difference of inverses, and the submultiplicative property of matrix norms.

2.2.2.7 Bounding the perturbation in A

From a roundoff analysis of forward and back substitution, which we do not reproduce here, we have the bounds

$$\begin{aligned}
\max_{i,j} |\delta \bar{L}_{ij}| &\leq n\mathbf{u}\ell + O(\mathbf{u}^2), \\
\max_{i,j} |\delta \bar{U}_{ij}| &\leq n\mathbf{u}Ga + O(\mathbf{u}^2)
\end{aligned}$$

where $a = \max_{i,j} |a_{ij}|$, $\ell = \max_{i,j} |\bar{L}_{ij}|$, and G is the growth factor. Putting our bounds together, we have

$$\begin{aligned}
\max_{i,j} |\delta A_{ij}| &\leq \max_{i,j} |e_{ij}| + \max_{i,j} |\bar{L}\delta \bar{U}_{ij}| + \max_{i,j} |\bar{U}\delta \bar{L}_{ij}| + \max_{i,j} |\delta \bar{L}\delta \bar{U}_{ij}| \\
&\leq n(1+\ell)Ga\mathbf{u} + n^2\ell Ga\mathbf{u} + n^2\ell Ga\mathbf{u} + O(\mathbf{u}^2)
\end{aligned}$$

from which it follows that

$$\|\delta A\|_\infty \leq n^2(2n\ell + \ell + 1)Ga\mathbf{u} + O(\mathbf{u}^2).$$

2.2.2.8 Bounding the error in the solution

Let $\bar{\mathbf{x}} = \mathbf{x} + \delta\mathbf{x}$ be the computed solution. Since the exact solution to $A\mathbf{x} = \mathbf{b}$ is given by $\mathbf{x} = A^{-1}\mathbf{b}$, we are also interested in examining $\bar{\mathbf{x}} = (A+\delta A)^{-1}\mathbf{b}$. Can we say something about $\|(A+\delta A)^{-1}\mathbf{b} - A^{-1}\mathbf{b}\|$?

We assume that $\|A^{-1}\delta A\| = r < 1$. We have

$$A + \delta A = A(I + A^{-1}\delta A) = A(I-F), \quad F = -A^{-1}\delta A.$$

From the manipulations

$$\begin{aligned}(A+\delta A)^{-1}\mathbf{b}-A^{-1}\mathbf{b} &= (I+A^{-1}\delta A)^{-1}A^{-1}\mathbf{b}-A^{-1}\mathbf{b}\\ &= (I+A^{-1}\delta A)^{-1}(A^{-1}-(I+A^{-1}\delta A)A^{-1})\mathbf{b}\\ &= (I+A^{-1}\delta A)^{-1}(-A^{-1}(\delta A)A^{-1})\mathbf{b}\end{aligned}$$

and this result from Section 2.2.2.6,

$$\|(I-F)^{-1}\| \leq \frac{1}{1-r},$$

we obtain

$$\begin{aligned}\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} &= \frac{\|(\mathbf{x}+\delta\mathbf{x})-\mathbf{x}\|}{\|\mathbf{x}\|}\\ &= \frac{\|(A+\delta A)^{-1}\mathbf{b}-A^{-1}\mathbf{b}\|}{\|A^{-1}\mathbf{b}\|}\\ &\leq \frac{1}{1-r}\|A^{-1}\|\|\delta A\|\\ &\leq \frac{1}{1-\|A^{-1}\delta A\|}\kappa(A)\frac{\|\delta A\|}{\|A\|}\\ &\leq \frac{\kappa(A)}{1-\kappa(A)\frac{\|\delta A\|}{\|A\|}}\frac{\|\delta A\|}{\|A\|}.\end{aligned}$$

Note that a similar conclusion is reached if we assume that the computed solution $\bar{\mathbf{x}}$ solves a nearby problem in which *both* A and $\mathbf{b}$ are perturbed, rather than just A.

We see that the important factors in the accuracy of the computed solution are

- The growth factor G
- The size of the multipliers m_{ik}, bounded by ℓ
- The condition number $\kappa(A)$
- The precision $\mathbf{u}$

In particular, $\kappa(A)$ must be large with respect to the accuracy in order to be troublesome. For example, consider the scenario where $\kappa(A) = 10^2$ and $\mathbf{u} = 10^{-3}$, as opposed to the case where $\kappa(A) = 10^2$ and $\mathbf{u} = 10^{-50}$. However, it is important to note that even if A is well-conditioned, the error in the solution can still be very large, if G and ℓ are large.

2.2.3 Pivoting

During Gaussian elimination, it is necessary to interchange rows of the augmented matrix whenever the diagonal element of the column currently being processed, known as the *pivot element*, is equal to zero.

However, if we examine the main step in Gaussian elimination,

$$a_{ik}^{(j+1)} = a_{ik}^{(j)} - m_{ij}a_{jk}^{(j)},$$

we can see that any roundoff error in the computation of $a_{jk}^{(j)}$ is amplified by m_{ij}. Because the multipliers can be arbitrarily large, it follows from the previous analysis that the error in the computed solution can be arbitrarily large, meaning that Gaussian elimination is *numerically unstable*.

Therefore, it is helpful if it can be ensured that the multipliers are small. This can be accomplished by performing row interchanges, or *pivoting*, even when it is not absolutely necessary to do so for elimination to proceed.

2.2.3.1 Partial Pivoting

One approach is called *partial pivoting*. When eliminating elements in column j, we seek the largest element in column j, on or below the main diagonal, and then interchanging that element's row with row j. That is, we find an integer p, $j \leq p \leq n$, such that

$$|a_{pj}| = \max_{j\leq i\leq n} |a_{ij}|.$$

Then, we interchange rows p and j.

In view of the definition of the multiplier, $m_{ij} = a_{ij}^{(j)}/a_{jj}^{(j)}$, it follows that $|m_{ij}| \leq 1$ for $j = 1, \ldots, n-1$ and $i = j+1, \ldots, n$. Furthermore, while pivoting in this manner requires $O(n^2)$ comparisons to determine the appropriate row interchanges, that extra expense is negligible compared to the overall cost of Gaussian elimination, and therefore is outweighed by the potential reduction in roundoff error. We note that when partial pivoting is used, the growth factor G is 2^{n-1}, where A is $n \times n$.

2.2.3.2 Complete Pivoting

While partial pivoting helps to control the propagation of roundoff error, loss of significant digits can still result if, in the abovementioned main step of Gaussian elimination, $m_{ij}a_{jk}^{(j)}$ is much larger in magnitude than $a_{ij}^{(j)}$. Even though m_{ij} is not large, this can still occur if $a_{jk}^{(j)}$ is particularly large.

Complete pivoting entails finding integers p and q such that

$$|a_{pq}| = \max_{j\leq i\leq n, j\leq q\leq n} |a_{ij}|,$$

and then using both row *and column* interchanges to move a_{pq} into the pivot position in row j and column j. It has been proven that this is an effective strategy for ensuring that Gaussian elimination is *backward stable*, meaning it does not cause the entries of the matrix to grow exponentially as they are updated by elementary row operations, which is undesirable because it can cause undue amplification of roundoff error.

2.2.3.3 The *LU* Decomposition with Pivoting

Suppose that pivoting is performed during Gaussian elimination. Then, if row j is interchanged with row p, for $p > j$, before entries in column j are eliminated, the matrix $A^{(j)}$ is effectively multiplied by a *permutation matrix* $P^{(j)}$. A permutation matrix is a matrix obtained by permuting the rows (or columns) of the identity matrix I. In $P^{(j)}$, rows j and p of I are interchanged, so that multiplying $A^{(j)}$ on the left by $P^{(j)}$ interchanges these rows of $A^{(j)}$. It follows that the process of Gaussian elimination with pivoting can be described in terms of the matrix multiplications.

Exercise 2.2.10 *(a) Find the order, as described above, in which the permutation matrices P and the multiplier matrices M should be multiplied by A.*

(b) What permutation matrix P would constitute no change in the previous matrix?

However, because each permutation matrix $P^{(k)}$ at most interchanges row k with row p, where $p > k$, there is no difference between applying all of the row interchanges "up front", instead of applying $P^{(k)}$ immediately before applying $M^{(k)}$ for each k. It follows that

$$[M^{(n-1)}M^{(n-2)}\cdots M^{(1)}][P^{(n-1)}P^{(n-2)}\cdots P^{(1)}]A = U,$$

and because a product of permutation matrices is a permutation matrix, we have

$$PA = LU,$$

where L is defined as before, and $P = P^{(n-1)}P^{(n-2)}\cdots P^{(1)}$. This decomposition exists for *any* nonsingular matrix A.

Once the LU decomposition $PA = LU$ has been computed, we can solve the system $A\mathbf{x} = \mathbf{b}$ by first noting that if $\mathbf{x}$ is the solution, then

$$PA\mathbf{x} = LU\mathbf{x} = P\mathbf{b}.$$

Therefore, we can obtain $\mathbf{x}$ by first solving the system $L\mathbf{y} = P\mathbf{b}$, and then solving $U\mathbf{x} = \mathbf{y}$. Then, if $\mathbf{b}$ should change, then only these last two systems need to be solved in order to obtain the new solution; as in the case of Gaussian elimination without pivoting, the LU decomposition does not need to be recomputed.

Example Let

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 8 & 5 \\ 3 & 6 & 9 \end{bmatrix}.$$

Applying Gaussian elimination to A, we subtract twice the first row from the second, and three times the first row from the third, to obtain

$$A^{(2)} = \begin{bmatrix} 1 & 4 & 7 \\ 0 & 0 & -9 \\ 0 & -6 & -12 \end{bmatrix}.$$

At this point, Gaussian elimination breaks down, because the multiplier $m_{32} = a_{32}/a_{22} = -6/0$ is undefined.

Therefore, we must interchange the second and third rows, which yields the upper triangular matrix

$$U = A^{(3)} = P^{(2)}A^{(2)} = \begin{bmatrix} 1 & 4 & 7 \\ 0 & -6 & -12 \\ 0 & 0 & -9 \end{bmatrix},$$

where $P^{(2)}$ is the *permutation matrix*

$$P^{(2)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

obtained by interchanging the second and third rows of the identity matrix.

It follows that we have computed the factorization

$$PA = LU,$$

or

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 \\ 2 & 8 & 5 \\ 3 & 6 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 \\ 0 & -6 & -12 \\ 0 & 0 & -9 \end{bmatrix}.$$

It can be seen in advance that A does not have an LU factorization because the second minor of A, $a_{1:2,1:2}$, is a singular matrix. □

2.2.3.4 Practical Computation of Determinants, Revisited

When Gaussian elimination is used without pivoting to obtain the factorization $A = LU$, we have $\det(A) = \det(U)$, because $\det(L) = 1$ due to L being unit lower-triangular. When pivoting is used, we have the factorization $PA = LU$, where P is a permutation matrix. Because a permutation matrix is orthogonal; that is, $P^TP = I$, and $\det(A) = \det(A^T)$ for any square matrix, it follows that $\det(P)^2 = 1$, or $\det(P) = \pm 1$. Therefore, $\det(A) = \pm\det(U)$, where the sign depends on the sign of $\det(P)$.

To determine this sign, we note that when two rows (or columns) of A are interchanged, the sign of the determinant changes. Therefore, $\det(P) = (-1)^p$, where p is the number of row interchanges that are performed during Gaussian elimination. The number p is known as the *sign* of the permutation represented by P that determines the final ordering of the rows. We conclude that $\det(A) = (-1)^p \det(U)$.

2.3 Estimating and Improving Accuracy

2.3.1 The Condition Number

Let $A \in \mathbb{R}^{m\times n}$ be nonsingular, and let $A = U\Sigma V^T$ be the SVD of A. Then the solution $\mathbf{x}$ of the system of linear equations $A\mathbf{x} = \mathbf{b}$ can be expressed as

$$\mathbf{x} = A^{-1}\mathbf{b} = V\Sigma^{-1}U^T\mathbf{b} = \sum_{i=1}^{n} \frac{\mathbf{u}_i^T\mathbf{b}}{\sigma_i}\mathbf{v}_i.$$

This formula for $\mathbf{x}$ suggests that if σ_n is small relative to the other singular values, then the system $A\mathbf{x} = \mathbf{b}$ can be sensitive to perturbations in A or $\mathbf{b}$. This makes sense, considering that σ_n is the distance between A and the set of all singular $n \times n$ matrices.

In an attempt to measure the sensitivity of this system, we consider the parameterized system

$$(A + \epsilon E)\mathbf{x}(\epsilon) = \mathbf{b} + \epsilon\mathbf{e},$$

where $\mathbf{E} \in \mathbb{R}^{n\times n}$ and $\mathbf{e} \in \mathbb{R}^n$ are perturbations of A and $\mathbf{b}$, respectively. Taking the Taylor expansion of $\mathbf{x}(\epsilon)$ around $\epsilon = 0$ yields

$$\mathbf{x}(\epsilon) = \mathbf{x} + \epsilon\mathbf{x}'(0) + O(\epsilon^2),$$

where

$$\mathbf{x}'(\epsilon) = (A + \epsilon E)^{-1}(\mathbf{e} - E\mathbf{x}),$$

which yields $\mathbf{x}'(0) = A^{-1}(\mathbf{e} - E\mathbf{x})$.

Using norms to measure the relative error in $\mathbf{x}$, we obtain

$$\frac{\|\mathbf{x}(\epsilon) - \mathbf{x}\|}{\|\mathbf{x}\|} = |\epsilon|\frac{\|A^{-1}(\mathbf{e} - E\mathbf{x})\|}{\|\mathbf{x}\|} + O(\epsilon^2) \leq |\epsilon|\|A^{-1}\|\left\{\frac{\|\mathbf{e}\|}{\|\mathbf{x}\|} + \|E\|\right\} + O(\epsilon^2).$$

Multiplying and dividing by $\|A\|$, and using $A\mathbf{x} = \mathbf{b}$ to obtain $\|\mathbf{b}\| \leq \|A\|\|\mathbf{x}\|$, yields

$$\frac{\|\mathbf{x}(\epsilon) - \mathbf{x}\|}{\|\mathbf{x}\|} = \kappa(A)|\epsilon|\left(\frac{\|\mathbf{e}\|}{\|\mathbf{b}\|} + \frac{\|E\|}{\|A\|}\right),$$

where

$$\kappa(A) = \|A\|\|A^{-1}\|$$

is called the *condition number* of A. We conclude that the relative errors in A and $\mathbf{b}$ can be amplified by $\kappa(A)$ in the solution. Therefore, if $\kappa(A)$ is large, the problem $A\mathbf{x} = \mathbf{b}$ can be quite sensitive to perturbations in A and $\mathbf{b}$. In this case, we say that A is *ill-conditioned*; otherwise, we say that A is *well-conditioned.*

The definition of the condition number depends on the matrix norm that is used. Using the ℓ_2-norm, we obtain

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \frac{\sigma_1(A)}{\sigma_n(A)}.$$

It can readily be seen from this formula that $\kappa_2(A)$ is large if σ_n is small relative to σ_1. We also note that because the singular values are the lengths of the semi-axes of the hyperellipsoid $\{A\mathbf{x} \mid \|\mathbf{x}\|_2 = 1\}$, the condition number in the ℓ_2-norm measures the elongation of this hyperellipsoid.

Example The matrices

$$A_1 = \begin{bmatrix} 0.7674 & 0.0477 \\ 0.6247 & 0.1691 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 0.7581 & 0.1113 \\ 0.6358 & 0.0933 \end{bmatrix}$$

do not appear to be very different from one another, but $\kappa_2(A_1) = 10$ while $\kappa_2(A_2) = 10^{10}$. That is, A_1 is well-conditioned while A_2 is ill-conditioned.

To illustrate the ill-conditioned nature of A_2, we solve the two systems of equations $A_2\mathbf{x}_1 = \mathbf{b}_1$ and $A_2\mathbf{x}_2 = \mathbf{b}_2$ for the unknown vectors $\mathbf{x}_1$ and $\mathbf{x}_2$, where

$$\mathbf{b}_1 = \begin{bmatrix} 0.7662 \\ 0.6426 \end{bmatrix}, \quad \mathbf{b}_2 = \begin{bmatrix} 0.7019 \\ 0.7192 \end{bmatrix}.$$

These vectors differ from one another by roughly 10%, but the solutions

$$\mathbf{x}_1 = \begin{bmatrix} 0.9894 \\ 0.1452 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} -1.4522 \times 10^8 \\ 9.8940 \times 10^8 \end{bmatrix}$$

differ by several orders of magnitude, because of the sensitivity of A_2 to perturbations. □

Just as the largest singular value of A is the ℓ_2-norm of A, and the smallest singular value is the distance from A to the nearest singular matrix in the ℓ_2-norm, we have, for any ℓ_p-norm,

$$\frac{1}{\kappa_p(A)} = \min_{A+\Delta A \,\text{singular}} \frac{\|\Delta A\|_p}{\|A\|_p}.$$

That is, in any ℓ_p-norm, $\kappa_p(A)$ measures the relative distance in that norm from A to the set of singular matrices.

Because $\det(A) = 0$ if and only if A is singular, it would appear that the determinant could be used to measure the distance from A to the nearest singular matrix. However, this is generally not the case. It is possible for a matrix to have a relatively large determinant, but be very close to a singular matrix, or for a matrix to have a relatively small determinant, but not be nearly singular. In other words, there is very little correlation between $\det(A)$ and the condition number of A.

Example Let

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

Then $\det(A) = 1$, but $\kappa_2(A) \approx 1,918$, and $\sigma_{10} \approx 0.0029$. That is, A is quite close to a singular matrix, even though $\det(A)$ is not near zero. For example, the nearby matrix $\tilde{A} = A - \sigma_{10}\mathbf{u}_{10}\mathbf{v}_{10}^T$, whose entries are equal to those of A to within two decimal places, is singular. □

2.3.2 Iterative Refinement

Although we have learned about solving a system of linear equations $A\mathbf{x} = \mathbf{b}$. we have yet to discuss methods of estimating the error in a computed solution $\tilde{\mathbf{x}}$. A simple approach to judging the accuracy of $\tilde{\mathbf{x}}$ is to compute the *residual* vector $\mathbf{r} = \mathbf{b} - A\tilde{\mathbf{x}}$, and then compute the magnitude of $\mathbf{r}$ using any vector norm. However, this approach can be misleading, as a small residual does not necessarily imply that the *error* in the solution, which is $\mathbf{e} = \mathbf{x} - \tilde{\mathbf{x}}$, is small.

To see this, we first note that

$$A\mathbf{e} = A(\mathbf{x} - \tilde{\mathbf{x}}) = A\mathbf{x} - A\tilde{\mathbf{x}} = \mathbf{b} - A\tilde{\mathbf{x}} = \mathbf{r}.$$

It follows that for any vector norm $\|\cdot\|$, and the corresponding induced matrix norm, we have

$$\begin{aligned}
\|\mathbf{e}\| &= \|A^{-1}\mathbf{r}\| \\
&\leq \|A^{-1}\|\|\mathbf{r}\| \\
&\leq \|A^{-1}\|\|\mathbf{r}\|\frac{\|\mathbf{b}\|}{\|\mathbf{b}\|} \\
&\leq \|A^{-1}\|\|\mathbf{r}\|\frac{\|A\mathbf{x}\|}{\|\mathbf{b}\|} \\
&\leq \|A\|\|A^{-1}\|\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}\|\mathbf{x}\|.
\end{aligned}$$

We conclude that the magnitude of the *relative error* in $\tilde{\mathbf{x}}$ is bounded as follows:

$$\frac{\|\mathbf{e}\|}{\|\mathbf{x}\|} \leq \kappa(A)\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|},$$

where

$$\kappa(A) = \|A\|\|A^{-1}\|$$

is the condition number of A. Therefore, it is possible for the residual to be small, and the error to still be large.

We can exploit the relationship between the error $\mathbf{e}$ and the residual $\mathbf{r}$, $A\mathbf{e} = \mathbf{r}$, to obtain an estimate of the error, $\tilde{\mathbf{e}}$, by solving the system $A\mathbf{e} = \mathbf{r}$ in the same manner in which we obtained $\tilde{\mathbf{x}}$ by attempting to solve $A\mathbf{x} = \mathbf{b}$.

Since $\tilde{\mathbf{e}}$ is an estimate of the error $\mathbf{e} = \mathbf{x} - \tilde{\mathbf{x}}$ in $\tilde{\mathbf{x}}$, it follows that $\tilde{\mathbf{x}} + \tilde{\mathbf{e}}$ is a more accurate approximation of $\mathbf{x}$ than $\tilde{\mathbf{x}}$ is. This is the basic idea behind *iterative refinement*, also known as *iterative improvement* or *residual correction*. The algorithm is as follows:

```
Choose a desired accuracy level (error tolerance) $TOL$
$\tilde{\mathbf{x}}^{(0)} = \mathbf{0}$
$\mathbf{r}^{(0)} = \mathbf{b}$
for $k = 0, 1, 2, \ldots$
    Solve $A\tilde{\mathbf{e}}^{(k)} = \mathbf{r}^{(k)}$
    if $\|\tilde{\mathbf{e}}^{(k)}\|_\infty < TOL$
        break
    end
    $\tilde{\mathbf{x}}^{(k+1)} = \tilde{\mathbf{x}}^{(k)} + \tilde{\mathbf{e}}^{(k)}$
    $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - A\tilde{\mathbf{e}}^{(k)}$
end
```

The algorithm repeatedly applies the relationship $A\mathbf{e} = \mathbf{r}$, where $\mathbf{e}$ is the error and $\mathbf{r}$ is the residual, to update the computed solution with an estimate of its error. For this algorithm to be effective, it is important that the residual $\tilde{\mathbf{r}}^{(k)}$ be computed as accurately as possible, for example using higher-precision arithmetic than for the rest of the computation.

It can be shown that if the vector $\mathbf{r}^{(k)}$ is computed using double or extended precision that $\mathbf{x}^{(k)}$ converges to a solution where almost all digits are correct when $\kappa(A)\mathbf{u} \leq 1$.

It is important to note that in the above algorithm, the new residual $\mathbf{r}^{(k+1)}$ is computed using the formula $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - A\tilde{\mathbf{e}}^{(k)}$, rather than the definition $\mathbf{r}^{(k+1)} = \mathbf{b} - A\mathbf{x}^{(k+1)}$. To see that these formulas are equivalent, we use the definition of $\mathbf{x}^{(k+1)}$ to obtain

$$\begin{aligned}
\mathbf{r}^{(k+1)} &= \mathbf{b} - A\tilde{\mathbf{x}}^{(k+1)} \\
&= \mathbf{b} - A(\tilde{\mathbf{x}}^{(k)} + \tilde{\mathbf{e}}^{(k)}) \\
&= \mathbf{b} - A\tilde{\mathbf{x}}^{(k)} - A\tilde{\mathbf{e}}^{(k)} \\
&= \mathbf{r}^{(k)} - A\tilde{\mathbf{e}}^{(k)}.
\end{aligned}$$

This formula is preferable to the definition because as k increases, both $\mathbf{r}^{(k)}$ and $\tilde{\mathbf{e}}^{(k)}$ should approach the zero vector, and therefore smaller vectors than $\mathbf{b}$ and $A\tilde{\mathbf{x}}^{(k+1)}$ will be subtracted to obtain $\mathbf{r}^{(k+1)}$, thus reducing the amount of cancellation error that occurs.

2.3.3 Scaling and Equilibration

As we have seen, the bounds for the error depend on $\kappa(A) = \|A\|\|A^{-1}\|$. Perhaps we can re-scale the equations so that the condition number is changed. We replace the system

$$A\mathbf{x} = \mathbf{b}$$

by the equivalent system

$$DA\mathbf{x} = D\mathbf{b}$$

or possibly

$$DAE\mathbf{y} = D\mathbf{b}$$

where D and E are diagonal matrices and $y = E^{-1}\mathbf{x}$.

Suppose A is *symmetric positive definite*; that is, $A = A^T$ and $\mathbf{x}^T A\mathbf{x} > 0$ for any nonzero vector $\mathbf{x}$. We want to replace A by DAD, that is, replace a_{ij} by $d_i d_j a_{ij}$, so that $\kappa(DAD)$ is minimized.

It turns out that for a class of symmetric matrices, such minimization is possible. A symmetric positive definite matrix A is said to have *Property A* if there exists a permutation matrix Π such that

$$\Pi A \Pi^T = \begin{bmatrix} D & F \\ F^T & D \end{bmatrix},$$

where D is a diagonal matrix. For example, all tridiagonal matrices that are symmetric positive definite have Property A.

For example, suppose

$$A = \begin{bmatrix} 50 & 7 \\ 7 & 1 \end{bmatrix}.$$

Then $\lambda_{\max} \approx 51$ and $\lambda_{\min} \approx 1/51$, which means that $\kappa(A) \approx 2500$. However,

$$DAD = \begin{bmatrix} \frac{1}{\sqrt{50}} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 50 & 7 \\ 7 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{50}} & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & \frac{7}{\sqrt{50}} \\ \frac{7}{\sqrt{50}} & 1 \end{bmatrix}$$

and

$$\kappa = \frac{1 + \frac{7}{\sqrt{50}}}{1 - \frac{7}{\sqrt{50}}} \approx 200.$$

One scaling strategy is called *equilibration.* The idea is to set $A^{(0)} = A$ and compute $A^{(1/2)} = D^{(1)}A^{(0)} = \{d_i^{(1)}a_{ij}\}$, choosing the diagonal matrix $D^{(1)}$ so that $d_i^{(1)} \sum_{j=1}^{n} |a_{ij}^{(0)}| = 1$. That is, all *row sums* of $|D^{(1)}A^{(0)}|$ are equal to one. Then, we compute $A^{(1)} = A^{(1/2)}E^{(1)} = \{a_{ij}^{(1/2)}e_j^{(1)}\}$, choosing each element of the diagonal matrix $E^{(1)}$ so that $e_j^{(1)} \sum_{i=1}^{n} |a_{ij}^{(1/2)}| = 1$. That is, all *column sums* of $|A^{(1/2)}E^{(1)}|$ are equal to one. We then repeat this process, which yields

$$\begin{aligned} A^{(k+1/2)} &= D^{(k+1)}A^{(k)}, \\ A^{(k+1)} &= A^{(k+1/2)}E^{(k+1)}. \end{aligned}$$

Under very general conditions, the $A^{(k)}$ converge to a matrix whose row and column sums are all equal.

2.4 Special Matrices

2.4.1 Banded Matrices

We now take a look at different types of special matrices. The types of matrices you will see in this section have special properties that often make them easier to solve than a general matrix. The first type of special matrix we will discuss is called a banded matrix. An $n \times n$ matrix A is said to have *upper bandwidth* p if $a_{ij} = 0$ whenever $j - i > p$. Let's take a moment to visualize this scenario.

$$B = \begin{bmatrix} -2 & 1 & 5 & 0 & 0 \\ 1 & 1 & 8 & 4 & 0 \\ 0 & 4 & 3 & -2 & 7 \\ 0 & 0 & 11 & 1 & 1 \\ 0 & 0 & 0 & 3 & -2 \end{bmatrix},$$

The above matrix would have an upper bandwidth of $p = 2$ since $a_{ij} = 0$ whenever $i = 1, j = 4$; therefore, $4 - 1 > 2$. Similarly, A has *lower bandwidth* q if $a_{ij} = 0$ whenever $i - j > q$. A matrix that has upper bandwidth p and lower bandwidth q is said to have *bandwidth* $w = p + q + 1$. The above matrix has a lower bandwidth of $q = 1$ since $a_{ij} = 0$ whenever $i = 3, j = 1$; therefore, $3 - 1 > 1$.

Any $n \times n$ matrix A has a bandwidth $w \leq 2n - 1$. If $w < 2n - 1$, then A is said to be *banded.* However, cases in which the bandwidth is $O(1)$, such as when A is a *tridiagonal* matrix for which $p = q = 1$, are of particular interest because for such matrices, Gaussian elimination, forward substitution and back substitution are much more efficient.

Exercise 2.4.1 *(a) If A has lower bandwidth q, and $A = LU$ is the LU decomposition of A (without pivoting), then what is the lower bandwidth of the lower triangular matrix L?*

(b) How many elements, at most, need to be eliminated per column?

(c) If A has upper bandwidth p, and $A = LU$ is the LU decomposition of A (without pivoting), then what is the upper bandwidth of the upper triangular matrix U?

(d) How many elements, at most, need to be eliminated per column?

Exercise 2.4.2 *If A has $O(1)$ bandwidth, then how many FLOPS do Gaussian elimination, forward substitution and back substitution require?*

Example The matrix

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

which arises from discretization of the second derivative operator, is banded with lower bandwith and upper bandwith 1, and total bandwidth 3. Its LU factorization is

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

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 & 0 \\ 0 & -\frac{2}{3} & 1 & 0 & 0 \\ 0 & 0 & -\frac{3}{4} & 1 & 0 \\ 0 & 0 & 0 & -\frac{4}{5} & 1 \end{bmatrix} \begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 0 & -\frac{3}{2} & 1 & 0 & 0 \\ 0 & 0 & -\frac{4}{3} & 1 & 0 \\ 0 & 0 & 0 & -\frac{5}{4} & 1 \\ 0 & 0 & 0 & 0 & -\frac{6}{5} \end{bmatrix}.$$

We see that L has lower bandwith 1, and U has upper bandwith 1. □

Exercise 2.4.3 *(a) Write a* MATLAB *function to find the LU factorization of a tridiagonal matrix.*

(b) Now write a MATLAB *function for any banded matrix with bandwidth w.*

(c) How do the two functions differ in performance? How many FLOPS do each require?

When a matrix A is banded with bandwidth w, it is wasteful to store it in the traditional 2-dimensional array. Instead, it is much more efficient to store the elements of A in w vectors of length at most n. Then, the algorithms for Gaussian elimination, forward substitution and back substitution can be modified appropriately to work with these vectors. For example, to perform Gaussian elimination on a tridiagonal matrix, we can proceed as in the following algorithm. We assume that the main diagonal of A is stored in the vector $\mathbf{a}$, the *subdiagonal* (entries $a_{j+1,j}$) is stored in the vector $\mathbf{l}$, and the *superdiagonal* (entries $a_{j,j+1}$) is stored in the vector $\mathbf{u}$.

```
for $j = 1, 2, \ldots, n-1$ do
    $l_j = l_j / a_j$
    $a_{j+1} = a_{j+1} - l_j u_j$
end
```

Notice that this algorithm is much shorter than regular Gaussian Elimination. That is because the number of operations for solving a tridiagonal system is significantly reduced.

back subtitution as follows:

```
$y_1 = b_1$
for $i = 2, 3, \ldots, n$ do
    $y_i = b_i - l_{i-1} y_{i-1}$
end
$x_n = y_n / a_n$
for $i = n-1, n-2, \ldots, 1$ do
    $x_i = (y_i - u_i x_{i+1}) / a_i$
end
```

After Gaussian elimination, the components of the vector $\mathbf{l}$ are the subdiagonal entries of L in the LU decomposition of A, and the components of the vector $\mathbf{u}$ are the superdiagonal entries of U.

Pivoting can cause difficulties for banded systems because it can cause *fill-in*: the introduction of nonzero entries outside of the band. For this reason, when pivoting is necessary, pivoting schemes that offer more flexibility than partial pivoting are typically used. The resulting trade-off is that the entries of L are permitted to be somewhat larger, but the sparsity (that is, the occurrence of zero entries) of A is preserved to a greater extent.

2.4.2 Symmetric Matrices

2.4.2.1 The LDL^T Factorization

Suppose that A is a nonsingular $n \times n$ matrix that has an LU factorization $A = LU$. We know that L is unit lower-triangular, and U is upper triangular. If we define the diagonal matrix D by

$$D = \begin{bmatrix} u_{11} & 0 & \cdots & 0 \\ 0 & u_{22} & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{bmatrix} = \operatorname{diag}(u_{11}, u_{22}, \ldots, u_{nn}),$$

then D is also nonsingular, and then the matrix $D^{-1}U$, which has entries

$$[D^{-1}U]_{ij} = \frac{u_{ij}}{u_{ii}}, \quad i, j = 1, 2, \ldots, n.$$

The diagonal entries of this matrix are equal to one, and therefore $D^{-1}U$ is *unit upper-triangular*.

Therefore, if we define the matrix M by $M^T = D^{-1}U$, then we have the factorization

$$A = LU = LDD^{-1}U = LDM^T,$$

where *both* L and M are unit lower-triangular, and D is diagonal. This is called the LDM^T factorization of A.

Because of the close connection between the LDM^T factorization and the LU factorization, the LDM^T factorization is not normally used in practice for solving the system $A\mathbf{x} = \mathbf{b}$ for a general nonsingular matrix A. However, this factorization becomes much more interesting when A is symmetric.

If $A = A^T$, then $LDM^T = (LDM^T)^T = MD^TL^T = MDL^T$, because D, being a diagonal matrix, is also symmetric. Because L and M, being unit lower-triangular, are nonsingular, it follows that

$$M^{-1}LD = DL^TM^{-T} = D(M^{-1}L)^T.$$

The matrix $M^{-1}L$ is unit lower-triangular. Therefore, the above equation states that a lower-triangular matrix is equal to an upper-triangular matrix, which implies that both matrices must be diagonal. It follows that $M^{-1}L = I$, because its diagonal entries are already known to be equal to one.

We conclude that $L = M$, and thus we have the LDL^T factorization

$$A = LDL^T.$$

Let's take a look at the factorization of the symmetric matrix

Example 2.4.1 Example

$$S = \begin{bmatrix} 4 & 2 \\ 2 & 3 \end{bmatrix}.$$

The first step would be to subtract half of row one from row 2.

$$S = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 4 & 2 \\ 0 & 2 \end{bmatrix}$$

The symmetry has been lost, but can be regained by finishing the factorization.

$$S = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{2} \\ 0 & 1 \end{bmatrix}$$

From the above example we see that whenever we have a symmetric matrix we have the factorization LDU where L is lower unit triangular, D is diagonal, and U is upper unit triangular. Note that $U = L^T$, and therefore we have the factorization LDL^T. This factorization is quite economical, compared to the LU and LDM^T factorizations, because only $n(n+1)/2$ entries are needed to represent L and D. Once these factors are obtained, we can solve $A\mathbf{x} = \mathbf{b}$ by solving the simple systems

$$L\mathbf{y} = \mathbf{b}, \quad D\mathbf{z} = \mathbf{y}, \quad L^T\mathbf{x} = \mathbf{z},$$

using forward substitution, simple divisions, and back substitution.

The LDL^T factorization can be obtained by performing Gaussian elimination, but this is not efficient, because Gaussian elimination requires performing operations on entire rows of A, which does not exploit symmetry. This can be addressed by omitting updates of the upper-triangular portion of A, as they do not influence the computation of L and D. An alternative approach, that is equally efficient in terms of the number of floating-point operations, but more desirable overall due to its use of vector operations, involves computing L column-by-column.

If we multiply both sides of the equation $A = LDL^T$ by the standard basis vector $\mathbf{e}_j$ to extract the jth column of this matrix equation, we obtain

$$\mathbf{a}_j = \sum_{k=1}^{j} \ell_k v_{kj},$$

where

$$A = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{bmatrix}, \quad L = \begin{bmatrix} \ell_1 & \cdots & \ell_n \end{bmatrix}$$

are column partitions of A and L, and $\mathbf{v}_j = DL^T\mathbf{e}_j$.

Suppose that columns $1, 2, \ldots, j-1$ of L, as well as $d_{11}, d_{22}, \ldots, d_{j-1,j-1}$, the first $j-1$ diagonal elements of D, have already been computed. Then, we can compute $v_{kj} = d_{kk}\ell_{jk}$ for $k = 1, 2, \ldots, j-1$, because these quantities depend on elements of L and D that are available. It follows that

$$\mathbf{a}_j - \sum_{k=1}^{j-1} \ell_k v_{kj} = \ell_j v_{jj} = \ell_j d_{jj} \ell_{jj}.$$

However, $\ell_{jj} = 1$, which means that we can obtain d_{jj} from the jth component of the vector

$$\mathbf{u}_j = \mathbf{a}_j - \sum_{k=1}^{j-1} \ell_k v_{kj},$$

and then obtain the "interesting" portion of the new column ℓ_j, that is, entries $j : n$, by computing $\ell_j = \mathbf{u}_j / d_{jj}$. The remainder of this column is zero, because L is lower-triangular.

Exercise 2.4.4 *Write a* MATLAB *function that implements the following LDL^T algorithm. This algorithm should take the random symmetric matrix A as input, and return L and D as output. Check to see if $A = LDL^T$.*

The entire algorithm proceeds as follows:

```
$L = 0$
$D = 0$
for $j = 1 : n$ do
    for $k = 1 : j - 1$ do
        $v_{kj} = d_{kk}\ell_{jk}$
    end
    $\mathbf{u}_j = \mathbf{a}_{j:n,j}$
    for $k = 1 : j - 1$ do
        $\mathbf{u}_j = \mathbf{u}_j - \ell_{j:n,k}v_{kj}$
    end
    $d_{jj} = u_{1j}$
    $\ell_{j:n,j} = \mathbf{u}_j / d_{jj}$
end
```

This algorithm requires approximately $\frac{1}{3}n^3$ floating-point operations, which is half as many as Gaussian elimination. If pivoting is required, then we obtain a factorization of the form $PA = LDL^T$. However, we will soon see that for an important class of symmetric matrices, pivoting is unnecessary.

2.4.3 Symmetric Positive Definite Matrices

In the last section we looked at solving symmetric systems where $A = A^T$. Now we still consider the same symmetric systems, but we consider the special case where all the elements of D of $A = LDL^T$ are positive. This type of matrix has the following properties:

2.4.3.1 Properties

A real, $n \times n$ symmetric matrix A is *symmetric positive definite* if $A = A^T$ and, for any nonzero vector $\mathbf{x}$,

$$\mathbf{x}^T A\mathbf{x} > 0.$$

Exercise 2.4.5 *Show that if matrices A and B are positive definite, then $A+B$ is positive definite.*

A symmetric positive definite matrix is the generalization to $n \times n$ matrices of a positive number. If A is symmetric positive definite, then it has the following properties:

- A is nonsingular; in fact, $\det(A) > 0$.
- All of the diagonal elements of A are positive.
- The largest element of the matrix lies on the diagonal.
- All of the eigenvalues of A are positive.

In general it is not easy to determine whether a given $n \times n$ symmetric matrix A is also positive definite. One approach is to check the matrices

$$A_k = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} \end{bmatrix}, \quad k = 1, 2, \ldots, n,$$

which are the *leading principal submatrices* of A.

Exercise 2.4.6 *Show that A is positive definite if and only if $\det(A_k) > 0$ for $k = 1, 2, \ldots, n$.*

There are other types of matrices with all the same properties as above except zeros are allowed. These matrices are defined as follows: *negative definite*, where $\mathbf{x}^T A\mathbf{x} < 0$; *positive semi-definite*, where $\mathbf{x}^T A\mathbf{x} \geq 0$; and *negative semi-definite*, where $\mathbf{x}^T A\mathbf{x} \leq 0$.

Exercise 2.4.7 *Find the values of c for which the following matrix is*

(a) positive definite

(b) positive semi-definite

(c) negative definite

(d) negative semi-definite

$$\begin{bmatrix} 3 & -1 & c \\ -1 & 3 & -1 \\ c & -1 & 3 \end{bmatrix}$$

One desirable property of symmetric positive definite matrices is that Gaussian elimination can be performed on them without pivoting, and all pivot elements are positive. Furthermore, Gaussian elimination applied to such matrices is robust with respect to the accumulation of roundoff error. However, Gaussian elimination is not the most practical approach to solving systems of linear equations involving symmetric positive definite matrices, because it is not the most efficient approach in terms of the number of floating-point operations that are required.

2.4.3.2 The Cholesky Factorization

Instead, it is preferable to compute the *Cholesky factorization* of A,

$$A = GG^T,$$

where G is a lower triangular matrix with positive diagonal entries. Because A is factored into two matrices that are the transpose of one another, the process of computing the Cholesky factorization requires about half as many operations as the LU decomposition.

The algorithm for computing the Cholesky factorization can be derived by matching entries of GG^T with those of A. This yields the following relation between the entries of G and A,

$$a_{ik} = \sum_{j=1}^{k} g_{ij} g_{kj}, \quad i, k = 1, 2, \ldots, n, \quad i \geq k.$$

From this relation, we obtain the following algorithm.

```
for $j = 1, 2, \ldots, n$ do
    $g_{jj} = \sqrt{a_{jj}}$
    for $i = j + 1, j + 2, \ldots, n$ do
        $g_{ij} = a_{ij}/g_{jj}$
        for $k = j + 1, \ldots, i$ do
            $a_{ik} = a_{ik} - g_{ij} g_{kj}$
        end
    end
end
```

The innermost loop subtracts off all terms but the last (corresponding to $j = k$) in the above summation that expresses a_{ik} in terms of entries of G. Equivalently, for each j, this loop subtracts the matrix $\mathbf{g}_j \mathbf{g}_j^T$ from A, where $\mathbf{g}_j$ is the jth column of G. Note that based on the outer product view of matrix multiplication, the equation $A = GG^T$ is equivalent to

$$A = \sum_{j=1}^{n} \mathbf{g}_j \mathbf{g}_j^T.$$

Therefore, for each j, the contributions of all columns $\mathbf{g}_\ell$ of G, where $\ell < j$, have already been subtracted from A, thus allowing column j of G to easily be computed by the steps in the outer loops, which account for the last term in the summation for a_{ik}, in which $j = k$.

Exercise 2.4.8 *Write a* MATLAB *function that performs the Cholesky factorization* $A = GG^T$*. Have your function:*

(a) take the matrix symmetric matrix A *as input,*

(b) return variable isposdef that checks to see if the matrix A *is positive definite, and*

(c) return the lower triangular matrix G *as output.*

Exercise 2.4.9 *How many FLOPs does this algorithm require?*

(a) Use the MATLAB *commands 'tic' and 'toc' for when the size of the matrix is double. Compare calculation times for these matrices, and try using large matrices.*

(b) Count how many FLOPs are performed in your implementation of the Cholesky algorithm.

Example 2.4.2 Example *Let*

$$A = \begin{bmatrix} 9 & -3 & 3 & 9 \\ -3 & 17 & -1 & -7 \\ 3 & -1 & 17 & 15 \\ 9 & -7 & 15 & 44 \end{bmatrix}.$$

A is a symmetric positive definite matrix. To compute its Cholesky decomposition $A = GG^T$, we equate entries of A to those of GG^T, which yields the matrix equation

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} g_{11} & 0 & 0 & 0 \\ g_{21} & g_{22} & 0 & 0 \\ g_{31} & g_{32} & g_{33} & 0 \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix} \begin{bmatrix} g_{11} & g_{21} & g_{31} & g_{41} \\ 0 & g_{22} & g_{32} & g_{42} \\ 0 & 0 & g_{33} & g_{43} \\ 0 & 0 & 0 & g_{44} \end{bmatrix},$$

and the equivalent scalar equations

$$\begin{aligned} a_{11} &= g_{11}^2, \\ a_{21} &= g_{21}g_{11}, \\ a_{31} &= g_{31}g_{11}, \\ a_{41} &= g_{41}g_{11}, \\ a_{22} &= g_{21}^2 + g_{22}^2, \\ a_{32} &= g_{31}g_{21} + g_{32}g_{22}, \\ a_{42} &= g_{41}g_{21} + g_{42}g_{22}, \\ a_{33} &= g_{31}^2 + g_{32}^2 + g_{33}^2, \\ a_{43} &= g_{41}g_{31} + g_{42}g_{32} + g_{43}g_{33}, \\ a_{44} &= g_{41}^2 + g_{42}^2 + g_{43}^2 + g_{44}^2. \end{aligned}$$

We compute the nonzero entries of G one column at a time. For the first column, we have

$$\begin{aligned} g_{11} &= \sqrt{a_{11}} = \sqrt{9} = 3, \\ g_{21} &= a_{21}/g_{11} = -3/3 = -1, \\ g_{31} &= a_{31}/g_{11} = 3/3 = 1, \\ g_{41} &= a_{41}/g_{11} = 9/3 = 3. \end{aligned}$$

Before proceeding to the next column, we first subtract all contributions to the remaining entries of A from the entries of the first column of G. That is, we update A as follows:

$$\begin{aligned} a_{22} &= a_{22} - g_{21}^2 = 17 - (-1)^2 = 16, \\ a_{32} &= a_{32} - g_{31}g_{21} = -1 - (1)(-1) = 0, \\ a_{42} &= a_{42} - g_{41}g_{21} = -7 - (3)(-1) = -4, \\ a_{33} &= a_{33} - g_{31}^2 = 17 - 1^2 = 16, \\ a_{43} &= a_{43} - g_{41}g_{31} = 15 - (3)(1) = 12, \\ a_{44} &= a_{44} - g_{41}^2 = 44 - 3^2 = 35. \end{aligned}$$

Now, we can compute the nonzero entries of the second column of G just as for the first column:

$$\begin{aligned} g_{22} &= \sqrt{a_{22}} = \sqrt{16} = 4, \\ g_{32} &= a_{32}/g_{22} = 0/4 = 0, \\ g_{42} &= a_{42}/g_{22} = -4/4 = -1. \end{aligned}$$

We then remove the contributions from G's second column to the remaining entries of A:

$$\begin{aligned} a_{33} &= a_{33} - g_{32}^2 = 16 - 0^2 = 16, \\ a_{43} &= a_{43} - g_{42}g_{32} = 12 - (-1)(0) = 12, \\ a_{44} &= a_{44} - g_{42}^2 = 35 - (-1)^2 = 34. \end{aligned}$$

The nonzero portion of the third column of G is then computed as follows:

$$\begin{aligned} g_{33} &= \sqrt{a_{33}} = \sqrt{16} = 4, \\ g_{43} &= a_{43}/g_{43} = 12/4 = 3. \end{aligned}$$

Finally, we compute g_{44}:

$$a_{44} = a_{44} - g_{43}^2 = 34 - 3^2 = 25, \quad g_{44} = \sqrt{a_{44}} = \sqrt{25} = 5.$$

Thus the complete Cholesky factorization of A is

$$\begin{bmatrix} 9 & -3 & 3 & 9 \\ -3 & 17 & -1 & -7 \\ 3 & -1 & 17 & 15 \\ 9 & -7 & 15 & 44 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ -1 & 4 & 0 & 0 \\ 1 & 0 & 4 & 0 \\ 3 & -1 & 3 & 5 \end{bmatrix} \begin{bmatrix} 3 & -1 & 1 & 3 \\ 0 & 4 & 0 & -1 \\ 0 & 0 & 4 & 3 \\ 0 & 0 & 0 & 5 \end{bmatrix}.$$

□

If A is not symmetric positive definite, then the algorithm will break down, because it will attempt to compute g_{jj}, for some j, by taking the square root of a negative number, or divide by a zero g_{jj}.

Example 2.4.3 Example *The matrix*

$$A = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}$$

is symmetric but not positive definite, because $\det(A) = 4(2) - 3(3) = -1 < 0$*. If we attempt to compute the Cholesky factorization* $A = GG^T$*, we have*

$$\begin{aligned} g_{11} &= \sqrt{a_{11}} = \sqrt{4} = 2, \\ g_{21} &= a_{21}/g_{11} = 3/2, \\ a_{22} &= a_{22} - g_{21}^2 = 2 - 9/4 = -1/4, \\ g_{22} &= \sqrt{a_{22}} = \sqrt{-1/4}, \end{aligned}$$

and the algorithm breaks down. □

In fact, due to the expense involved in computing determinants, the Cholesky factorization is also an efficient method for checking whether a symmetric matrix is also positive definite. Once the Cholesky factor G *of* A *is computed, a system* $A\mathbf{x} = \mathbf{b}$ *can be solved by first solving* $G\mathbf{y} = \mathbf{b}$ *by forward substitution, and then solving* $G^T\mathbf{x} = \mathbf{y}$ *by back substitution.*

This is similar to the process of solving $A\mathbf{x} = \mathbf{b}$ using the LDL^T factorization, except that there is no diagonal system to solve. In fact, the LDL^T factorization is also known as the "square-root-free Cholesky factorization", since it computes factors that are similar in structure to the Cholesky factors, but without computing any square roots. Specifically, if $A = GG^T$ is the Cholesky factorization of A, then $G = LD^{1/2}$. As with the LU factorization, the Cholesky factorization is unique, because the diagonal is required to be positive.

2.5 Iterative Methods

Given a system of n linear equations in n unknowns, described by the matrix-vector equation

$$A\mathbf{x} = \mathbf{b},$$

where A is an invertible $n \times n$ matrix, we can obtain the solution using a *direct method* such as Gaussian elimination in conjunction with forward and back substitution. However, there are several drawbacks to this approach:

- If we have an approximation to the solution $\mathbf{x}$, a direct method does not provide any means of taking advantage of this information to reduce the amount of computation required.
- If we only require an approximate solution, rather than the exact solution except for roundoff error, it is not possible to terminate the algorithm for a direct method early in order to obtain such an approximation.

- If the matrix A is sparse, Gaussian elimination or similar methods can cause *fill-in*, which is the introduction of new nonzero elements in the matrix, thus reducing efficiency.

- In some cases, A may not be represented as a two-dimensional array or a set of vectors; instead, it might be represented implicitly by a function that returns as output the matrix-vector product $A\mathbf{x}$, given the vector $\mathbf{x}$ as input. A direct method is not practical for such a representation, because the individual entries of A are not readily available.

For this reason, it is worthwhile to consider alternative approaches to solving $A\mathbf{x} = \mathbf{b}$, such as *iterative methods*. In particular, an iterative method based on matrix-vector multiplication will address all of these drawbacks of direct methods.

There are two general classes of iterative methods: *stationary* iterative methods and *non-stationary* methods. Either type of method accepts as input an initial guess $\mathbf{x}^{(0)}$ (usually chosen to be the zero vector) and computes a sequence of iterates $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$, $\ldots$, that, hopefully, converges to the solution $\mathbf{x}$. A stationary method has the form

$$\mathbf{x}^{(k+1)} = g(\mathbf{x}^{(k)}),$$

for some function $g : \mathbb{R}^n \to \mathbb{R}^n$. The solution $\mathbf{x}$ is a *fixed point*, or *stationary point*, of g. In other words, a stationary iterative method is one in which *fixed-point iteration*, which we have previously applied to solve nonlinear equations, is used to obtain the solution.

2.5.1 Stationary Iterative Methods

To construct a suitable function g, we compute a *splitting* of the matrix $A = M - N$, where M is nonsingular. Then, the solution $\mathbf{x}$ satisfies

$$M\mathbf{x} = N\mathbf{x} + \mathbf{b},$$

or

$$\mathbf{x} = M^{-1}(N\mathbf{x} + \mathbf{b}).$$

We therefore define

$$g(\mathbf{x}) = M^{-1}(N\mathbf{x} + \mathbf{b}),$$

so that the iteration takes the form

$$M\mathbf{x}^{(k+1)} = N\mathbf{x}^{(k)} + \mathbf{b}.$$

It follows that for the sake of efficiency, the splitting $A = M - N$ should be chosen so that the system $M\mathbf{y} = \mathbf{c}$ is easily solved.

2.5.1.1 Convergence Analysis

Before we describe specific splittings, we examine the convergence of this type of iteration. Using the fact that $\mathbf{x}$ is the exact solution of $A\mathbf{x} = \mathbf{b}$, we obtain

$$M(\mathbf{x}^{(k+1)} - \mathbf{x}) = N(\mathbf{x}^{(k)} - \mathbf{x}) + \mathbf{b} - \mathbf{b},$$

which yields

$$\mathbf{x}^{(k+1)} - \mathbf{x} = M^{-1}N(\mathbf{x}^{(k)} - \mathbf{x})$$

and

$$\mathbf{x}^{(k)} - \mathbf{x} = (M^{-1}N)^k(\mathbf{x}^{(0)} - \mathbf{x}).$$

That is, the error after each iteration is obtained by multiplying the error from the previous iteration by $T = M^{-1}N$. Therefore, in order for the error to converge to the zero vector, for any choice of the initial guess $\mathbf{x}^{(0)}$, we must have $\rho(T) < 1$, where $\rho(T)$ is the *spectral radius* of T. Let $\lambda_1 \ldots \lambda_n$ be the eigenvalues of a matrix A. Then $\rho(A) = max\,\{\mid \lambda_1 \mid \ldots \mid \lambda_n \mid\}$.

2.5.1.2 Jacobi Method

We now discuss some basic stationary iterative methods. For convenience, we write

$$A = D + L + U,$$

where D is a diagonal matrix whose diagonal entries are the diagonal entries of A, L is a strictly lower triangular matrix defined by

$$\ell_{ij} = \left\{ \begin{array}{ll} a_{ij} & i > j \\ 0 & i \leq j \end{array} \right. ,$$

and U is a strictly upper triangular matrix that is similarly defined: $u_{ij} = a_{ij}$ if $i < j$, and 0 otherwise.

The *Jacobi method* is defined by the splitting

$$A = M - N, \quad M = D, \quad N = -(L + U).$$

That is,

$$\mathbf{x}^{(k+1)} = D^{-1}[-(L + U)\mathbf{x}^{(k)} + \mathbf{b}].$$

If we write each row of this vector equation individually, we obtain

$$\mathbf{x}_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij}\mathbf{x}_j^{(k)} \right).$$

This description of the Jacobi method is helpful for its practical implementation, but it also reveals how the method can be improved. If the components of $\mathbf{x}^{(k+1)}$ are computed in order, then the computation of $\mathbf{x}_i^{(k+1)}$ uses components $1, 2, \ldots, i - 1$ of $\mathbf{x}^{(k)}$ even though these components of $x^{(k+1)}$ have already been computed.

Exercise 2.5.1 *Solve the linear system* $A\mathbf{x} = \mathbf{b}$ *by using the Jacobi method, where*

$$A = \begin{bmatrix} 2 & 7 & 1 \\ 4 & 1 & -1 \\ 1 & -3 & 12 \end{bmatrix}$$

and

$$\mathbf{b} = \begin{bmatrix} 19 \\ 3 \\ 31 \end{bmatrix}.$$

Compute the iteration matrix T *using the fact that* $M = D$ *and* $N = -(L+U)$ *for the Jacobi method. Is* $\rho(T) < 1$*?*
Hint: First rearrange the order of the equations so that the matrix is strictly diagonally dominant.

2.5.1.3 Gauss-Seidel Method

By modifying the Jacobi method to use the most up-to-date information available, we obtain the *Gauss-Seidel method*

$$\mathbf{x}_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij}\mathbf{x}_j^{(k+1)} - \sum_{j>i} a_{ij}\mathbf{x}_j^{(k)} \right).$$

This is equivalent to using the splitting $A = M - N$ where $M = D + L$ and $N = -U$; that is,

$$\mathbf{x}^{(k+1)} = (D+L)^{-1}[-U\mathbf{x}^{(k)} + \mathbf{b}].$$

Typically, this iteration converges more rapidly than the Jacobi method, but the Jacobi method retains one significant advantage: because each component of $\mathbf{x}^{(k+1)}$ is computed independently of the others, the Jacobi method can be parallelized, whereas the Gauss-Seidel method cannot, because the computation of $\mathbf{x}_i^{(k+1)}$ depends on $\mathbf{x}_j^{(k+1)}$ for $j < i$.

Exercise 2.5.2 *Solve the same system* $A\mathbf{x} = \mathbf{b}$ *from above, where*

$$A = \begin{bmatrix} 2 & 7 & 1 \\ 4 & 1 & -1 \\ 1 & -3 & 12 \end{bmatrix}$$

and

$$\mathbf{b} = \begin{bmatrix} 19 \\ 3 \\ 31 \end{bmatrix}.$$

using the Gauss-Seidel Method. What are the differences between this computation and the one from Exercise 2.5.1?

2.5.1.4 Successive Overrelaxation

Both iterations are guaranteed to converge if A is strictly diagonally dominant. Furthermore, Gauss-Seidel is guaranteed to converge if A is symmetric positive definite. However, in certain important applications, both methods can converge quite slowly. To accelerate convergence, we first rewrite the Gauss-Seidel method as follows:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + [\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}].$$

The quantity in brackets is the step taken from $\mathbf{x}^{(k)}$ to $\mathbf{x}^{(k+1)}$. However, if the direction of this step corresponds closely to the step $\mathbf{x} - \mathbf{x}^{(k)}$ to the exact solution, it may be possible to accelerate convergence by increasing the length of this step. That is, we introduce a parameter ω so that

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega[\mathbf{x}_{GS}^{(k+1)} - \mathbf{x}^{(k)}],$$

where $\mathbf{x}_{GS}^{(k+1)}$ is the iterate obtained from $\mathbf{x}^{(k)}$ by the Gauss-Seidel method. By choosing $\omega > 1$, which is called *overrelaxation*, we take a larger step in the direction of $[\mathbf{x}_{GS}^{(k+1)} - \mathbf{x}^{(k)}]$ than Gauss-Seidel would call for.

This approach leads to the method of *successive overrelaxation (SOR)*,

$$(D + \omega L)\mathbf{x}^{(k+1)} = [(1-\omega)D - \omega U]\mathbf{x}^{(k)} + \omega\mathbf{b}.$$

Note that if $\omega = 1$, then SOR reduces to the Gauss-Seidel method. If we examine the iteration matrix T_ω for SOR, we have

$$T_\omega = (D + \omega L)^{-1}[(1-\omega)D - \omega U].$$

Because the matrices $(D + \omega L)$ and $[(1-\omega)D - \omega U]$ are both triangular, it follows that

$$\det(T_\omega) = \left(\prod_{i=1}^n a_{ii}^{-1}\right)\left(\prod_{i=1}^n (1-\omega)a_{ii}\right) = (1-\omega)^n.$$

Because the determinant is the product of the eigenvalues, it follows that $\rho(T_\omega) \geq |1-\omega|$.

Exercise 2.5.3 *By the above argument, find a lower and upper bound for the parameter ω. Hint: Consider criteria for $\rho(T_\omega)$ if this method converges.*

In some cases, it is possible to analytically determine the optimal value of ω, for which convergence is most rapid. For example, if A is symmetric positive definite and tridiagonal, then the optimal value is

$$\omega = \frac{2}{1 + \sqrt{1 - [\rho(T_j)]^2}},$$

where T_j is the iteration matrix $-D^{-1}(L+U)$ for the Jacobi method.

Exercise 2.5.4 *Suppose we wish to solve*

$$A\mathbf{x} = \mathbf{b}.$$

Show that there exists a diagonal matrix D such that

$$B = DAD^{-1}$$

is symmetric and positive definite, then SOR converges for the original problem. Hint: Use the fact that SOR converges for any symmetric positive definite matrix.

A natural criterion for stopping any iterative method is to check whether $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|$ is less than some tolerance. However, if $\|T\| < 1$ in some natural matrix norm, then we have

$$\|\mathbf{x}^{(k)} - \mathbf{x}\| \leq \|T\|\|\mathbf{x}^{(k-1)} - \mathbf{x}\| \leq \|T\|\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\| + \|T\|\|\mathbf{x}^{(k)} - \mathbf{x}\|,$$

which yields the estimate

$$\|\mathbf{x}^{(k)} - \mathbf{x}\| \leq \frac{\|T\|}{1 - \|T\|}\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|.$$

Therefore, the tolerance must be chosen with $\|T\|/(1 - \|T\|)$ in mind, as this can be quite large when $\|T\| \approx 1$.

2.5.2 Krylov Subspace Methods

We have learned about *stationary* iterative methods for solving $A\mathbf{x} = \mathbf{b}$, that have the form of a fixed-point iteration. Now, we will consider an alternative approach to developing iterative methods, that leads to *non-stationary iterative methods*, in which *search directions* are used to progress from each iterate to the next. That is,

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}_k$$

where $\mathbf{p}_k$ is a search direction that is chosen so as to be approximately parallel to the error $\mathbf{e}_k = \mathbf{x} - \mathbf{x}^{(k)}$, and α_k is a constant that determines how far along that direction to proceed so that $\mathbf{x}^{(k+1)}$, in some sense, will be as close to $\mathbf{x}$ as possible.

2.5.2.1 Steepest Descent

We assume that A is symmetric positive definite, and consider the problem of minimizing the function

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}.$$

Differentiating, we obtain

$$\nabla\phi(\mathbf{x}) = A\mathbf{x} - \mathbf{b}.$$

Therefore, this function has one critical point, when $A\mathbf{x} = \mathbf{b}$. Differentiating $\nabla\phi$, we find that the Hessian matrix of ϕ is A. Because A is symmetric positive definite, it follows that the unique minimizer of ϕ is the solution to $A\mathbf{x} = \mathbf{b}$. Therefore, we can use techniques for minimizing ϕ to solve $A\mathbf{x} = \mathbf{b}$.

From any vector $\mathbf{x}_0$, the *direction of steepest descent* is given by

$$-\nabla\phi(\mathbf{x}_0) = \mathbf{b} - A\mathbf{x}_0 = \mathbf{r}_0,$$

the *residual* vector. This suggests a simple non-stationary iterative method, which is called the *method of steepest descent.* The basic idea is to choose the search direction $\mathbf{p}_k$ to be $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}^{(k)}$, and then to choose α_k so as to minimize $\phi(\mathbf{x}^{(k+1)}) = \phi(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}_k)$. This entails solving a single-variable minimization problem to obtain α_k. We have

$$\begin{aligned}
\frac{d}{d\alpha_k}[\phi(\mathbf{x}^{(k)} + \alpha_k\mathbf{r}_k)] &= \frac{d}{d\alpha_k}\left[\frac{1}{2}(\mathbf{x}^{(k)} + \alpha_k\mathbf{r}_k)^T A(\mathbf{x}^{(k)} + \alpha_k\mathbf{r}_k) - \right. \\
&\qquad \left. \mathbf{b}^T(\mathbf{x}^{(k)} + \alpha_k\mathbf{r}_k)\right] \\
&= \mathbf{r}_k^T A\mathbf{x}^{(k)} + \alpha_k\mathbf{r}_k^T A\mathbf{r}_k - \mathbf{b}^T\mathbf{r}_k] \\
&= -\mathbf{r}_k^T\mathbf{r}_k + \alpha_k\mathbf{r}_k^T A\mathbf{r}_k.
\end{aligned}$$

It follows that the optimal choice for α_k is

$$\alpha_k = \frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{r}_k^T A\mathbf{r}_k},$$

and since A is symmetric positive definite, the denominator is guaranteed to be positive.

The method of steepest descent is effective when A is well-conditioned, but when A is ill-conditioned, convergence is very slow, because the level curves of ϕ become long, thin hyperellipsoids in which the direction of steepest descent does not yield much progress toward the minimum. Another problem with this method is that while it can be shown that $\mathbf{r}_{k+1}$ is orthogonal to $\mathbf{r}_k$, so that each direction is completely independent of the previous one, $\mathbf{r}_{k+1}$ is not necessarily independent of previous search directions.

Exercise 2.5.5 *Show that each search direction* $\mathbf{r}_k$ *is orthogonal to the previous search direction* $\mathbf{r}_{k-1}$.

In fact, even in the 2 × 2 case, where only two independent search directions are available, the method of steepest descent exhibits a "zig-zag" effect because it continually alternates between two orthogonal search directions, and the more ill-conditioned A is, the smaller each step tends to be.

2.5.2.2 The Lanczos Algorithm

A more efficient iteration can be obtained if it can be guaranteed that each residual is orthogonal to *all* previous residuals. While it would be preferable to require that all *search directions* are orthogonal, this goal is unrealistic, so we settle for orthogonality of the residuals instead. For simplicity, and without loss of generality, we assume that $\mathbf{b}^T\mathbf{b} = \|\mathbf{b}\|_2^2 = 1$. Then, this orthogonality can be realized if we prescribe that each iterate $\mathbf{x}^{(k)}$ has the form

$$\mathbf{x}^{(k)} = Q_k\mathbf{y}_k$$

where Q_k is an $n \times k$ *orthogonal* matrix, meaning that $Q_k^T Q_k = I$, and $\mathbf{y}_k$ is a k-vector of coefficients.

If we also prescribe that the first column of Q_k is $\mathbf{b}$, then, to ensure that each residual is orthogonal to all previous residuals, we first note that

$$\mathbf{b} - A\mathbf{x}^{(k)} = Q_k\mathbf{e}_1 - A\mathbf{Q}_k\mathbf{y}_k.$$

That is, the residual lies in the span of the space spanned by the columns of Q_k, and the vectors obtained by multiplying A by those columns. Therefore, if we choose the columns of Q_k so that they form an orthonormal basis for the *Krylov subspace*

$$\mathcal{K}(\mathbf{b}, A, k) = \text{span}\{\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, \ldots, A^{k-1}\mathbf{b}\},$$

then we ensure that $\mathbf{r}_k$ is in the span of the columns of Q_{k+1}.

Then, we can guarantee that the residual is orthogonal to the columns of Q_k, which span the space that contains all previous residuals, by requiring

$$\mathbf{Q}_k^T(\mathbf{b} - A\mathbf{x}^{(k)}) = Q_k^T\mathbf{b} - Q_k^T A Q_k \mathbf{y}_k = \mathbf{e}_1 - T_k\mathbf{y}_k = \mathbf{0},$$

where

$$T_k = Q_k^T A Q_k.$$

It is easily seen that T_k is symmetric positive definite, since A is.

The columns of each matrix Q_k, denoted by $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k$, are defined to be

$$\mathbf{q}_k = p_{k-1}(A)\mathbf{b},$$

where $p_0, p_1, \ldots$ define a sequence of *orthogonal polynomials* with respect to the inner product

$$\langle p, q\rangle = \mathbf{b}^T p(A) q(A)\mathbf{b}.$$

That is, these polynomials satisfy

$$\langle p_i, p_j\rangle = \mathbf{b}^T p_i(A) p_j(A)\mathbf{b} = \left\{ \begin{array}{cc} 1 & i = j \\ 0 & i \neq j \end{array} \right. .$$

Like any sequence of orthogonal polynomials, they satisfy a *3-term recurrence relation*

$$\beta_j p_j(\lambda) = (\lambda - \alpha_j) p_{j-1}(\lambda) - \beta_{j-1} p_{j-2}(\lambda), \quad j \geq 1, \quad p_0(\lambda) \equiv 1, p_{-1}(\lambda) \equiv 0,$$

which is obtained by applying Gram-Schmidt orthogonalization to the monomial basis.

To obtain the *recursion coefficients* α_j and β_j, we use the requirement that the polynomials must be orthogonal. This yields

$$\alpha_j = \langle p_{j-1}(\lambda), \lambda p_{j-1}(\lambda)\rangle, \quad \beta_j^2 = \langle p_{j-1}(\lambda), \lambda^2 p_{j-1}(\lambda)\rangle - \alpha_j^2.$$

It also follows from the 3-term recurrence relation that

$$A\mathbf{q}_j = \beta_{j-1}\mathbf{q}_{j-1} + \alpha_j \mathbf{q}_j + \beta_j \mathbf{q}_{j+1},$$

and therefore T_k is tridiagonal. In fact, we have

$$AQ_k = Q_k T_k + \beta_k \mathbf{q}_{k+1}\mathbf{e}_k^T,$$

where

$$T_k = \left[\begin{array}{cccccc} \alpha_1 & \beta_1 & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \beta_{k-2} & \alpha_{k-1} & \beta_{k-1} \\ & & & \beta_{k-1} & \alpha_k \end{array} \right].$$

Furthermore, the residual is given by

$$\mathbf{b} - A\mathbf{x}^{(k)} = Q_k\mathbf{e}_1 - AQ_k\mathbf{y}_k = Q_k\mathbf{e}_1 - Q_k T_k \mathbf{y}_k - \beta_k \mathbf{q}_{k+1}\mathbf{e}_k^T\mathbf{y}_k = -\beta_k\mathbf{q}_{k+1} y_k.$$

We now have the following algorithm for generating a sequence of approximations to the solution of $A\mathbf{x} = \mathbf{b}$, for which each residual is orthogonal to all previous residuals.

```
$k = 0$, $\mathbf{r}_k = \mathbf{b}$, $\mathbf{q}_k = \mathbf{0}$, $\mathbf{x}^{(k)} = \mathbf{0}$
while $\mathbf{x}^{(k)}$ is not converged do
    $\beta_k = \|\mathbf{r}_k\|_2$
    $\mathbf{q}_{k+1} = \mathbf{r}_k/\beta_k$
    $k = k + 1$
    $\mathbf{v}_k = A\mathbf{q}_k$
    $\alpha_k = \mathbf{q}_k^T\mathbf{v}_k$
    $\mathbf{r}_k = \mathbf{v}_k - \alpha_k\mathbf{q}_k - \beta_{k-1}\mathbf{q}_{k-1}$
    $\mathbf{x}^{(k)} = \beta_0 Q_k T_k^{-1}\mathbf{e}_1$
end
```

This method is the *Lanczos iteration.* It is not only used for solving linear systems; the matrix T_k is also useful for approximating extremal eigenvalues of A, and for approximating quadratic or bilinear forms involving functions of A, such as the inverse or exponential.

Exercise 2.5.6 *Implement the above Lanczos algorithm in* MATLAB*. Your function should take a random symmetric matrix A, a random initial vector* $\mathbf{u}$*, and n where n is the number of iterations as input. Also, your function should return the symmetric matrix T, where T is the tridiagonal matrix described above that contains quantities α_j and β_j that are computed by the algorithm.*

2.5.2.3 The Conjugate Gradient Method

To improve efficiency, the tridiagonal structure of T_k can be exploited so that the vector $\mathbf{y}_k$ can easily be obtained from $\mathbf{y}_{k-1}$ by a few simple recurrence relations. However, the Lanczos iteration is not normally used directly to solve $A\mathbf{x} = \mathbf{b}$, because it does not provide a simple method of computing $\mathbf{x}^{(k+1)}$ from $\mathbf{x}^{(k)}$, as in a general non-stationary iterative method.

Instead, we note that because T_k is tridiagonal, and symmetric positive definite, it has an LDL^T factorization

$$T_k = L_k D_k L_k^T,$$

where

$$L_k = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_1 & 1 & 0 & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & l_{k-1} & 1 \end{bmatrix}, \quad D_k = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & d_k \end{bmatrix},$$

and the diagonal entries of D_k are positive. Note that because T_k is tridiagonal, L_k has lower bandwidth 1.

It follows that if we define the matrix $\tilde{P}_k$ by the matrix equation

$$\tilde{P}_k L_k^T = Q_k,$$

then

$$x^{(k)} = Q_k\mathbf{y}_k = \tilde{P}_k L_k^T T_k^{-1}\mathbf{e}_1 = \tilde{P}_k\mathbf{w}_k$$

where $\mathbf{w}_k$ satisfies

$$L_k D_k \mathbf{w}_k = \mathbf{e}_1.$$

This representation of $\mathbf{x}^{(k)}$ is more convenient than $Q_k\mathbf{y}_k$, because, as a consequence of the recursive definitions of L_k and D_k, and the fact that L_kD_k is lower triangular, we have

$$\mathbf{w}_k = \left[\begin{array}{c} \mathbf{w}_{k-1} \\ w_k \end{array}\right],$$

that is, the first $k-1$ elements of $\mathbf{w}_k$ are the elements of $\mathbf{w}_{k-1}$.

Therefore,

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + w_k\tilde{\mathbf{p}}_k, \quad \mathbf{x}^{(0)} = \mathbf{0}.$$

That is, the columns of $\tilde{P}_k$, which are the vectors $\tilde{\mathbf{p}}_1, \ldots, \tilde{\mathbf{p}}_k$, are search directions. Furthermore, in view of

$$\tilde{P}_k^T A\tilde{P}_k = L_k^{-1}Q_k^TAQ_kL_k^{-T} = L_k^{-1}T_kL_k^{-T} = L_k^{-1}L_kD_kL_k^TL_k^{-T} = D_k,$$

which implies

$$\tilde{\mathbf{p}}_i^T A\tilde{\mathbf{p}}_j = \left\{ \begin{array}{ll} d_i > 0 & i = j \\ 0 & i \neq j \end{array} \right. ,$$

we see that these search directions, while not orthogonal, are A-*orthogonal*, or A-*conjugate*. Therefore, they are linearly independent, thus guaranteeing convergence, in exact arithmetic, within n iterations. It is for this reason that the Lanczos iteration, reformulated in this manner with these search directions, is called the *conjugate gradient method*.

From the definition of P_k, we obtain the relation

$$l_{k-1}\tilde{\mathbf{p}}_{k-1} + \tilde{\mathbf{p}}_k = \tilde{\mathbf{q}}_k, \quad k > 1, \quad \tilde{\mathbf{p}}_1 = \tilde{\mathbf{q}}_1.$$

It follows that each search direction $\tilde{\mathbf{p}}_k$ is a linear combination of the residual $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}^{(k-1)}$, which is a scalar multiple of $\mathbf{q}_k$, and the previous search direction $\mathbf{p}_{k-1}$, except for the first direction, which is equal to $\mathbf{q}_1 = \mathbf{b}$. The exact linear combination can be determined by the requirement that the search directions be A-conjugate.

Specifically, if we define $\mathbf{p}_k = \|\mathbf{r}_k\|_2\tilde{\mathbf{p}}_k$, then, from the previous linear combination, we have

$$\mathbf{p}_k = \mathbf{r}_k + \mu_k\mathbf{p}_{k-1},$$

for some constant μ_k. From the requirement that $\mathbf{p}_{k-1}^TA\mathbf{p}_k = 0$, we obtain

$$\mu_k = -\frac{\mathbf{p}_{k-1}^TA\mathbf{r}_k}{\mathbf{p}_{k-1}^TA\mathbf{p}_{k-1}}.$$

We have eliminated the computation of the $\mathbf{q}_k$ from the algorithm, as we can now use the residuals $\mathbf{r}_k$ instead to obtain the search directions that we will actually use, the $\mathbf{p}_k$.

The relationship between the residuals and the search direction also provides a simple way to compute each residual from the previous one. We have

$$\mathbf{r}_{k+1} = \mathbf{b} - A\mathbf{x}^{(k)} = \mathbf{b} - A[\mathbf{x}^{(k-1)} + w_k\tilde{\mathbf{p}}_k] = \mathbf{r}_k - w_kA\tilde{\mathbf{p}}_k.$$

The orthogonality of the residuals, and the A-orthogonality of the search directions, yields the relations

$$\mathbf{r}_k^T\mathbf{r}_k = -w_{k-1}\mathbf{r}_k^TA\tilde{\mathbf{p}}_{k-1} = \frac{w_{k-1}}{\|\mathbf{r}_k\|_2}\mathbf{r}_k^TA\mathbf{p}_{k-1},$$

and

$$\begin{aligned}
\mathbf{r}_{k-1}^T\mathbf{r}_{k-1} &= w_{k-1}\mathbf{r}_{k-1}^T A\tilde{\mathbf{p}}_{k-1} \\
&= w_{k-1}(\mathbf{p}_{k-1} - \beta_k\mathbf{p}_{k-2})^T A\tilde{\mathbf{p}}_{k-1} \\
&= w_{k-1}\mathbf{p}_{k-1}^T A\tilde{\mathbf{p}}_{k-1} \\
&= \frac{w_{k-1}}{\|\mathbf{r}_{k-1}\|_2}\mathbf{p}_{k-1}^T A\mathbf{p}_{k-1}.
\end{aligned}$$

This yields the alternative formula

$$\mu_k = -\frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{r}_{k-1}^T\mathbf{r}_{k-1}},$$

which eliminates one unnecessary matrix-vector multiplication per iteration.

To complete the algorithm, we need an efficient method of computing

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + w_k\tilde{\mathbf{p}}_k = \mathbf{x}^{(k-1)} + \nu_k\mathbf{p}_k, \quad \mathbf{r}_{k+1} = \mathbf{r}_k - \nu_k A\mathbf{p}_k$$

for some constant ν_k. From the definition of $\mathbf{p}_k = \|\mathbf{r}_k\|_2\tilde{\mathbf{p}}_k$, we know that $\nu_k = w_k/\|\mathbf{r}_k\|_2$, but we wish to avoid the explicit computation of T_k and its LDL^T factorization that are needed to compute w_k. Instead, we use the relation

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \nu_k A\mathbf{p}_k$$

and the orthogonality of the residuals to obtain

$$\nu_k = \frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{r}_k^T A\mathbf{p}_k}.$$

From the relationship between the residuals and search directions, and the A-orthogonality of the search directions, we obtain

$$\nu_k = \frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{p}_k^T A\mathbf{p}_k}.$$

The following algorithm results:

```
$k = 1$, $\mathbf{r}_1 = \mathbf{b}$, $\mathbf{x}^{(0)} = \mathbf{0}$
while not converged do
    if $k > 1$ then
        $\mu_k = -\mathbf{r}_k^T\mathbf{r}_k/\mathbf{r}_{k-1}^T\mathbf{r}_{k-1}$
        $\mathbf{p}_k = \mathbf{r}_k + \mu_k\mathbf{p}_{k-1}$
    else
        $\mathbf{p}_1 = \mathbf{r}_1$
    end if
    $\mathbf{v}_k = A\mathbf{p}_k$
    $\nu_k = \mathbf{r}_k^T\mathbf{r}_k/\mathbf{p}_k^T\mathbf{v}_k$
    $\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \nu_k\mathbf{p}_k$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \nu_k\mathbf{v}_k$
    $k = k + 1$
end while
```

An appropriate stopping criterion is that the norm of the residual $\mathbf{r}_{k+1}$ is smaller than some tolerance. It is also important to impose a maximum number of iterations. Note that only one matrix-vector multiplication per iteration is required.

2.5.2.4 Preconditioning

The conjugate gradient method is far more effective than the method of steepest descent, but it can also suffer from slow convergence when A is ill-conditioned. Therefore, the conjugate gradient method is often paired with a *preconditioner* that transforms the problem $A\mathbf{x} = \mathbf{b}$ into an equivalent problem in which the matrix is close to I. The basic idea is to solve the problem

$$\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$$

where

$$\tilde{A} = C^{-1}AC^{-1}, \quad \tilde{\mathbf{x}} = C\mathbf{x}, \quad \tilde{\mathbf{b}} = C^{-1}\mathbf{b},$$

and C is symmetric positive definite. Modifying the conjugate gradient method to solve this problem, we obtain the algorithm

```
$k = 1$, $C^{-1}\mathbf{r}_1 = C^{-1}\mathbf{b}$, $C\mathbf{x}^{(0)} = \mathbf{0}$
while not converged do
    if $k > 1$ then
        $\mu_k = -\mathbf{r}_k^T C^{-2}\mathbf{r}_k / \mathbf{r}_{k-1}^T C^{-2}\mathbf{r}_{k-1}$
        $C\mathbf{p}_k = C^{-1}\mathbf{r}_k + \mu_k C\mathbf{p}_{k-1}$
    else
        $C\mathbf{p}_1 = C^{-1}\mathbf{r}_1$
    end if
    $C^{-1}\mathbf{v}_k = C^{-1}A\mathbf{p}_k$
    $\nu_k = \mathbf{r}_k^T C^{-2}\mathbf{r}_k / \mathbf{p}_k^T C\mathbf{v}_k$
    $C\mathbf{x}^{(k)} = C\mathbf{x}^{(k-1)} + \nu_k C\mathbf{p}_k$
    $C^{-1}\mathbf{r}_{k+1} = C^{-1}\mathbf{r}_k - \nu_k\mathbf{v}_k$
    $k = k + 1$
end while
```

which, upon defining $M = C^2$, simplifies to

```
$k = 1$, $\mathbf{r}_1 = \mathbf{b}$, $\mathbf{x}^{(0)} = \mathbf{0}$
while not converged do
    Solve $M\mathbf{z}_k = \mathbf{r}_k$
    if $k > 1$ then
        $\mu_k = -\mathbf{r}_k^T\mathbf{z}_k / \mathbf{r}_{k-1}^T\mathbf{z}_{k-1}$
        $\mathbf{p}_k = \mathbf{z}_k + \mu_k\mathbf{p}_{k-1}$
    else
        $\mathbf{p}_1 = \mathbf{z}_1$
    end if
    $\mathbf{v}_k = A\mathbf{p}_k$
    $\nu_k = \mathbf{r}_k^T\mathbf{z}_k / \mathbf{p}_k^T\mathbf{v}_k$
    $\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \nu_k\mathbf{p}_k$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \nu_k\mathbf{v}_k$
    $k = k + 1$
end while
```

We see that the action of the transformation is only felt through the *preconditioner* $M = C^2$. Because a system involving M is solved during each iteration, it is essential that such a system is easily solved. One example of such a preconditioner is to define $M = HH^T$, where H is an "incomplete Cholesky factor" of A, which is a sparse matrix that approximates the true Cholesky factor.

Chapter 3

Least Squares Problems

3.1 The Full Rank Least Squares Problem

Given an $m \times n$ matrix A, with $m \geq n$, and an m-vector $\mathbf{b}$, we consider the *overdetermined* system of equations $A\mathbf{x} = \mathbf{b}$, in the case where A has full column rank. If $\mathbf{b}$ is in the range of A, then there exists a unique solution $\mathbf{x}^*$. For example, there exists a unique solution in the case of

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

but not if $\mathbf{b} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T$. In such cases, when $\mathbf{b}$ is not in the range of A, then we seek to minimize $\|\mathbf{b} - A\mathbf{x}\|_p$ for some p. Recall that the vector $\mathbf{r} = \mathbf{b} - A\mathbf{x}$ is known as the *residual* vector.

Different norms give different solutions. If $p = 1$ or $p = \infty$, then the function we seek to minimize, $f(x) = \|\mathbf{b} - A\mathbf{x}\|_p$ is not differentiable, so we cannot use standard minimization techniques. However, if $p = 2$, $f(x)$ is differentiable, and thus the problem is more tractable. We now consider two methods.

3.1.1 Derivation of the Normal Equations

The second method is to define $\phi(\mathbf{x}) = \frac{1}{2}\|\mathbf{b} - A\mathbf{x}\|_2^2$, which is a differentiable function of $\mathbf{x}$. To help us to characterize the minimum of this function, we first compute the gradient of simpler functions. To that end, let

$$\psi(\mathbf{x}) = \mathbf{c}^T\mathbf{x}$$

where $\mathbf{c}$ is a constant vector. Then, from

$$\psi(\mathbf{x}) = \sum_{j=1}^{n} c_j x_j,$$

we have

$$\frac{\partial \psi}{\partial x_k} = \sum_{j=1}^{n} c_j \frac{\partial x_j}{\partial x_k} = \sum_{j=1}^{n} c_j \delta_{jk} = c_k,$$

and therefore

$$\nabla\psi(\mathbf{x}) = \mathbf{c}.$$

Now, let

$$\varphi(\mathbf{x}) = \mathbf{x}^T B\mathbf{x} = \sum_{i=1}^n \sum_{j=1}^n b_{ij} x_i x_j.$$

Then

$$\begin{aligned}
\frac{\partial\varphi}{\partial x_k} &= \sum_{i=1}^n \sum_{j=1}^n b_{ij} \frac{\partial(x_i x_j)}{\partial x_k} \\
&= \sum_{i=1}^n \sum_{j=1}^n b_{ij}(\delta_{ik} x_j + x_i \delta_{jk}) \\
&= \sum_{i=1}^n \sum_{j=1}^n b_{ij} x_j \delta_{ik} + \sum_{i=1}^n \sum_{j=1}^n b_{ij} x_i \delta_{jk} \\
&= \sum_{j=1}^n b_{kj} x_j + \sum_{i=1}^n b_{ik} x_i \\
&= (B\mathbf{x})_k + \sum_{i=1}^n (B^T)_{ki} x_i \\
&= (B\mathbf{x})_k + (B^T\mathbf{x})_k.
\end{aligned}$$

We conclude that

$$\nabla\varphi(\mathbf{x}) = (B + B^T)\mathbf{x}.$$

From

$$\|\mathbf{y}\|_2^2 = \mathbf{y}^T\mathbf{y},$$

and the properties of the transpose, we obtain

$$\begin{aligned}
\frac{1}{2}\|\mathbf{b} - A\mathbf{x}\|_2^2 &= \frac{1}{2}(\mathbf{b} - A\mathbf{x})^T(\mathbf{b} - A\mathbf{x}) \\
&= \frac{1}{2}\mathbf{b}^T\mathbf{b} - \frac{1}{2}(A\mathbf{x})^T\mathbf{b} - \frac{1}{2}\mathbf{b}^T A\mathbf{x} + \frac{1}{2}\mathbf{x}^T A^T A\mathbf{x} \\
&= \frac{1}{2}\mathbf{b}^T\mathbf{b} - \mathbf{b}^T A\mathbf{x} + \frac{1}{2}\mathbf{x}^T A^T A\mathbf{x} \\
&= \frac{1}{2}\mathbf{b}^T\mathbf{b} - (A^T\mathbf{b})^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T A^T A\mathbf{x}.
\end{aligned}$$

Using the above formulas, with $\mathbf{c} = \frac{1}{2}A^T\mathbf{b}$ and $B = \frac{1}{2}A^T A$, we have

$$\nabla\left(\frac{1}{2}\|\mathbf{b} - A\mathbf{x}\|_2^2\right) = -A^T\mathbf{b} + \frac{1}{2}(A^T A + (A^T A)^T)\mathbf{x}.$$

However, because

$$(A^T A)^T = A^T (A^T)^T = A^T A,$$

this simplifies to

$$\nabla\left(\frac{1}{2}\|\mathbf{b}-A\mathbf{x}\|_2^2\right)=-A^T\mathbf{b}+A^TA\mathbf{x}=A^TA\mathbf{x}-A^T\mathbf{b}.$$

The *Hessian* of the function $\varphi(\mathbf{x})$, denoted by $H_\varphi(\mathbf{x})$, is the matrix with entries

$$h_{ij}=\frac{\partial^2\varphi}{\partial x_i\partial x_j}.$$

Because mixed second partial derivatives satisfy

$$\frac{\partial^2\varphi}{\partial x_i\partial x_j}=\frac{\partial^2\varphi}{\partial x_j\partial x_i}$$

as long as they are continuous, the Hessian is symmetric under these assumptions.

In the case of $\varphi(\mathbf{x})=\mathbf{x}^TB\mathbf{x}$, whose gradient is $\nabla\varphi(\mathbf{x})=(B+B^T)\mathbf{x}$, the Hessian is $H_\varphi(\mathbf{x})=B+B^T$. It follows from the previously computed gradient of $\frac{1}{2}\|\mathbf{b}-A\mathbf{x}\|_2^2$ that its Hessian is A^TA.

Recall that A is $m\times n$, with $m\geq n$ and $\text{rank}(A)=n$. Then, if $\mathbf{x}\neq\mathbf{0}$, it follows from the linear independence of A's columns that $A\mathbf{x}\neq\mathbf{0}$. We then have

$$\mathbf{x}^TA^TA\mathbf{x}=(A\mathbf{x})^TA\mathbf{x}=\|A\mathbf{x}\|_2^2>0,$$

since the norm of a nonzero vector must be positive. It follows that A^TA is not only symmetric, but positive definite as well. Therefore, the Hessian of $\phi(\mathbf{x})$ is positive definite, which means that the unique critical point $\mathbf{x}$, the solution to the equations $A^TA\mathbf{x}-A^T\mathbf{b}=\mathbf{0}$, is a *minimum*.

In general, if the Hessian at a critical point is

- positive definite, meaning that its eigenvalues are all positive, then the critical point is a local minimum.

- negative definite, meaning that its eigenvalues are all negative, then the critical point is a local maximum.

- indefinite, meaning that it has both positive and negative eigenvalues, then the critical point is a sadddle point.

- singular, meaning that one of its eigenvalues is zero, then the second derivative test is inconclusive.

In summary, we can minimize $\phi(\mathbf{x})$ by noting that $\nabla\phi(\mathbf{x})=A^T(\mathbf{b}-A\mathbf{x})$, which means that $\nabla\phi(\mathbf{x})=\mathbf{0}$ if and only if $A^TA\mathbf{x}=A^T\mathbf{b}$. This system of equations is called the *normal equations*, and were used by Gauss to solve the least squares problem. If $m\gg n$ then A^TA is $n\times n$, which is a much smaller system to solve than $A\mathbf{x}=\mathbf{b}$, and if $\kappa(A^TA)$ is not too large, we can use the Cholesky factorization to solve for $\mathbf{x}$, as A^TA is symmetric positive definite.

3.1.2 Solving the Normal Equations

Instead of using the Cholesky factorization, we can solve the linear least squares problem using the normal equations

$$A^TA\mathbf{x} = A^T\mathbf{b}$$

as follows: first, we solve the above system to obtain an approximate solution $\hat{\mathbf{x}}$, and compute the residual vector $\mathbf{r} = \mathbf{b} - A\hat{\mathbf{x}}$. Now, because $A^T\mathbf{r} = A^T\mathbf{b} - A^TA\hat{\mathbf{x}} = \mathbf{0}$, we obtain the system

$$\begin{aligned} \mathbf{r} + A\hat{\mathbf{x}} &= \mathbf{b} \\ A^T\mathbf{r} &= \mathbf{0} \end{aligned}$$

or, in matrix form,

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \hat{\mathbf{x}} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix}.$$

This is a large system, but it preserves the sparsity of A. It can be used in connection with iterative refinement, but unfortunately this procedure does not work well because it is very sensitive to the residual.

3.1.3 The Condition Number of A^TA

Because the coefficient matrix of the normal equations is A^TA, it is important to understand its condition number. When A is $n \times n$ and invertible,

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \frac{\sigma_1}{\sigma_n},$$

where σ_1 and σ_n are the largest and smallest singular values, respectively, of A. If A is $m \times n$ with $m > n$ and $\text{rank}(A) = n$, A^{-1} does not exist, but the quantity σ_1/σ_n is still defined and an appropriate measure of the sensitivity of the least squares problem to perturbations in the data, so we denote this ratio by $\kappa_2(A)$ in this case as well.

From the relations

$$A\mathbf{v}_j = \sigma_j\mathbf{u}_j, \quad A^T\mathbf{u}_j = \sigma_j\mathbf{v}_j, \quad j = 1, \dots, n,$$

where $\mathbf{u}_j$ and $\mathbf{v}_j$ are the left and right singular vectors, respectively, of A, we have

$$A^TA\mathbf{v}_j = \sigma_j^2\mathbf{v}_j.$$

That is, σ_j^2 is an eigenvalue of A^TA. Furthermore, because A^TA is symmetric positive definite, the eigenvalues of A^TA are also its singular values. Specifically, if $A = U\Sigma V^T$ is the SVD of A, then $V^T(\Sigma^T\Sigma)V$ is the SVD of A^TA.

It follows that the condition number in the 2-norm of A^TA is

$$\kappa_2(A^TA) = \|A^TA\|_2\|(A^TA)^{-1}\|_2 = \frac{\sigma_1^2}{\sigma_n^2} = \left(\frac{\sigma_1}{\sigma_n}\right)^2 = \kappa_2(A)^2.$$

Note that because A has full column rank, A^TA is nonsingular, and therefore $(A^TA)^{-1}$ exists, even though A^{-1} may not.

3.2 The QR Factorization

The first approach is to take advantage of the fact that the 2-norm is invariant under orthogonal transformations, and seek an orthogonal matrix Q such that the transformed problem

$$\min \|\mathbf{b} - A\mathbf{x}\|_2 = \min \|Q^T(\mathbf{b} - A\mathbf{x})\|_2$$

is "easy" to solve. Let

$$A = QR = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1,$$

where Q_1 is $m \times n$ and R_1 is $n \times n$. Then, because Q is orthogonal, $Q^T A = R$ and

$$\begin{aligned} \min \|\mathbf{b} - A\mathbf{x}\|_2 &= \min \|Q^T(\mathbf{b} - A\mathbf{x})\|_2 \\ &= \min \|Q^T\mathbf{b} - (Q^T A)\mathbf{x}\|_2 \\ &= \min \left\| Q^T\mathbf{b} - \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \mathbf{x} \right\|_2 \end{aligned}$$

If we partition

$$Q^T\mathbf{b} = \begin{bmatrix} \mathbf{c} \\ \mathbf{d} \end{bmatrix},$$

where $\mathbf{c}$ is an n-vector, then

$$\min \|\mathbf{b} - A\mathbf{x}\|_2^2 = \min \left\| \begin{bmatrix} \mathbf{c} \\ \mathbf{d} \end{bmatrix} - \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \mathbf{x} \right\|_2^2 = \min \|\mathbf{c} - R_1\mathbf{x}\|_2^2 + \|\mathbf{d}\|_2^2.$$

Therefore, the minimum is achieved by the vector $\mathbf{x}$ such that $R_1\mathbf{x} = \mathbf{c}$ and therefore

$$\min_{\mathbf{x}} \|\mathbf{b} - A\mathbf{x}\|_2 = \|\mathbf{d}\|_2 \equiv \rho_{LS}.$$

It makes sense to seek a factorization of the form $A = QR$ where Q is orthogonal, and R is upper-triangular, so that $R_1\mathbf{x} = \mathbf{c}$ is easily solved. This is called the *QR factorization* of A.

Let A be an $m \times n$ matrix with full column rank. The *QR factorization* of A is a decomposition $A = QR$, where Q is an $m \times m$ orthogonal matrix and R is an $m \times n$ upper triangular matrix. There are three ways to compute this decomposition:

1. Using Householder matrices, developed by Alston S. Householder
2. Using Givens rotations, also known as Jacobi rotations, used by W. Givens and originally invented by Jacobi for use with in solving the symmetric eigenvalue problem in 1846.
3. A third, less frequently used approach is the *Gram-Schmidt orthogonalization.*

3.2.1 Gram-Schmidt Orthogonalization

Givens rotations or Householder reflections can be used to compute the "full" QR decomposition

$$A = QR = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

where Q is an $m \times m$ orthogonal matrix, and R_1 is an $n \times n$ upper-triangular matrix that is nonsingular if and only if A is of full column rank (that is, $\text{rank}(A) = n$).

It can be seen from the above partitions of Q and R that $A = Q_1R_1$. Furthermore, it can be shown that $\text{range}(A) = \text{range}(Q_1)$, and $(\text{range}(A))^\perp = \text{range}(Q_2)$. In fact, for $k = 1, \ldots, n$, we have

$$\text{span}\{\mathbf{a}_1, \ldots, \mathbf{a}_k\} = \text{span}\{\mathbf{q}_1, \ldots, \mathbf{q}_k\},$$

where

$$A = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{bmatrix}, \quad Q = \begin{bmatrix} \mathbf{q}_1 & \cdots & \mathbf{q}_m \end{bmatrix}$$

are column partitions of A and Q, respectively.

We now examine two methods for computing the "thin" or "economy-size" QR factorization $A = Q_1R_1$, which is sufficient for solving full-rank least-squares problems, as the least-squares solution $\mathbf{x}$ that minimizes $\|\mathbf{b} - A\mathbf{x}\|_2$ can be obtained by solving the upper-triangular system $R_1\mathbf{x} = Q_1^T\mathbf{b}$ by back substitution.

3.2.2 Classical Gram-Schmidt

Consider the "thin" QR factorization

$$A = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1 & \cdots & \mathbf{q}_n \end{bmatrix} \begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{bmatrix} = QR.$$

From the above matrix product we can see that $\mathbf{a}_1 = r_{11}\mathbf{q}_1$, from which it follows that

$$r_{11} = \pm\|\mathbf{a}_1\|_2, \quad \mathbf{q}_1 = \frac{1}{\|\mathbf{a}_1\|_2}\mathbf{a}_1.$$

For convenience, we choose the + sign for r_{11}.

Next, by taking the inner product of both sides of the equation $\mathbf{a}_2 = r_{12}\mathbf{q}_1 + r_{22}\mathbf{q}_2$ with $\mathbf{q}_1$, and imposing the requirement that the columns of Q form an orthonormal set, we obtain

$$r_{12} = \mathbf{q}_1^T\mathbf{a}_2, \quad r_{22} = \|\mathbf{a}_2 - r_{12}\mathbf{q}_1\|_2, \quad \mathbf{q}_2 = \frac{1}{r_{22}}(\mathbf{a}_2 - r_{12}\mathbf{q}_1).$$

In general, we use the relation

$$\mathbf{a}_k = \sum_{j=1}^{k} r_{jk}\mathbf{q}_j$$

to obtain

$$\mathbf{q}_k = \frac{1}{r_{kk}}\left(\mathbf{a}_k - \sum_{j=1}^{k-1} r_{jk}\mathbf{q}_j\right), \quad r_{jk} = \mathbf{q}_j^T\mathbf{a}_k.$$

Note that $\mathbf{q}_k$ can be rewritten as

$$\begin{aligned}\mathbf{q}_k &= \frac{1}{r_{kk}}\left(\mathbf{a}_k - \sum_{j=1}^{k-1}(\mathbf{q}_j^T\mathbf{a}_k)\mathbf{q}_j\right) \\ &= \frac{1}{r_{kk}}\left(\mathbf{a}_k - \sum_{j=1}^{k-1}\mathbf{q}_j\mathbf{q}_j^T\mathbf{a}_k\right) \\ &= \frac{1}{r_{kk}}\left(I - \sum_{j=1}^{k-1}\mathbf{q}_j\mathbf{q}_j^T\right)\mathbf{a}_k.\end{aligned}$$

If we define $P_i = \mathbf{q}_i\mathbf{q}_i^T$, then P_i is a *symmetric projection* that satisfies $P_i^2 = P_i$, and $P_iP_j = \delta_{ij}$. Thus we can write

$$\mathbf{q}_k = \frac{1}{r_{kk}}\left(I - \sum_{j=1}^{k-1}P_j\right)\mathbf{a}_k = \frac{1}{r_{kk}}\prod_{j=1}^{k-1}(I - P_j)\mathbf{a}_k.$$

Unfortunately, Gram-Schmidt orthogonalization, as described, is numerically unstable, because, for example, if $\mathbf{a}_1$ and $\mathbf{a}_2$ are almost parallel, then $\mathbf{a}_2 - r_{12}\mathbf{q}_1$ is almost zero, and roundoff error becomes significant due to catastrophic cancellation.

3.2.3 Modified Gram-Schmidt

The *Modified Gram-Schmidt* method alleviates the numerical instability of "Classical" Gram-Schmidt. Recall

$$A = Q_1R_1 = \left[\begin{array}{ccc} r_{11}\mathbf{q}_1 & r_{12}\mathbf{q}_1 + r_{22}\mathbf{q}_2 & \cdots \end{array}\right]$$

We define

$$C^{(k)} = \sum_{i=1}^{k-1}\mathbf{q}_i\mathbf{r}_i^T, \quad \mathbf{r}_i^T = \left[\begin{array}{ccccccc} 0 & \cdots & 0 & r_{ii} & r_{i,i+1} & \cdots & r_{in} \end{array}\right]$$

which means

$$A - C^{(k)} = \left[\begin{array}{ccccc} \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & A^{(k)} \end{array}\right],$$

because the first $k-1$ columns of A are linear combinations of the first $k-1$ columns of Q_1, and the contributions of these columns of Q_1 to *all* columns of A are removed by subtracting $C^{(k)}$.

If we write

$$A^{(k)} = \left[\begin{array}{cc} \mathbf{z}_k & B_k \end{array}\right]$$

then, because the kth column of A is a linear combination of the first k columns of Q_1, and the contributions of the first $k-1$ columns are removed in $A^{(k)}$, $\mathbf{z}_k$ must be a multiple of $\mathbf{q}_k$. Therefore,

$$r_{kk} = \|\mathbf{z}_k\|_2, \quad \mathbf{q}_k = \frac{1}{r_{kk}}\mathbf{z}_k.$$

We then compute

$$\left[\begin{array}{ccc} r_{k,k+1} & \cdots & r_{k,n} \end{array}\right] = \mathbf{q}_k^T B_k$$

which yields

$$A^{(k+1)} = B_k - \mathbf{q}_k \left[\begin{array}{ccc} r_{k,k+1} & \cdots & r_{kn} \end{array} \right].$$

This process is numerically stable.

Note that Modified Gram-Schmidt computes the entries of R_1 row-by-row, rather than column-by-column, as Classical Gram-Schmidt does. This rearrangement of the order of operations, while mathematically equivalent to Classical Gram-Schmidt, is much more stable, numerically, because each entry of R_1 is obtained by computing an inner product of a column of Q_1 with a *modified* column of A, from which the contributions of all previous columns of Q_1 have been removed.

To see why this is significant, consider the inner products

$$\mathbf{u}^T\mathbf{v}, \quad \mathbf{u}^T(\mathbf{v}+\mathbf{w}),$$

where $\mathbf{u}^T\mathbf{w} = 0$. The above inner products are equal, but suppose that $|\mathbf{u}^T\mathbf{v}| \ll \|\mathbf{w}\|$. Then $\mathbf{u}^T\mathbf{v}$ is a small number that is being computed by subtraction of potentially large numbers, which is susceptible to catastrophic cancellation.

It can be shown that Modified Gram-Schmidt produces a matrix $\hat{Q}_1$ such that

$$\hat{Q}_1^T\hat{Q}_1 = I + E_{MGS}, \quad \|E_{MGS}\| \approx \mathbf{u}\kappa_2(A),$$

and $\hat{Q}_1$ can be computed in approximately $2mn^2$ flops (floating-point operations), whereas with Householder QR,

$$\hat{Q}_1^T\hat{Q}_1 = I + E_n, \quad \|E_n\| \approx \mathbf{u},$$

with $\hat{Q}_1$ being computed in approximately $2mn^2 - 2n^2/3$ flops to factor A and an additional $2mn^2 - 2n^2/3$ flops to obtain Q_1, the first n columns of Q. That is, Householder QR is much less sensitive to roundoff error than Gram-Schmidt, even with modification, although Gram-Schmidt is more efficient if an explicit representation of Q_1 desired.

3.2.4 Householder Reflections

It is natural to ask whether we can introduce more zeros with each orthogonal rotation. To that end, we examine *Householder reflections*. Consider a matrix of the form $P = I - \tau\mathbf{u}\mathbf{u}^T$, where $\mathbf{u} \neq \mathbf{0}$ and τ is a nonzero constant. It is clear that P is a symmetric rank-1 change of I. Can we choose τ so that P is also orthogonal? From the desired relation $P^TP = I$ we obtain

$$\begin{aligned} P^TP &= (I - \tau\mathbf{u}\mathbf{u}^T)^T(I - \tau\mathbf{u}\mathbf{u}^T) \\ &= I - 2\tau\mathbf{u}\mathbf{u}^T + \tau^2\mathbf{u}\mathbf{u}^T\mathbf{u}\mathbf{u}^T \\ &= I - 2\tau\mathbf{u}\mathbf{u}^T + \tau^2(\mathbf{u}^T\mathbf{u})\mathbf{u}\mathbf{u}^T \\ &= I + (\tau^2\mathbf{u}^T\mathbf{u} - 2\tau)\mathbf{u}\mathbf{u}^T \\ &= I + \tau(\tau\mathbf{u}^T\mathbf{u} - 2)\mathbf{u}\mathbf{u}^T. \end{aligned}$$

It follows that if $\tau = 2/\mathbf{u}^T\mathbf{u}$, then $P^TP = I$ for any nonzero $\mathbf{u}$. Without loss of generality, we can stipulate that $\mathbf{u}^T\mathbf{u} = 1$, and therefore P takes the form $P = I - 2\mathbf{v}\mathbf{v}^T$, where $\mathbf{v}^T\mathbf{v} = 1$.

Why is the matrix P called a reflection? This is because for any nonzero vector $\mathbf{x}$, $P\mathbf{x}$ is the reflection of $\mathbf{x}$ across the hyperplane that is normal to $\mathbf{v}$. To see this, we consider the 2×2 case and set $\mathbf{v} = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$ and $\mathbf{x} = \begin{bmatrix} 1 & 2 \end{bmatrix}^T$. Then

$$\begin{aligned} P &= I - 2\mathbf{v}\mathbf{v}^T \\ &= I - 2\begin{bmatrix} 1 \\ 0 \end{bmatrix}\begin{bmatrix} 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - 2\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \end{aligned}$$

Therefore

$$P\mathbf{x} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}.$$

Now, let $\mathbf{x}$ be any vector. We wish to construct P so that $P\mathbf{x} = \alpha\mathbf{e}_1$ for some α, where $\mathbf{e}_1 = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}^T$. From the relations

$$\|P\mathbf{x}\|_2 = \|\mathbf{x}\|_2, \quad \|\alpha\mathbf{e}_1\|_2 = |\alpha|\|\mathbf{e}_1\|_2 = |\alpha|,$$

we obtain $\alpha = \pm\|\mathbf{x}\|_2$. To determine P, we begin with the equation

$$P\mathbf{x} = (I - 2\mathbf{v}\mathbf{v}^T)\mathbf{x} = \mathbf{x} - 2\mathbf{v}\mathbf{v}^T\mathbf{x} = \alpha\mathbf{e}_1.$$

Rearranging, we obtain

$$\frac{1}{2}(\mathbf{x} - \alpha\mathbf{e}_1) = (\mathbf{v}^T\mathbf{x})\mathbf{v}.$$

It follows that the vector $\mathbf{v}$, which is a unit vector, must be a scalar multiple of $\mathbf{x} - \alpha\mathbf{e}_1$. Therefore, $\mathbf{v}$ is defined by the equations

$$\begin{aligned} v_1 &= \frac{x_1 - \alpha}{\|\mathbf{x} - \alpha\mathbf{e}_1\|_2} \\ &= \frac{x_1 - \alpha}{\sqrt{\|\mathbf{x}\|_2^2 - 2\alpha x_1 + \alpha^2}} \\ &= \frac{x_1 - \alpha}{\sqrt{2\alpha^2 - 2\alpha x_1}} \\ &= -\frac{\alpha - x_1}{\sqrt{2\alpha(\alpha - x_1)}} \\ &= -\text{sgn}(\alpha)\sqrt{\frac{\alpha - x_1}{2\alpha}}, \\ v_2 &= \frac{x_2}{\sqrt{2\alpha(\alpha - x_1)}} \\ &= -\frac{x_2}{2\alpha v_1}, \\ &\vdots \\ v_n &= -\frac{x_n}{2\alpha v_1}. \end{aligned}$$

To avoid catastrophic cancellation, it is best to choose the sign of α so that it has the opposite sign of x_1. It can be seen that the computation of $\mathbf{v}$ requires about $3n$ operations.

Note that the matrix P is never formed explicitly. For any vector $\mathbf{b}$, the product $P\mathbf{b}$ can be computed as follows:

$$P\mathbf{b} = (I - 2\mathbf{v}\mathbf{v}^T)\mathbf{b} = \mathbf{b} - 2(\mathbf{v}^T\mathbf{b})\mathbf{v}.$$

This process requires only $4n$ operations. It is easy to see that we can represent P simply by storing only $\mathbf{v}$.

Now, suppose that that $\mathbf{x} = \mathbf{a}_1$ is the first column of a matrix A. Then we construct a Householder reflection $H_1 = I - 2\mathbf{v}_1\mathbf{v}_1^T$ such that $H\mathbf{x} = \alpha\mathbf{e}_1$, and we have

$$A^{(2)} = H_1 A = \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & & & \\ \vdots & \mathbf{a}^{(2)}_{2:m,2} & \cdots & \mathbf{a}^{(2)}_{2:m,n} \\ 0 & & & \end{bmatrix}.$$

where we denote the constant α by r_{11}, as it is the $(1,1)$ element of the updated matrix $A^{(2)}$. Now, we can construct $\tilde{H}_2$ such that

$$\tilde{H}_2\mathbf{a}^{(2)}_{2:m,2} = \begin{bmatrix} r_{22} \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

$$A^{(3)} = \begin{bmatrix} 1 & 0 \\ 0 & \tilde{H}_2 \end{bmatrix} A^{(2)} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1n} \\ 0 & r_{22} & r_{23} & \cdots & r_{2n} \\ 0 & 0 & & & \\ \vdots & \vdots & \mathbf{a}^{(3)}_{3:m,3} & \cdots & \mathbf{a}^{(3)}_{3:m,n} \\ 0 & 0 & & & \end{bmatrix}.$$

Note that the first column of $A^{(2)}$ is unchanged by $\tilde{H}_2$, because $\tilde{H}_2$ only operates on rows 2 through m, which, in the first column, have zero entries. Continuing this process, we obtain

$$H_n \cdots H_1 A = A^{(n+1)} = R,$$

where, for $j = 1, 2, \ldots, n$,

$$H_j = \begin{bmatrix} I_{j-1} & 0 \\ 0 & \tilde{H}_j \end{bmatrix}$$

and R is an upper triangular matrix. We have thus factored $A = QR$, where $Q = H_1 H_2 \cdots H_n$ is an orthogonal matrix.

Note that for each $j = 1, 2, \ldots, n$, $\tilde{H}_j$ is also a Householder reflection, based on a vector whose first $j - 1$ components are equal to zero. Therefore, application of H_j to a matrix does not affect the first j rows or columns. We also note that

$$A^T A = R^T Q^T Q R = R^T R,$$

and thus R is the Cholesky factor of $A^T A$.

Example We apply Householder reflections to compute the QR factorization of the matrix from the previous example,

$$A^{(1)} = A = \begin{bmatrix} 0.8147 & 0.0975 & 0.1576 \\ 0.9058 & 0.2785 & 0.9706 \\ 0.1270 & 0.5469 & 0.9572 \\ 0.9134 & 0.9575 & 0.4854 \\ 0.6324 & 0.9649 & 0.8003 \end{bmatrix}.$$

First, we work with the first column of A,

$$\mathbf{x}_1 = \mathbf{a}^{(1)}_{1:5,1} = \begin{bmatrix} 0.8147 \\ 0.9058 \\ 0.1270 \\ 0.9134 \\ 0.6324 \end{bmatrix}, \quad \|\mathbf{x}_1\|_2 = 1.6536.$$

The corresponding Householder vector is

$$\tilde{\mathbf{v}}_1 = \mathbf{x}_1 + \|\mathbf{x}_1\|_2\mathbf{e}_1 = \begin{bmatrix} 0.8147 \\ 0.9058 \\ 0.1270 \\ 0.9134 \\ 0.6324 \end{bmatrix} + 1.6536 \begin{bmatrix} 1.0000 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2.4684 \\ 0.9058 \\ 0.1270 \\ 0.9134 \\ 0.6324 \end{bmatrix}.$$

From this vector, we build the Householder reflection

$$c = \frac{2}{\tilde{\mathbf{v}}_1^T\tilde{\mathbf{v}}_1} = 0.2450, \quad \tilde{H}_1 = I - c\tilde{\mathbf{v}}_1\tilde{\mathbf{v}}_1^T.$$

Applying this reflection to $A^{(1)}$ yields

$$\tilde{H}_1 A^{(1)}_{1:5,1:3} = \begin{bmatrix} -1.6536 & -1.1405 & -1.2569 \\ 0 & -0.1758 & 0.4515 \\ 0 & 0.4832 & 0.8844 \\ 0 & 0.4994 & -0.0381 \\ 0 & 0.6477 & 0.4379 \end{bmatrix},$$

$$A^{(2)} = \begin{bmatrix} -1.6536 & -1.1405 & -1.2569 \\ 0 & -0.1758 & 0.4515 \\ 0 & 0.4832 & 0.8844 \\ 0 & 0.4994 & -0.0381 \\ 0 & 0.6477 & 0.4379 \end{bmatrix}.$$

Next, we take the "interesting" portion of the second column of the updated matrix $A^{(2)}$, from rows 2 to 5:

$$\mathbf{x}_2 = \mathbf{a}^{(2)}_{2:5,2} = \begin{bmatrix} -0.1758 \\ 0.4832 \\ 0.4994 \\ 0.6477 \end{bmatrix}, \quad \|\mathbf{x}_2\|_2 = 0.9661.$$

The corresponding Householder vector is

$$\tilde{\mathbf{v}}_2 = \mathbf{x}_2 - \|\mathbf{x}_2\|_2\mathbf{e}_1 = \begin{bmatrix} -0.1758 \\ 0.4832 \\ 0.4994 \\ 0.6477 \end{bmatrix} - 0.9661 \begin{bmatrix} 1.0000 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1.1419 \\ 0.4832 \\ 0.4994 \\ 0.6477 \end{bmatrix}.$$

From this vector, we build the Householder reflection

$$c = \frac{2}{\tilde{\mathbf{v}}_2^T\tilde{\mathbf{v}}_2} = 0.9065, \quad \tilde{H}_2 = I - c\tilde{\mathbf{v}}_2\tilde{\mathbf{v}}_2^T.$$

Applying this reflection to $A^{(2)}$ yields

$$\tilde{H}_2 A^{(2)}_{2:5,2:3} = \begin{bmatrix} 0.9661 & 0.6341 \\ 0 & 0.8071 \\ 0 & -0.1179 \\ 0 & 0.3343 \end{bmatrix},$$

$$A^{(3)} = \begin{bmatrix} -1.6536 & -1.1405 & -1.2569 \\ 0 & 0.9661 & 0.6341 \\ 0 & 0 & 0.8071 \\ 0 & 0 & -0.1179 \\ 0 & 0 & 0.3343 \end{bmatrix}.$$

Finally, we take the interesting portion of the third column of $A^{(3)}$, from rows 3 to 5:

$$\mathbf{x}_3 = \mathbf{a}^{(3)}_{3:5,3} = \begin{bmatrix} 0.8071 \\ -0.1179 \\ 0.3343 \end{bmatrix}, \quad \|\mathbf{x}_3\|_2 = 0.8816.$$

The corresponding Householder vector is

$$\tilde{\mathbf{v}}_3 = \mathbf{x}_3 + \|\mathbf{x}_3\|_2\mathbf{e}_1 = \begin{bmatrix} 0.8071 \\ -0.1179 \\ 0.3343 \end{bmatrix} + 0.8816 \begin{bmatrix} 1.0000 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.6887 \\ -0.1179 \\ 0.3343 \end{bmatrix}.$$

From this vector, we build the Householder reflection

$$c = \frac{2}{\tilde{\mathbf{v}}_3^T\tilde{\mathbf{v}}_3} = 0.6717, \quad \tilde{H}_3 = I - c\tilde{\mathbf{v}}_3\tilde{\mathbf{v}}_3^T.$$

Applying this reflection to $A^{(3)}$ yields

$$\tilde{H}_3 A^{(3)}_{3:5,3:3} = \begin{bmatrix} -0.8816 \\ 0 \\ 0 \end{bmatrix}, \quad A^{(4)} = \begin{bmatrix} -1.6536 & -1.1405 & -1.2569 \\ 0 & 0.9661 & 0.6341 \\ 0 & 0 & -0.8816 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Applying these same Householder reflections, in order, on the *right* of the identity matrix, yields the orthogonal matrix

$$Q = H_1H_2H_3 = \begin{bmatrix} -0.4927 & -0.4806 & 0.1780 & -0.6015 & -0.3644 \\ -0.5478 & -0.3583 & -0.5777 & 0.3760 & 0.3104 \\ -0.0768 & 0.4754 & -0.6343 & -0.1497 & -0.5859 \\ -0.5523 & 0.3391 & 0.4808 & 0.5071 & -0.3026 \\ -0.3824 & 0.5473 & 0.0311 & -0.4661 & 0.5796 \end{bmatrix}$$

such that

$$A^{(4)} = R = Q^TA = H_3H_2H_1A = \begin{bmatrix} -1.6536 & -1.1405 & -1.2569 \\ 0 & 0.9661 & 0.6341 \\ 0 & 0 & -0.8816 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

is upper triangular, where

$$H_1 = \tilde{H}_1, \quad H_2 = \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \tilde{H}_2 \end{bmatrix}, \quad H_3 = \begin{bmatrix} 1 & 0 & \mathbf{0} \\ 0 & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \tilde{H}_3 \end{bmatrix},$$

are the same Householder transformations before, defined in such a way that they can be applied to the *entire* matrix A. Note that for $j = 1, 2, 3,,$

$$H_j = I - 2\mathbf{v}_j\mathbf{v}_j^T, \quad \mathbf{v}_j = \begin{bmatrix} \mathbf{0} \\ \tilde{\mathbf{v}}_j \end{bmatrix}, \quad \|\mathbf{v}_j\|_2 = \|\tilde{\mathbf{v}}_j\|_2 = 1,$$

where the first $j-1$ components of $\mathbf{v}_j$ are equal to zero.

Also, note that the first $n = 3$ columns of Q are the same as those of the matrix Q that was computed in the previous example, except for possible negation. The fourth and fifth columns are not the same, but they do span the same subspace, the orthogonal complement of the range of A. □

3.2.5 Givens Rotations

We illustrate the process in the case where A is a 2×2 matrix. In Gaussian elimination, we compute $L^{-1}A = U$ where L^{-1} is unit lower triangular and U is upper triangular. Specifically,

$$\begin{bmatrix} 1 & 0 \\ m_{21} & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} a_{11}^{(2)} & a_{12}^{(2)} \\ 0 & a_{22}^{(2)} \end{bmatrix}, \quad m_{21} = -\frac{a_{21}}{a_{11}}.$$

By contrast, the QR decomposition computes $Q^TA = R$, or

$$\begin{bmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{bmatrix}^T \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} \\ 0 & r_{22} \end{bmatrix},$$

where $\gamma^2 + \sigma^2 = 1$.

From the relationship $-\sigma a_{11} + \gamma a_{21} = 0$ we obtain

$$\begin{aligned} \gamma a_{21} &= \sigma a_{11} \\ \gamma^2 a_{21}^2 &= \sigma^2 a_{11}^2 = (1-\gamma^2) a_{11}^2 \end{aligned}$$

which yields

$$\gamma = \pm \frac{a_{11}}{\sqrt{a_{21}^2 + a_{11}^2}}.$$

It is conventional to choose the + sign. Then, we obtain

$$\sigma^2 = 1 - \gamma^2 = 1 - \frac{a_{11}^2}{a_{21}^2 + a_{11}^2} = \frac{a_{21}^2}{a_{21}^2 + a_{11}^2},$$

or

$$\sigma = \pm \frac{a_{21}}{\sqrt{a_{21}^2 + a_{11}^2}}.$$

Again, we choose the + sign. As a result, we have

$$r_{11} = a_{11} \frac{a_{11}}{\sqrt{a_{21}^2 + a_{11}^2}} + a_{21} \frac{a_{21}}{\sqrt{a_{21}^2 + a_{11}^2}} = \sqrt{a_{21}^2 + a_{11}^2}.$$

The matrix

$$Q = \begin{bmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{bmatrix}^T$$

is called a *Givens rotation.* It is called a rotation because it is orthogonal, and therefore length-preserving, and also because there is an angle θ such that $\sin\theta = \sigma$ and $\cos\theta = \gamma$, and its effect is to rotate a vector clockwise through the angle θ. In particular,

$$\begin{bmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{bmatrix}^T \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \rho \\ 0 \end{bmatrix}$$

where $\rho = \sqrt{\alpha^2 + \beta^2}$, $\alpha = \rho\cos\theta$ and $\beta = \rho\sin\theta$. It is easy to verify that the product of two rotations is itself a rotation. Now, in the case where A is an $n \times n$ matrix, suppose that we have the vector

$$\begin{bmatrix} \times \\ \vdots \\ \times \\ \alpha \\ \times \\ \vdots \\ \times \\ \beta \\ \times \\ \vdots \\ \times \end{bmatrix}.$$

Then

$$\begin{bmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & \gamma & & & & \sigma & & & \\ & & & & 1 & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & 1 & & & & \\ & & & -\sigma & & & & \gamma & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{bmatrix} \begin{bmatrix} \times \\ \vdots \\ \times \\ \alpha \\ \times \\ \vdots \\ \times \\ \beta \\ \times \\ \vdots \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \vdots \\ \times \\ \rho \\ \times \\ \vdots \\ \times \\ 0 \\ \times \\ \vdots \\ \times \end{bmatrix}.$$

So, in order to transform A into an upper triangular matrix R, we can find a product of rotations Q such that $Q^T A = R$. It is easy to see that $O(n^2)$ rotations are required. Each rotation takes $O(n)$ operations, so the entire process of computing the QR factorization requires $O(n^3)$ operations.

It is important to note that the straightforward approach to computing the entries γ and σ of the Givens rotation,

$$\gamma = \frac{\alpha}{\sqrt{\alpha^2 + \beta^2}}, \quad \sigma = \frac{\beta}{\sqrt{\alpha^2 + \beta^2}},$$

is not always advisable, because in floating-point arithmetic, the computation of $\sqrt{\alpha^2 + \beta^2}$ could overflow. To get around this problem, suppose that $|\beta| \geq |\alpha|$. Then, we can instead compute

$$\tau = \frac{\alpha}{\beta}, \quad \sigma = \frac{1}{\sqrt{1 + \tau^2}}, \quad \gamma = \sigma\tau,$$

which is guaranteed not to overflow since the only number that is squared is less than one in magnitude. On the other hand, if $|\alpha| \geq |\beta|$, then we compute

$$\tau = \frac{\beta}{\alpha}, \quad \gamma = \frac{1}{\sqrt{1 + \tau^2}}, \quad \sigma = \gamma\tau.$$

Now, we describe the entire algorithm for computing the QR factorization using Givens rotations. Let $[c, s] =$ `givens`(a, b) be a MATLAB-style function that computes c and s such that

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \quad r = \sqrt{a^2 + b^2}.$$

Then, let $G(i, j, c, s)^T$ be the Givens rotation matrix that rotates the ith and jth elements of a vector $\mathbf{v}$ clockwise by the angle θ such that $\cos\theta = c$ and $\sin\theta = s$, so that if $v_i = a$ and $v_j = b$, and $[c, s] =$ `givens`(a, b), then in the updated vector $\mathbf{u} = G(i, j, c, s)^T \mathbf{v}$, $u_i = r = \sqrt{a^2 + b^2}$ and $u_j = 0$. The QR factorization of an $m \times n$ matrix A is then computed as follows.

```
$Q = I$
$R = A$ for $j = 1 : n$ do
    for $i = m : -1 : j + 1$ do
```

```
        $[c, s] = \texttt{givens}(r_{i-1,j}, r_{ij})$
        $R = G(i, j, c, s)^T R$
        $Q = QG(i, j, c, s)$
    end
end
```

Note that the matrix Q is accumulated by *column* rotations of the identity matrix, because the matrix by which A is multiplied to reduce A to upper-triangular form, a product of row rotations, is Q^T.

Example We use Givens rotations to compute the QR factorization of

$$A = \begin{bmatrix} 0.8147 & 0.0975 & 0.1576 \\ 0.9058 & 0.2785 & 0.9706 \\ 0.1270 & 0.5469 & 0.9572 \\ 0.9134 & 0.9575 & 0.4854 \\ 0.6324 & 0.9649 & 0.8003 \end{bmatrix}.$$

First, we compute a Givens rotation that, when applied to a_{41} and a_{51}, zeros a_{51}:

$$\begin{bmatrix} 0.8222 & -0.5692 \\ 0.5692 & 0.8222 \end{bmatrix}^T \begin{bmatrix} 0.9134 \\ 0.6324 \end{bmatrix} = \begin{bmatrix} 1.1109 \\ 0 \end{bmatrix}.$$

Applying this rotation to rows 4 and 5 yields

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.8222 & -0.5692 \\ 0 & 0 & 0 & 0.5692 & 0.8222 \end{bmatrix}^T \begin{bmatrix} 0.8147 & 0.0975 & 0.1576 \\ 0.9058 & 0.2785 & 0.9706 \\ 0.1270 & 0.5469 & 0.9572 \\ 0.9134 & 0.9575 & 0.4854 \\ 0.6324 & 0.9649 & 0.8003 \end{bmatrix} =$$

$$\begin{bmatrix} 0.8147 & 0.0975 & 0.1576 \\ 0.9058 & 0.2785 & 0.9706 \\ 0.1270 & 0.5469 & 0.9572 \\ 1.1109 & 1.3365 & 0.8546 \\ 0 & 0.2483 & 0.3817 \end{bmatrix}.$$

Next, we compute a Givens rotation that, when applied to a_{31} and a_{41}, zeros a_{41}:

$$\begin{bmatrix} 0.1136 & -0.9935 \\ 0.9935 & 0.1136 \end{bmatrix}^T \begin{bmatrix} 0.1270 \\ 1.1109 \end{bmatrix} = \begin{bmatrix} 1.1181 \\ 0 \end{bmatrix}.$$

Applying this rotation to rows 3 and 4 yields

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.1136 & -0.9935 & 0 \\ 0 & 0 & 0.9935 & 0.1136 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} 0.8147 & 0.0975 & 0.1576 \\ 0.9058 & 0.2785 & 0.9706 \\ 0.1270 & 0.5469 & 0.9572 \\ 1.1109 & 1.3365 & 0.8546 \\ 0 & 0.2483 & 0.3817 \end{bmatrix} = \begin{bmatrix} 0.8147 & 0.0975 & 0.1576 \\ 0.9058 & 0.2785 & 0.9706 \\ 1.1181 & 1.3899 & 0.9578 \\ 0 & -0.3916 & -0.8539 \\ 0 & 0.2483 & 0.3817 \end{bmatrix}.$$

Next, we compute a Givens rotation that, when applied to a_{21} and a_{31}, zeros a_{31}:

$$\begin{bmatrix} 0.6295 & -0.7770 \\ 0.7770 & 0.6295 \end{bmatrix}^T \begin{bmatrix} 0.9058 \\ 1.1181 \end{bmatrix} = \begin{bmatrix} 1.4390 \\ 0 \end{bmatrix}.$$

Applying this rotation to rows 2 and 3 yields

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0.6295 & -0.7770 & 0 & 0 \\ 0 & 0.7770 & 0.6295 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} 0.8147 & 0.0975 & 0.1576 \\ 0.9058 & 0.2785 & 0.9706 \\ 1.1181 & 1.3899 & 0.9578 \\ 0 & -0.3916 & -0.8539 \\ 0 & 0.2483 & 0.3817 \end{bmatrix} = \begin{bmatrix} 0.8147 & 0.0975 & 0.1576 \\ 1.4390 & 1.2553 & 1.3552 \\ 0 & 0.6585 & -0.1513 \\ 0 & -0.3916 & -0.8539 \\ 0 & 0.2483 & 0.3817 \end{bmatrix}.$$

To complete the first column, we compute a Givens rotation that, when applied to a_{11} and a_{21}, zeros a_{21}:

$$\begin{bmatrix} 0.4927 & -0.8702 \\ 0.8702 & 0.4927 \end{bmatrix}^T \begin{bmatrix} 0.8147 \\ 1.4390 \end{bmatrix} = \begin{bmatrix} 1.6536 \\ 0 \end{bmatrix}.$$

Applying this rotation to rows 1 and 2 yields

$$\begin{bmatrix} 0.4927 & -0.8702 & 0 & 0 & 0 \\ 0.8702 & 0.4927 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} 0.8147 & 0.0975 & 0.1576 \\ 1.4390 & 1.2553 & 1.3552 \\ 0 & 0.6585 & -0.1513 \\ 0 & -0.3916 & -0.8539 \\ 0 & 0.2483 & 0.3817 \end{bmatrix} = \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.5336 & 0.5305 \\ 0 & 0.6585 & -0.1513 \\ 0 & -0.3916 & -0.8539 \\ 0 & 0.2483 & 0.3817 \end{bmatrix}.$$

Moving to the second column, we compute a Givens rotation that, when applied to a_{42} and a_{52}, zeros a_{52}:

$$\begin{bmatrix} 0.8445 & 0.5355 \\ -0.5355 & 0.8445 \end{bmatrix}^T \begin{bmatrix} -0.3916 \\ 0.2483 \end{bmatrix} = \begin{bmatrix} 0.4636 \\ 0 \end{bmatrix}.$$

Applying this rotation to rows 4 and 5 yields

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.8445 & 0.5355 \\ 0 & 0 & 0 & -0.5355 & 0.8445 \end{bmatrix}^T \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.5336 & 0.5305 \\ 0 & 0.6585 & -0.1513 \\ 0 & -0.3916 & -0.8539 \\ 0 & 0.2483 & 0.3817 \end{bmatrix} = \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.5336 & 0.5305 \\ 0 & 0.6585 & -0.1513 \\ 0 & -0.4636 & -0.9256 \\ 0 & 0 & -0.1349 \end{bmatrix}.$$

Next, we compute a Givens rotation that, when applied to a_{32} and a_{42}, zeros a_{42}:

$$\begin{bmatrix} 0.8177 & 0.5757 \\ -0.5757 & 0.8177 \end{bmatrix}^T \begin{bmatrix} 0.6585 \\ -0.4636 \end{bmatrix} = \begin{bmatrix} 0.8054 \\ 0 \end{bmatrix}.$$

Applying this rotation to rows 3 and 4 yields

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.8177 & 0.5757 & 0 \\ 0 & 0 & -0.5757 & 0.8177 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.5336 & 0.5305 \\ 0 & 0.6585 & -0.1513 \\ 0 & -0.4636 & -0.9256 \\ 0 & 0 & -0.1349 \end{bmatrix} = \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.5336 & 0.5305 \\ 0 & 0.8054 & 0.4091 \\ 0 & 0 & -0.8439 \\ 0 & 0 & -0.1349 \end{bmatrix}.$$

Next, we compute a Givens rotation that, when applied to a_{22} and a_{32}, zeros a_{32}:

$$\begin{bmatrix} 0.5523 & -0.8336 \\ 0.8336 & 0.5523 \end{bmatrix}^T \begin{bmatrix} 0.5336 \\ 0.8054 \end{bmatrix} = \begin{bmatrix} 0.9661 \\ 0 \end{bmatrix}.$$

Applying this rotation to rows 3 and 4 yields

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0.5523 & -0.8336 & 0 & 0 \\ 0 & 0.8336 & 0.5523 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.5336 & 0.5305 \\ 0 & 0.8054 & 0.4091 \\ 0 & 0 & -0.8439 \\ 0 & 0 & -0.1349 \end{bmatrix} = \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.9661 & 0.6341 \\ 0 & 0 & -0.2163 \\ 0 & 0 & -0.8439 \\ 0 & 0 & -0.1349 \end{bmatrix}.$$

Moving to the third column, we compute a Givens rotation that, when applied to a_{43} and a_{53}, zeros a_{53}:

$$\begin{bmatrix} 0.9875 & -0.1579 \\ 0.1579 & 0.9875 \end{bmatrix}^T \begin{bmatrix} -0.8439 \\ -0.1349 \end{bmatrix} = \begin{bmatrix} 0.8546 \\ 0 \end{bmatrix}.$$

Applying this rotation to rows 4 and 5 yields

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.9875 & -0.1579 \\ 0 & 0 & 0 & 0.1579 & 0.9875 \end{bmatrix}^T \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.9661 & 0.6341 \\ 0 & 0 & -0.2163 \\ 0 & 0 & -0.8439 \\ 0 & 0 & -0.1349 \end{bmatrix} = \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.9661 & 0.6341 \\ 0 & 0 & -0.2163 \\ 0 & 0 & -0.8546 \\ 0 & 0 & 0 \end{bmatrix}.$$

Finally, we compute a Givens rotation that, when applied to a_{33} and a_{43}, zeros a_{43}:

$$\begin{bmatrix} 0.2453 & -0.9694 \\ 0.9694 & 0.2453 \end{bmatrix}^T \begin{bmatrix} -0.2163 \\ -0.8546 \end{bmatrix} = \begin{bmatrix} 0.8816 \\ 0 \end{bmatrix}.$$

Applying this rotation to rows 3 and 4 yields

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.2453 & -0.9694 & 0 \\ 0 & 0 & 0.9694 & 0.2453 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.9661 & 0.6341 \\ 0 & 0 & -0.2163 \\ 0 & 0 & -0.8546 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1.6536 & 1.1405 & 1.2569 \\ 0 & 0.9661 & 0.6341 \\ 0 & 0 & -0.8816 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = R.$$

Applying the transpose of each Givens rotation, in order, to the *columns* of the identity matrix yields the matrix

$$Q = \begin{bmatrix} 0.4927 & -0.4806 & 0.1780 & -0.7033 & 0 \\ 0.5478 & -0.3583 & -0.5777 & 0.4825 & 0.0706 \\ 0.0768 & 0.4754 & -0.6343 & -0.4317 & -0.4235 \\ 0.5523 & 0.3391 & 0.4808 & 0.2769 & -0.5216 \\ 0.3824 & 0.5473 & 0.0311 & -0.0983 & 0.7373 \end{bmatrix}$$

such that $Q^T A = R$ is upper triangular. □

We showed how to construct Givens rotations in order to rotate two elements of a column vector so that one element would be zero, and that approximately $n^2/2$ such rotations could be used to transform A into an upper triangular matrix R. Because each rotation only modifies two rows of A, it is possible to interchange the order of rotations that affect different rows, and thus apply sets of rotations in parallel. This is the main reason why Givens rotations can be preferable to Householder reflections. Other reasons are that they are easy to use when the QR factorization needs to be updated as a result of adding a row to A or deleting a column of A. They are also more efficient when A is sparse.

3.3 Rank-Deficient Least Squares

3.3.1 QR with Column Pivoting

When A does not have full column rank, the property

$$\text{span}\{\mathbf{a}_1, \ldots, \mathbf{a}_k\} = \text{span}\{\mathbf{q}_1, \ldots, \mathbf{q}_k\}$$

can not be expected to hold, because the first k columns of A could be linearly dependent, while the first k columns of Q, being orthonormal, must be linearly independent.

Example The matrix

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

has rank 2, because the first two columns are parallel, and therefore are linearly dependent, while the third column is not parallel to either of the first two. Columns 1 and 3, or columns 2 and 3, form linearly independent sets. □

Therefore, in the case where $\text{rank}(A) = r < n$, we seek a decomposition of the form $A\Pi = QR$, where Π is a *permutation matrix* chosen so that the diagonal elements of R are maximized at each stage. Specifically, suppose H_1 is a Householder reflection chosen so that

$$H_1 A = \begin{bmatrix} r_{11} & \\ 0 & \\ \vdots & * \\ 0 & \end{bmatrix}, \quad r_{11} = \|\mathbf{a}_1\|_2.$$

To maximize r_{11}, we choose Π_1 so that in the column-permuted matrix $A = A\Pi_1$, we have $\|\mathbf{a}_1\|_2 \geq \|\mathbf{a}_j\|_2$ for $j \geq 2$. For Π_2, we examine the lengths of the columns of the submatrix of A obtained by removing the first row and column. It is not necessary to recompute the lengths of the columns, because we can update them by subtracting the square of the first component from the square of the total length.

This process is called *QR with column pivoting.* It yields the decomposition

$$Q \begin{bmatrix} R & S \\ 0 & 0 \end{bmatrix} = A\Pi$$

where $Q = H_1 \cdots H_r$, $\Pi = \Pi_1 \cdots \Pi_r$, and R is an upper triangular, $r \times r$ matrix. The last $m - r$ rows are necessarily zero, because every column of A is a linear combination of the first r columns of Q.

Example We perform QR with column pivoting on the matrix

$$A = \begin{bmatrix} 1 & 3 & 5 & 1 \\ 2 & -1 & 2 & 1 \\ 1 & 4 & 6 & 1 \\ 4 & 5 & 10 & 1 \end{bmatrix}.$$

Computing the 2-norms of the columns yields

$$\|\mathbf{a}_1\|^2 = 22, \quad \|\mathbf{a}_2\|^2 = 51, \quad \|\mathbf{a}_3\|^2 = 165, \quad \|\mathbf{a}_4\|^2 = 4.$$

We see that the third column has the largest 2-norm. We therefore interchange the first and third columns to obtain

$$A^{(1)} = A\Pi_1 = A \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 3 & 1 & 1 \\ 2 & -1 & 2 & 1 \\ 6 & 4 & 1 & 1 \\ 10 & 5 & 4 & 1 \end{bmatrix}.$$

We then apply a Householder transformation H_1 to $A^{(1)}$ to make the first column a multiple of $\mathbf{e}_1$, which yields

$$H_1 A^{(1)} = \begin{bmatrix} -12.8452 & -6.7729 & -4.2817 & -1.7905 \\ 0 & -2.0953 & 1.4080 & 0.6873 \\ 0 & 0.7141 & -0.7759 & 0.0618 \\ 0 & -0.4765 & 1.0402 & -0.5637 \end{bmatrix}.$$

Next, we consider the submatrix obtained by removing the first row and column of $H_1 A^{(1)}$:

$$\tilde{A}^{(2)} = \begin{bmatrix} -2.0953 & 1.4080 & 0.6873 \\ 0.7141 & -0.7759 & 0.0618 \\ -0.4765 & 1.0402 & -0.5637 \end{bmatrix}.$$

We compute the lengths of the columns, as before, except that this time, we update the lengths of the columns of A, rather than recomputing them. This yields

$$\begin{aligned} \|\tilde{\mathbf{a}}_1^{(2)}\|_2^2 &= \|\mathbf{a}_2^{(1)}\|^2 - [a_{12}^{(1)}]^2 = 51 - (-6.7729)^2 = 5.1273, \\ \|\tilde{\mathbf{a}}_2^{(2)}\|_2^2 &= \|\mathbf{a}_3^{(1)}\|^2 - [a_{13}^{(1)}]^2 = 22 - (-4.2817)^2 = 3.6667, \\ \|\tilde{\mathbf{a}}_3^{(2)}\|_2^2 &= \|\mathbf{a}_4^{(1)}\|^2 - [a_{14}^{(1)}]^2 = 4 - (-1.7905)^2 = 0.7939. \end{aligned}$$

The second column is the largest, so there is no need for a column interchange this time. We apply a Householder transformation $\tilde{H}_2$ to the first column of $\tilde{A}^{(2)}$ so that the updated column is a multiple of $\mathbf{e}_1$, which is equivalent to applying a 4×4 Householder transformation $H_2 = I - 2\mathbf{v}_2\mathbf{v}_2^T$, where the first component of $\mathbf{v}_2$ is zero, to the *second* column of $A^{(2)}$ so that the updated column is a linear combination of $\mathbf{e}_1$ and $\mathbf{e}_2$. This yields

$$\tilde{H}_2\tilde{A}^{(2)} = \left[\begin{array}{rrr} 2.2643 & -1.7665 & -0.4978 \\ 0 & -0.2559 & 0.2559 \\ 0 & 0.6933 & -0.6933 \end{array}\right].$$

Then, we consider the submatrix obtained by removing the first row and column of $H_2\tilde{A}^{(2)}$:

$$\tilde{A}^{(3)} = \left[\begin{array}{rr} -0.2559 & 0.2559 \\ 0.6933 & -0.6933 \end{array}\right].$$

Both columns have the same lengths, so no column interchange is required. Applying a Householder reflection $\tilde{H}_3$ to the first column to make it a multiple of $\mathbf{e}_1$ will have the same effect on the second column, because they are parallel. We have

$$\tilde{H}_3\tilde{A}^{(3)} = \left[\begin{array}{rr} 0.7390 & -0.7390 \\ 0 & 0 \end{array}\right].$$

It follows that the matrix $\tilde{A}^{(4)}$ obtained by removing the first row and column of $H_3\tilde{A}^{(3)}$ will be the zero matrix. We conclude that $\mathrm{rank}(A) = 3$, and that A has the factorization

$$A\Pi = Q\left[\begin{array}{cc} R & S \\ 0 & 0 \end{array}\right],$$

where

$$\Pi = \left[\begin{array}{cccc} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right],$$

$$R = \left[\begin{array}{rrr} -12.8452 & -6.7729 & -4.2817 \\ 0 & 2.2643 & -1.7665 \\ 0 & 0 & 0.7390 \end{array}\right], \quad S = \left[\begin{array}{r} -1.7905 \\ -0.4978 \\ -0.7390 \end{array}\right],$$

and $Q = H_1H_2H_3$ is the product of the Householder reflections used to reduce $A\Pi$ to upper-triangular form. □

Using this decomposition, we can solve the linear least squares problem $A\mathbf{x} = \mathbf{b}$ by observing that

$$\begin{aligned} \|\mathbf{b} - A\mathbf{x}\|_2^2 &= \left\|\mathbf{b} - Q\left[\begin{array}{cc} R & S \\ 0 & 0 \end{array}\right]\Pi^T\mathbf{x}\right\|_2^2 \\ &= \left\|Q^T\mathbf{b} - \left[\begin{array}{cc} R & S \\ 0 & 0 \end{array}\right]\left[\begin{array}{c} \mathbf{u} \\ \mathbf{v} \end{array}\right]\right\|_2^2 \\ &= \left\|\left[\begin{array}{c} \mathbf{c} \\ \mathbf{d} \end{array}\right] - \left[\begin{array}{c} R\mathbf{u} + S\mathbf{v} \\ \mathbf{0} \end{array}\right]\right\|_2^2 \\ &= \|\mathbf{c} - R\mathbf{u} - S\mathbf{v}\|_2^2 + \|\mathbf{d}\|_2^2, \end{aligned}$$

where

$$Q^T\mathbf{b} = \begin{bmatrix} \mathbf{c} \\ \mathbf{d} \end{bmatrix}, \quad \Pi^T\mathbf{x} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix},$$

with $\mathbf{c}$ and $\mathbf{u}$ being r-vectors. Thus $\min \|\mathbf{b} - A\mathbf{x}\|_2^2 = \|\mathbf{d}\|_2^2$, provided that $R\mathbf{u} + S\mathbf{v} = \mathbf{c}$. A *basic solution* is obtained by choosing $\mathbf{v} = \mathbf{0}$. A second solution is to choose $\mathbf{u}$ and $\mathbf{v}$ so that $\|\mathbf{u}\|_2^2 + \|\mathbf{v}\|_2^2$ is minimized. This criterion is related to the pseudo-inverse of A.

3.3.2 Complete Orthogonal Decomposition

After performing the QR factorization with column pivoting, we have

$$A = Q \begin{bmatrix} R & S \\ 0 & 0 \end{bmatrix} \Pi^T$$

where R is upper triangular. Then

$$A^T = \Pi \begin{bmatrix} R^T & 0 \\ S^T & 0 \end{bmatrix} Q^T$$

where R^T is lower triangular. We apply Householder reflections so that

$$Z_r \cdots Z_2 Z_1 \begin{bmatrix} R^T & 0 \\ S^T & 0 \end{bmatrix} = \begin{bmatrix} U & 0 \\ 0 & 0 \end{bmatrix},$$

where U is upper-triangular. Then

$$A^T = Z \begin{bmatrix} U & 0 \\ 0 & 0 \end{bmatrix} Q^T$$

where $Z = \Pi Z_1 \cdots Z_r$. In other words,

$$A = Q \begin{bmatrix} L & 0 \\ 0 & 0 \end{bmatrix} Z^T$$

where L is a lower-triangular matrix of size $r \times r$, where r is the rank of A. This is the *complete orthogonal decomposition* of A.

Recall that X is the *pseudo-inverse* of A if

1. $AXA = A$
2. $XAX = X$
3. $(XA)^T = XA$
4. $(AX)^T = AX$

Given the above complete orthogonal decomposition of A, the pseudo-inverse of A, denoted A^+, is given by

$$A^+ = Z \begin{bmatrix} L^{-1} & 0 \\ 0 & 0 \end{bmatrix} Q^T.$$

Let $\mathcal{X} = \{\mathbf{x} | \|\mathbf{b} - A\mathbf{x}\|_2 = \min\}$. If $\mathbf{x} \in \mathcal{X}$ and we desire $\|\mathbf{x}\|_2 = \min$, then $\mathbf{x} = A^+\mathbf{b}$. Note that in this case,

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} = \mathbf{b} - AA^+\mathbf{b} = (I - AA^+)\mathbf{b}$$

where the matrix $(I - AA^+)$ is a projection matrix $P^\perp$. To see that $P^\perp$ is a projection, note that

$$\begin{aligned} P &= AA^+ \\ &= Q \begin{bmatrix} L & 0 \\ 0 & 0 \end{bmatrix} Z^T Z \begin{bmatrix} L^{-1} & 0 \\ 0 & 0 \end{bmatrix} Q^T \\ &= Q \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix} Q^T. \end{aligned}$$

It can then be verified directly that $P = P^T$ and $P^2 = P$.

3.3.3 The Pseudo-Inverse

The singular value decomposition is very useful in studying the linear least squares problem. Suppose that we are given an m-vector $\mathbf{b}$ and an $m \times n$ matrix A, and we wish to find $\mathbf{x}$ such that

$$\|\mathbf{b} - A\mathbf{x}\|_2 = \text{minimum}.$$

From the SVD of A, we can simplify this minimization problem as follows:

$$\begin{aligned} \|\mathbf{b} - A\mathbf{x}\|_2^2 &= \|\mathbf{b} - U\Sigma V^T\mathbf{x}\|_2^2 \\ &= \|U^T\mathbf{b} - \Sigma V^T\mathbf{x}\|_2^2 \\ &= \|\mathbf{c} - \Sigma\mathbf{y}\|_2^2 \\ &= (c_1 - \sigma_1 y_1)^2 + \cdots + (c_r - \sigma_r y_r)^2 + \\ &\quad\; c_{r+1}^2 + \cdots + c_m^2 \end{aligned}$$

where $\mathbf{c} = U^T\mathbf{b}$ and $\mathbf{y} = V^T\mathbf{x}$. We see that in order to minimize $\|\mathbf{b} - A\mathbf{x}\|_2$, we must set $y_i = c_i/\sigma_i$ for $i = 1, \ldots, r$, but the unknowns y_i, for $i = r + 1, \ldots, n$, can have any value, since they do not influence $\|\mathbf{c} - \Sigma\mathbf{y}\|_2$. Therefore, if A does not have full rank, there are infinitely many solutions to the least squares problem. However, we can easily obtain the unique solution of minimum 2-norm by setting $y_{r+1} = \cdots = y_n = 0$.

In summary, the solution of minimum length to the linear least squares problem is

$$\begin{aligned} \mathbf{x} &= V\mathbf{y} \\ &= V\Sigma^+\mathbf{c} \\ &= V\Sigma^+U^T\mathbf{b} \\ &= A^+\mathbf{b} \end{aligned}$$

where Σ^+ is a diagonal matrix with entries

$$\Sigma^+ = \begin{bmatrix} \sigma_1^{-1} & & & & & \\ & \ddots & & & & \\ & & \sigma_r^{-1} & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{bmatrix}$$

and $A^+ = V\Sigma^+U^T$. The matrix A^+ is called the *pseudo-inverse* of A. In the case where A is square and has full rank, the pseudo-inverse is equal to A^{-1}. Note that A^+ is independent of $\mathbf{b}$. It also has the properties

1. $AA^+A = A$
2. $A^+AA^+ = A^+$
3. $(A^+A)^T = A^+A$
4. $(AA^+)^T = AA^+$

The solution $\mathbf{x}$ of the least-squares problem minimizes $\|\mathbf{b} - A\mathbf{x}\|_2$, and therefore is the vector that solves the system $A\mathbf{x} = \mathbf{b}$ as closely as possible. However, we can use the SVD to show that $\mathbf{x}$ is the exact solution to a related system of equations. We write $\mathbf{b} = \mathbf{b}_1 + \mathbf{b}_2$, where

$$\mathbf{b}_1 = AA^+\mathbf{b}, \quad \mathbf{b}_2 = (I - AA^+)\mathbf{b}.$$

The matrix AA^+ has the form

$$AA^+ = U\Sigma V^T V\Sigma^+U^T = U\Sigma\Sigma^+U^T = U \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix} U^T,$$

where I_r is the $r \times r$ identity matrix. It follows that $\mathbf{b}_1$ is a linear combination of $\mathbf{u}_1, \ldots, \mathbf{u}_r$, the columns of U that form an orthogonal basis for the range of A.

From $\mathbf{x} = A^+\mathbf{b}$ we obtain

$$A\mathbf{x} = AA^+\mathbf{b} = P\mathbf{b} = \mathbf{b}_1,$$

where $P = AA^+$. Therefore, the solution to the least squares problem, is also the exact solution to the system $A\mathbf{x} = P\mathbf{b}$. It can be shown that the matrix P has the properties

1. $P = P^T$
2. $P^2 = P$

In other words, the matrix P is a *projection.* In particular, it is a projection onto the space spanned by the columns of A, i.e. the range of A. That is, $P = U_rU_r^T$, where U_r is the matrix consisting of the first r columns of U.

The *residual vector* $\mathbf{r} = \mathbf{b} - A\mathbf{x}$ can be expressed conveniently using this projection. We have

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} = \mathbf{b} - AA^+\mathbf{b} = \mathbf{b} - P\mathbf{b} = (I - P)\mathbf{b} = P^\perp\mathbf{b}.$$

That is, the residual is the projection of $\mathbf{b}$ onto the *orthogonal complement* of the range of A, which is the null space of A^T. Furthermore, from the SVD, the 2-norm of the residual satisfies

$$\rho_{LS}^2 \equiv \|\mathbf{r}\|_2^2 = c_{r+1}^2 + \cdots + c_m^2,$$

where, as before, $\mathbf{c} = U^T\mathbf{b}$.

3.3.4 Perturbation Theory

Suppose that we perturb the data, so that we are solving $(A+\epsilon E)\mathbf{x}(\epsilon) = \mathbf{b}$. Then what is $\|\mathbf{x}-\mathbf{x}(\epsilon)\|_2$ or $\|\mathbf{r}-\mathbf{r}(\epsilon)\|_2$? Using the fact that $PA = AA^+A = A$, we differentiate with respect to ϵ and obtain

$$P\frac{dA}{d\epsilon} + \frac{dP}{d\epsilon}A = \frac{dA}{d\epsilon}.$$

It follows that

$$\frac{dP}{d\epsilon}A = (I-P)\frac{dA}{d\epsilon} = P^\perp \frac{dA}{d\epsilon}.$$

Multiplying through by A^+, we obtain

$$\frac{dP}{d\epsilon}P = P^\perp \frac{dA}{d\epsilon}A^+.$$

Because P is a projection,

$$\frac{d(P^2)}{d\epsilon} = P\frac{dP}{d\epsilon} + \frac{dP}{d\epsilon}P = \frac{dP}{d\epsilon},$$

so, using the symmetry of P,

$$\frac{dP}{d\epsilon} = P^\perp \frac{dA}{d\epsilon}A^+ + (A^+)^T\frac{dA^T}{d\epsilon}P^\perp.$$

Now, using a Taylor expansion around $\epsilon = 0$, as well as the relations $\hat{\mathbf{x}} = A^+\mathbf{b}$ and $\mathbf{r} = P^\perp\mathbf{b}$, we obtain

$$\begin{aligned}
\mathbf{r}(\epsilon) &= \mathbf{r}(0) + \epsilon\frac{dP^\perp}{d\epsilon}\mathbf{b} + O(\epsilon^2) \\
&= \mathbf{r}(0) + \epsilon\frac{d(I-P)}{d\epsilon}\mathbf{b} + O(\epsilon^2) \\
&= \mathbf{r}(0) - \epsilon\frac{dP}{d\epsilon}\mathbf{b} + O(\epsilon^2) \\
&= \mathbf{r}(0) - \epsilon[P^\perp E\hat{\mathbf{x}}(0) + (A^+)^T E^T\mathbf{r}(0)] + O(\epsilon^2).
\end{aligned}$$

Taking norms, we obtain

$$\frac{\|\mathbf{r}(\epsilon)-\mathbf{r}(0)\|_2}{\|\hat{\mathbf{x}}\|_2} = |\epsilon|\|E\|_2\left(1 + \|A^+\|_2\frac{\|\mathbf{r}(0)\|_2}{\|\hat{\mathbf{x}}(0)\|_2}\right) + O(\epsilon^2).$$

Note that if A is scaled so that $\|A\|_2 = 1$, then the second term above involves the condition number $\kappa_2(A)$. We also have

$$\frac{\|\mathbf{x}(\epsilon)-\mathbf{x}(0)\|_2}{\|\hat{\mathbf{x}}\|_2} = |\epsilon|\|E\|_2\left(2\kappa_2(A) + \kappa_2(A)^2\frac{\|\mathbf{r}(0)\|_2}{\|\hat{\mathbf{x}}(0)\|_2}\right) + O(\epsilon^2).$$

Note that a small perturbation in the residual does not imply a small perturbation in the solution.

3.4 The Singular Value Decomposition

3.4.1 Existence

The matrix ℓ_2-norm can be used to obtain a highly useful decomposition of any $m \times n$ matrix A. Let $\mathbf{x}$ be a unit ℓ_2-norm vector (that is, $\|\mathbf{x}\|_2 = 1$ such that $\|A\mathbf{x}\|_2 = \|A\|_2$. If $\mathbf{z} = A\mathbf{x}$, and we define $\mathbf{y} = \mathbf{z}/\|\mathbf{z}\|_2$, then we have $A\mathbf{x} = \sigma\mathbf{y}$, with $\sigma = \|A\|_2$, and both $\mathbf{x}$ and $\mathbf{y}$ are unit vectors.

We can extend $\mathbf{x}$ and $\mathbf{y}$, separately, to orthonormal bases of $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively, to obtain orthogonal matrices $V_1 = \begin{bmatrix} \mathbf{x} & V_2 \end{bmatrix} \in \mathbb{R}^{n\times n}$ and $U_1 = \begin{bmatrix} \mathbf{y} & U_2 \end{bmatrix} \in \mathbb{R}^{m\times m}$. We then have

$$\begin{aligned} U_1^T A V_1 &= \begin{bmatrix} \mathbf{y}^T \\ U_2^T \end{bmatrix} A \begin{bmatrix} \mathbf{x} & V_2 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{y}^T A\mathbf{x} & \mathbf{y}^T A V_2 \\ U_2^T A\mathbf{x} & U_2^T A V_2 \end{bmatrix} \\ &= \begin{bmatrix} \sigma\mathbf{y}^T\mathbf{y} & \mathbf{y}^T A V_2 \\ \sigma U_2^T\mathbf{y} & U_2^T A V_2 \end{bmatrix} \\ &= \begin{bmatrix} \sigma & \mathbf{w}^T \\ \mathbf{0} & B \end{bmatrix} \\ &\equiv A_1, \end{aligned}$$

where $B = U_2^T A V_2$ and $\mathbf{w} = V_2^T A^T \mathbf{y}$. Now, let

$$\mathbf{s} = \begin{bmatrix} \sigma \\ \mathbf{w} \end{bmatrix}.$$

Then $\|\mathbf{s}\|_2^2 = \sigma^2 + \|\mathbf{w}\|_2^2$. Furthermore, the first component of $A_1\mathbf{s}$ is $\sigma^2 + \|\mathbf{w}\|_2^2$.

It follows from the invariance of the matrix ℓ_2-norm under orthogonal transformations that

$$\sigma^2 = \|A\|_2^2 = \|A_1\|_2^2 \geq \frac{\|A_1\mathbf{s}\|_2^2}{\|\mathbf{s}\|_2^2} \geq \frac{(\sigma^2 + \|\mathbf{w}\|_2^2)^2}{\sigma^2 + \|\mathbf{w}\|_2^2} = \sigma^2 + \|\mathbf{w}\|_2^2,$$

and therefore we must have $\mathbf{w} = \mathbf{0}$. We then have

$$U_1^T A V_1 = A_1 = \begin{bmatrix} \sigma & 0 \\ 0 & B \end{bmatrix}.$$

Continuing this process on B, and keeping in mind that $\|B\|_2 \leq \|A\|_2$, we obtain the decomposition

$$U^T A V = \Sigma$$

where

$$U = \begin{bmatrix} \mathbf{u}_1 & \cdots & \mathbf{u}_m \end{bmatrix} \in \mathbb{R}^{m\times m}, \quad V = \begin{bmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_n \end{bmatrix} \in \mathbb{R}^{n\times n}$$

are both orthogonal matrices, and

$$\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m\times n}, \quad p = \min\{m, n\}$$

is a *diagonal* matrix, in which $\Sigma_{ii} = \sigma_i$ for $i = 1, 2, \ldots, p$, and $\Sigma_{ij} = 0$ for $i \neq j$. Furthermore, we have

$$\|A\|_2 = \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0.$$

This decomposition of A is called the *singular value decomposition*, or SVD. It is more commonly written as a factorization of A,

$$A = U\Sigma V^T.$$

3.4.2 Properties

The diagonal entries of Σ are the *singular values* of A. The columns of U are the *left singular vectors*, and the columns of V are the *right singluar vectors.* It follows from the SVD itself that the singular values and vectors satisfy the relations

$$A\mathbf{v}_i = \sigma_i \mathbf{u}_i, \quad A^T\mathbf{u}_i = \sigma_i \mathbf{v}_i, \quad i = 1, 2, \ldots, \min\{m, n\}.$$

For convenience, we denote the ith largest singular value of A by $\sigma_i(A)$, and the largest and smallest singular values are commonly denoted by $\sigma_{max}(A)$ and $\sigma_{min}(A)$, respectively.

The SVD conveys much useful information about the structure of a matrix, particularly with regard to systems of linear equations involving the matrix. Let r be the number of nonzero singular values. Then r is the rank of A, and

$$\text{range}(A) = \text{span}\{\mathbf{u}_1, \ldots, \mathbf{u}_r\}, \quad \text{null}(A) = \text{span}\{\mathbf{v}_{r+1}, \ldots, \mathbf{v}_n\}.$$

That is, the SVD yields orthonormal bases of the range and null space of A.

It follows that we can write

$$A = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^T.$$

This is called the *SVD expansion* of A. If $m \geq n$, then this expansion yields the "economy-size" SVD

$$A = U_1 \Sigma_1 V^T,$$

where

$$U_1 = \begin{bmatrix} \mathbf{u}_1 & \cdots & \mathbf{u}_n \end{bmatrix} \in \mathbb{R}^{m\times n}, \quad \Sigma_1 = \text{diag}(\sigma_1, \ldots, \sigma_n) \in \mathbb{R}^{n\times n}.$$

Example The matrix

$$A = \begin{bmatrix} 11 & 19 & 11 \\ 9 & 21 & 9 \\ 10 & 20 & 10 \end{bmatrix}$$

has the SVD $A = U\Sigma V^T$ where

$$U = \begin{bmatrix} 1/\sqrt{3} & -1/\sqrt{2} & 1/\sqrt{6} \\ 1/\sqrt{3} & 1/\sqrt{2} & 1/\sqrt{6} \\ 1/\sqrt{3} & 0 & -\sqrt{2/3} \end{bmatrix}, \quad V = \begin{bmatrix} 1/\sqrt{6} & -1/\sqrt{3} & 1/\sqrt{2} \\ \sqrt{2/3} & 1/\sqrt{3} & 0 \\ 1/\sqrt{6} & -1/\sqrt{3} & -1/\sqrt{2} \end{bmatrix},$$

and

$$S = \begin{bmatrix} 42.4264 & 0 & 0 \\ 0 & 2.4495 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Let $U = \left[\begin{array}{ccc} \mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_3 \end{array} \right]$ and $V = \left[\begin{array}{ccc} \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 \end{array} \right]$ be column partitions of U and V, respectively. Because there are only two nonzero singular values, we have $\text{rank}(A) = 2$, Furthermore, $\text{range}(A) = \text{span}\{\mathbf{u}_1, \mathbf{u}_2\}$, and $\text{null}(A) = \text{span}\{\mathbf{v}_3\}$. We also have

$$A = 42.4264\mathbf{u}_1\mathbf{v}_1^T + 2.4495\mathbf{u}_2\mathbf{v}_2^T.$$

□

The SVD is also closely related to the ℓ_2-norm and Frobenius norm. We have

$$\|A\|_2 = \sigma_1, \quad \|A\|_F^2 = \sigma_1^2 + \sigma_2^2 + \cdots + \sigma_r^2,$$

and

$$\min_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \sigma_p, \quad p = \min\{m, n\}.$$

These relationships follow directly from the invariance of these norms under orthogonal transformations.

3.4.3 Applications

The SVD has many useful applications, but one of particular interest is that the truncated SVD expansion

$$A_k = \sum_{i=1}^k \sigma_i \mathbf{u}_i \mathbf{v}_i^T,$$

where $k < r = \text{rank}(A)$, is the best approximation of A by a rank-k matrix. It follows that the distance between A and the set of matrices of rank k is

$$\min_{\text{rank}(B)=k} \|A - B\|_2 = \|A - A_k\|_2 = \left\| \sum_{i=k+1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^T \right\| = \sigma_{k+1},$$

because the ℓ_2-norm of a matrix is its largest singular value, and σ_{k+1} is the largest singular value of the matrix obtained by removing the first k terms of the SVD expansion. Therefore, σ_p, where $p = \min\{m, n\}$, is the distance between A and the set of all rank-deficient matrices (which is zero when A is already rank-deficient). Because a matrix of full rank can have arbitarily small, but still positive, singular values, it follows that the set of all full-rank matrices in $\mathbb{R}^{m\times n}$ is both open and dense.

Example The best approximation of the matrix A from the previous example, which has rank two, by a matrix of rank one is obtained by taking only the first term of its SVD expansion,

$$A_1 = 42.4264\mathbf{u}_1\mathbf{v}_1^T = \begin{bmatrix} 10 & 20 & 10 \\ 10 & 20 & 10 \\ 10 & 20 & 10 \end{bmatrix}.$$

The absolute error in this approximation is

$$\|A - A_1\|_2 = \sigma_2 \|\mathbf{u}_2\mathbf{v}_2^T\|_2 = \sigma_2 = 2.4495,$$

while the relative error is

$$\frac{\|A - A_1\|_2}{\|A\|_2} = \frac{2.4495}{42.4264} = 0.0577.$$

That is, the rank-one approximation of A is off by a little less than six percent. □

Suppose that the entries of a matrix A have been determined experimentally, for example by measurements, and the error in the entries is determined to be bounded by some value ϵ. Then, if $\sigma_k > \epsilon \geq \sigma_{k+1}$ for some $k < \min\{m, n\}$, then ϵ is at least as large as the distance between A and the set of all matrices of rank k. Therefore, due to the uncertainty of the measurement, A could very well be a matrix of rank k, but it cannot be of lower rank, because $\sigma_k > \epsilon$. We say that the ϵ-rank of A, defined by

$$\operatorname{rank}(A, \epsilon) = \min_{\|A-B\|_2 \leq \epsilon} \operatorname{rank}(B),$$

is equal to k. If $k < \min\{m, n\}$, then we say that A is *numerically rank-deficient.*

3.4.4 Minimum-norm least squares solution

One of the most well-known applications of the SVD is that it can be used to obtain the solution to the problem

$$\|\mathbf{b} - A\mathbf{x}\|_2 = \min\,, \quad \|\mathbf{x}\|_2 = \min\,.$$

The solution is

$$\hat{\mathbf{x}} = A^+\mathbf{b} = V\Sigma^+U^T\mathbf{b}$$

where A^+ is the *pseudo-inverse* of A.

3.4.5 Closest Orthogonal Matrix

Let $\mathcal{Q}_n$ be the set of all $n \times n$ orthogonal matrices. Given an $n \times n$ matrix A, we wish to find the matrix Q that satisfies

$$\|A - Q\|_F = \min\,, \quad Q \in \mathcal{Q}_n, \quad \sigma_i(Q) = 1.$$

Given $A = U\Sigma V^T$, if we compute $\hat{Q} = UIV^T$, then

$$\begin{aligned} \|A - \hat{Q}\|_F^2 &= \|U(\Sigma - I)V^T\|_F^2 \\ &= \|\Sigma - I\|_F^2 \\ &= (\sigma_1 - 1)^2 + \cdots + (\sigma_n - 1)^2 \end{aligned}$$

It can be shown that this is in fact the minimum.

A more general problem is to find $Q \in \mathcal{Q}_n$ such that

$$\|A - BQ\|_F = \min$$

for given matrices A and B. The solution is

$$\hat{Q} = UV^T, \quad B^TA = U\Sigma V^T.$$

3.4.6 Other Low-Rank Approximations

Let $\mathcal{M}_{m,n}^{(r)}$ be the set of all $m \times n$ matrices of rank r, and let $A \in \mathcal{M}_{m,n}^{(r)}$. We wish to find $B \in \mathcal{M}_{m,n}^{(k)}$, where $k < r$, such that $\|A - B\|_F = \min$.

To solve this problem, let $A = U\Sigma V^T$ be the SVD of A, and let $\hat{B} = U\Omega_k V^T$ where

$$\Omega_k = \begin{bmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_k & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{bmatrix}.$$

Then

$$\begin{aligned} \|A - \hat{B}\|_F^2 &= \|U(\Sigma - \Omega_k)V^T\|_F^2 \\ &= \|\Sigma - \Omega_k\|_F^2 \\ &= \sigma_{k+1}^2 + \cdots + \sigma_r^2. \end{aligned}$$

We now consider a variation of this problem. Suppose that B is a perturbation of A such that $A = B + E$, where $\|E\|_F^2 \leq \epsilon^2$. We wish to find $\hat{B}$ such that $\|A - \hat{B}\|_F^2 \leq \epsilon^2$, where the rank of $\hat{B}$ is minimized. We know that if $B_k = U\Omega_k V^T$ then

$$\|A - B_K\|_F^2 = \sigma_{k+1}^2 + \cdots + \sigma_r^2.$$

It follows that $\hat{B} = B_k$ is the solution if

$$\sigma_{k+1} + \cdots + \sigma_r^2 \leq \epsilon^2, \quad \sigma_k^2 + \cdots + \sigma_r^2 > \epsilon^2.$$

Note that

$$\|A^+ - \hat{B}^+\|_F^2 = \left(\frac{1}{\sigma_{k+1}^2} + \cdots + \frac{1}{\sigma_r^2} \right).$$

3.5 Least Squares with Constraints

3.5.1 Linear Constraints

Suppose that we wish to fit data as in the least squares problem, except that we are using different functions to fit the data on different subintervals. A common example is the process of fitting data using cubic splines, with a different cubic polynomial approximating data on each subinterval.

Typically, it is desired that the functions assigned to each piece form a function that is continuous on the entire interval within which the data lies. This requires that *constraints* be imposed on the functions themselves. It is also not uncommon to require that the function assembled from these pieces also has a continuous first or even second derivative, resulting in additional constraints. The result is a *least squares problem with linear constraints*, as the constraints are applied to coefficients of predetermined functions chosen as a basis for some function space, such as the space of polynomials of a given degree.

The general form of a least squares problem with linear constraints is as follows: we wish to find an n-vector $\mathbf{x}$ that minimizes $\|A\mathbf{x} - \mathbf{b}\|_2$, subject to the constraint $C^T\mathbf{x} = \mathbf{d}$, where C is a known $n \times p$ matrix and $\mathbf{d}$ is a known p-vector.

This problem is usually solved using *Lagrange multipliers.* We define

$$f(x; \lambda) = \|\mathbf{b} - A\mathbf{x}\|_2^2 + 2\lambda^T(C^T\mathbf{x} - \mathbf{d}).$$

Then

$$\nabla f = 2(A^TA\mathbf{x} - A^T\mathbf{b} + C\lambda).$$

To minimize f, we can solve the system

$$\begin{bmatrix} A^TA & C \\ C^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} = \begin{bmatrix} A^T\mathbf{b} \\ \mathbf{d} \end{bmatrix}.$$

From $A^TA\mathbf{x} = A^T\mathbf{b} - C\lambda$, we see that we can first compute $\mathbf{x} = \hat{\mathbf{x}} - (A^TA)^{-1}C\lambda$ where $\hat{\mathbf{x}}$ is the solution to the unconstrained least squares problem. Then, from the equation $C^T\mathbf{x} = \mathbf{d}$ we obtain the equation

$$C^T(A^TA)^{-1}C\lambda = C^T\hat{\mathbf{x}} - \mathbf{d},$$

which we can now solve for λ. The algorithm proceeds as follows:

1. Solve the unconstrained least squares problem $A\mathbf{x} = \mathbf{b}$ for $\hat{\mathbf{x}}$.
2. Compute $A = QR$.
3. Form $W = (R^T)^{-1}C$.
4. Compute $W = PU$, the QR factorization of W.
5. Solve $U^TU\lambda = \eta = C^T\hat{\mathbf{x}} - \mathbf{d}$ for λ. Note that

$$\begin{aligned} U^TU &= (P^TW)^T(P^TW) \\ &= W^TPP^TW \\ &= C^TR^{-1}(R^T)^{-1}C \\ &= C^T(R^TR)^{-1}C \\ &= C^T(R^TQ^TQR)^{-1}C \\ &= C^T(A^TA)^{-1}C \end{aligned}$$

6. Set $\mathbf{x} = \hat{\mathbf{x}} - (A^TA)^{-1}C\lambda$.

This method is not the most practical since it has more unknowns than the unconstrained least squares problem, which is odd because the constraints should have the effect of eliminating unknowns, not adding them. We now describe an alternate approach.

Suppose that we compute the QR factorization of C to obtain

$$Q^TC = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is a $p \times p$ upper triangular matrix. Then the constraint $C^T\mathbf{x} = \mathbf{d}$ takes the form

$$R^T\mathbf{u} = \mathbf{d}, \quad Q^T\mathbf{x} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix}.$$

Then

$$\begin{aligned}
\|\mathbf{b} - A\mathbf{x}\|_2 &= \|\mathbf{b} - AQQ^T\mathbf{x}\| \\
&= \left\| \mathbf{b} - \tilde{A} \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \right\|_2, \quad \tilde{A} = AQ \\
&= \left\| \mathbf{b} - \begin{bmatrix} \tilde{A}_1 & \tilde{A}_2 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \right\|_2 \\
&= \|\mathbf{b} - \tilde{A}_1\mathbf{u} - \tilde{A}_2\mathbf{v}\|_2
\end{aligned}$$

Thus we can obtain $\mathbf{x}$ by the following procedure:

1. Compute the QR factorization of C
2. Compute $\tilde{A} = AQ$
3. Solve $R^T\mathbf{u} = \mathbf{d}$
4. Solve the new least squares problem of minimizing $\|(\mathbf{b} - \tilde{A}_1\mathbf{u}) - \tilde{A}_2\mathbf{v}\|_2$
5. Compute
$$\mathbf{x} = Q \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix}.$$

This approach has the advantage that there are fewer unknowns in each system that needs to be solved, and also that $\kappa(\tilde{A}_2) \leq \kappa(A)$. The drawback is that sparsity can be destroyed.

3.5.2 Quadratic Constraints

We wish to solve the problem

$$\|\mathbf{b} - A\mathbf{x}\|_2 = \min, \quad \|\mathbf{x}\|_2 = \alpha, \quad \alpha \leq \|A^+\mathbf{b}\|_2.$$

This problem is known as *least squares with quadratic constraints.* To solve this problem, we define

$$\varphi(\mathbf{x}; \mu) = \|\mathbf{b} - A\mathbf{x}\|_2^2 + \mu(\|\mathbf{x}\|^2 - \alpha^2)$$

and seek to minimize φ. From

$$\nabla\varphi = 2A^T\mathbf{b} - 2A^TA\mathbf{x} + 2\mu\mathbf{x}$$

we obtain the system

$$(A^TA + \mu I)\mathbf{x} = A^T\mathbf{b}.$$

If we denote the eigenvalues of A^TA by

$$\lambda_i(A^TA) = \lambda_1, \ldots, \lambda_n, \quad \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$$

then

$$\lambda_i(A^TA+\mu I)=\lambda_1+\mu,\cdots,\lambda_n+\mu.$$

If $\mu \geq 0$, then $\kappa(A^TA+\mu I)\leq \kappa(A^TA)$, because

$$\frac{\lambda_1+\mu}{\lambda_n+\mu}\leq\frac{\lambda_1}{\lambda_n},$$

so $A^TA+\mu I$ is better conditioned.

Solving the least squares problem with quadratic constraints arises in many literatures, including

1. Statistics: Ridge Regression
2. Regularization: Tikhonov
3. Generalized cross-validation (GCV)

To solve this problem, we see that we need to compute

$$\mathbf{x}=(A^TA+\mu I)^{-1}A^T\mathbf{b}$$

where

$$\mathbf{x}^T\mathbf{x}=\mathbf{b}^TA(A^TA+\mu I)^{-2}A^T\mathbf{b}=\alpha^2.$$

If $A=U\Sigma V^T$ is the SVD of A, then we have

$$\begin{aligned}\alpha^2 &= \mathbf{b}^TU\Sigma V^T(V\Sigma^T\Sigma V^T+\mu I)^{-2}V\Sigma^TU^T\mathbf{b}\\ &= \mathbf{c}^T\Sigma(\Sigma^T\Sigma+\mu I)^{-2}\Sigma^T\mathbf{c},\quad U^T\mathbf{b}=\mathbf{c}\\ &= \sum_{i=1}^r\frac{c_i^2\sigma_i^2}{(\sigma_i^2+\mu)^2}\\ &= \chi(\mu)\end{aligned}$$

The function $\chi(\mu)$ has poles at $-\sigma_i^2$ for $i=1,\ldots,r$. Furthermore, $\chi(\mu)\to 0$ as $\mu\to\infty$.

We now have the following procedure for solving this problem, given A, $\mathbf{b}$, and α^2:

1. Compute the SVD of A to obtain $A=U\Sigma V^T$.
2. Compute $\mathbf{c}=U^T\mathbf{b}$.
3. Solve $\chi(\mu^*)=\alpha^2$ where $\mu^*\geq 0$. Don't use Newton's method on this equation directly; solving $1/\chi(\mu)=1/\alpha^2$ is much better.
4. Use the SVD to compute

$$\mathbf{x}=(A^TA+\mu I)^{-1}A^T\mathbf{b}=V(\Sigma^T\Sigma+\mu I)^{-1}\Sigma^TU^T\mathbf{b}.$$

3.6 Total Least Squares

In the ordinary least squares problem, we are solving

$$A\mathbf{x} = \mathbf{b} + \mathbf{r}, \quad \|\mathbf{r}\|_2 = \min .$$

In the *total least squares* problem, we wish to solve

$$(A + E)\mathbf{x} = \mathbf{b} + \mathbf{r}, \quad \|E\|_F^2 + \lambda^2\|\mathbf{r}\|_2^2 = \min .$$

From $A\mathbf{x} - \mathbf{b} + E\mathbf{x} - \mathbf{r} = \mathbf{0}$, we obtain the system

$$\left[\begin{array}{cc} A & \mathbf{b} \end{array}\right] \left[\begin{array}{c} \mathbf{x} \\ -1 \end{array}\right] + \left[\begin{array}{cc} E & \mathbf{r} \end{array}\right] \left[\begin{array}{c} \mathbf{x} \\ -1 \end{array}\right] = \mathbf{0},$$

or

$$(C + F)\mathbf{z} = \mathbf{0}.$$

We need the matrix $C + F$ to have rank $< n + 1$, and we want to minimize $\|F\|_F$.

To solve this problem, we compute the SVD of $C = \left[\begin{array}{cc} A & \mathbf{b} \end{array}\right] = U\Sigma V^T$. Let $\hat{C} = U\Omega_n V^T$. Then, if $\mathbf{v}_i$ is the ith column of V, we have

$$\hat{C}\mathbf{v}_{n+1} = U\Omega_n V^T \mathbf{v}_{n+1} = \mathbf{0}.$$

Our solution is

$$\left[\begin{array}{c} \hat{\mathbf{x}} \\ -1 \end{array}\right] = -\frac{1}{v_{n+1,n+1}}\mathbf{v}_{n+1}$$

provided that $v_{n+1,n+1} \neq 0$.

Now, suppose that only some of the data is contaminated, i.e. $E = \left[\begin{array}{cc} 0 & E_1 \end{array}\right]$ where the first p columns of E are zero. Then, in solving $(C + F)\mathbf{z} = \mathbf{0}$, we use Householder transformations to compute $Q^T(C+F)$ where the first p columns are zero below the diagonal. Since $\|F\|_F = \|Q^T F\|_F$, we then have a block upper triangular system

$$\left[\begin{array}{cc} R_{11} & R_{12} + F_{12} \\ 0 & R_{22} + F_{22} \end{array}\right] \mathbf{z} = \mathbf{0}, \quad \mathbf{z} = \left[\begin{array}{c} \mathbf{u} \\ \mathbf{v} \end{array}\right].$$

We can find the total least squares solution of

$$(R_{22} + F_{22})\mathbf{v} = \mathbf{0},$$

and then set $F_{12} = 0$ and solve

$$R_{11}\mathbf{u} + R_{12}\mathbf{v} = \mathbf{0}.$$

Chapter 4

Eigenvalue Problems

4.1 Eigenvalues and Eigenvectors

4.1.1 Definitions and Properties

Let A be an $n \times n$ matrix. A *nonzero* vector $\mathbf{x}$ is called an *eigenvector* of A if there exists a scalar λ such that

$$A\mathbf{x} = \lambda\mathbf{x}.$$

The scalar λ is called an *eigenvalue* of A, and we say that $\mathbf{x}$ is an eigenvector of A *corresponding* to λ. We see that an eigenvector of A is a vector for which matrix-vector multiplication with A is equivalent to scalar multiplication by λ.

We say that a nonzero vector $\mathbf{y}$ is a *left eigenvector* of A if there exists a scalar λ such that

$$\lambda\mathbf{y}^H = \mathbf{y}^H A.$$

The superscript H refers to the *Hermitian transpose*, which includes transposition and complex conjugation. That is, for any matrix A, $A^H = \overline{A^T}$. An eigenvector of A, as defined above, is sometimes called a *right eigenvector* of A, to distinguish from a left eigenvector. It can be seen that if $\mathbf{y}$ is a left eigenvector of A with eigenvalue λ, then $\mathbf{y}$ is also a right eigenvector of A^H, with eigenvalue $\overline{\lambda}$.

Because $\mathbf{x}$ is nonzero, it follows that if $\mathbf{x}$ is an eigenvector of A, then the matrix $A - \lambda I$ is singular, where λ is the corresponding eigenvalue. Therefore, λ satisfies the equation

$$\det(A - \lambda I) = 0.$$

The expression $\det(A-\lambda I)$ is a polynomial of degree n in λ, and therefore is called the *characteristic polynomial* of A (eigenvalues are sometimes called *characteristic values*). It follows from the fact that the eigenvalues of A are the roots of the characteristic polynomial that A has n eigenvalues, which can repeat, and can also be complex, even if A is real. However, if A is real, any complex eigenvalues must occur in complex-conjugate pairs.

The set of eigenvalues of A is called the *spectrum* of A, and denoted by $\lambda(A)$. This terminology explains why the magnitude of the largest eigenvalues is called the *spectral radius* of A. The *trace* of A, denoted by $\mathrm{tr}(A)$, is the sum of the diagonal elements of A. It is also equal to the sum of the eigenvalues of A. Furthermore, $\det(A)$ is equal to the *product* of the eigenvalues of A.

Example A 2×2 matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

has trace $\mathrm{tr}(A) = a + d$ and determinant $\det(A) = ad - bc$. Its characteristic polynomial is

$$\begin{aligned} \det(A - \lambda I) &= \begin{vmatrix} a - \lambda & b \\ c & d - \lambda \end{vmatrix} \\ &= (a - \lambda)(d - \lambda) - bc = \lambda^2 - (a + d)\lambda + (ad - bc) \\ &= \lambda^2 - \mathrm{tr}(A)\lambda + \det(A). \end{aligned}$$

From the quadratic formula, the eigenvalues are

$$\lambda_1 = \frac{a + d}{2} + \frac{\sqrt{(a - d)^2 + 4bc}}{2}, \quad \lambda_2 = \frac{a + d}{2} - \frac{\sqrt{(a - d)^2 + 4bc}}{2}.$$

It can be verified directly that the sum of these eigenvalues is equal to $\mathrm{tr}(A)$, and that their product is equal to $\det(A)$. □

4.1.2 Decompositions

A subspace W of $\mathbb{R}^n$ is called an *invariant subspace* of A if, for any vector $\mathbf{x} \in W$, $A\mathbf{x} \in W$. Suppose that $\dim(W) = k$, and let X be an $n \times k$ matrix such that $\mathrm{range}(X) = W$. Then, because each column of X is a vector in W, each column of AX is also a vector in W, and therefore is a linear combination of the columns of X. It follows that $AX = XB$, where B is a $k \times k$ matrix.

Now, suppose that $\mathbf{y}$ is an eigenvector of B, with eigenvalue λ. It follows from $B\mathbf{y} = \lambda\mathbf{y}$ that

$$XB\mathbf{y} = X(B\mathbf{y}) = X(\lambda\mathbf{y}) = \lambda X\mathbf{y},$$

but we also have

$$XB\mathbf{y} = (XB)\mathbf{y} = AX\mathbf{y}.$$

Therefore, we have

$$A(X\mathbf{y}) = \lambda(X\mathbf{y}),$$

which implies that λ is also an eigenvalue of A, with corresponding eigenvector $X\mathbf{y}$. We conclude that $\lambda(B) \subseteq \lambda(A)$.

If $k = n$, then X is an $n \times n$ invertible matrix, and it follows that A and B have the same eigenvalues. Furthermore, from $AX = XB$, we now have $B = X^{-1}AX$. We say that A and B are *similar matrices*, and that B is a *similarity transformation* of A.

Similarity transformations are essential tools in algorithms for computing the eigenvalues of a matrix A, since the basic idea is to apply a sequence of similarity transformations to A in order to obtain a new matrix B whose eigenvalues are easily obtained. For example, suppose that B has a 2×2 block structure

$$B = \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix},$$

where B_{11} is $p \times p$ and B_{22} is $q \times q$.

Let $\mathbf{x} = \left[\begin{array}{cc} \mathbf{x}_1^T & \mathbf{x}_2^T \end{array}\right]^T$ be an eigenvector of B, where $\mathbf{x}_1 \in \mathbb{C}^p$ and $\mathbf{x}_2 \in \mathbb{C}^q$. Then, for some scalar $\lambda \in \lambda(B)$, we have

$$\left[\begin{array}{cc} B_{11} & B_{12} \\ 0 & B_{22} \end{array}\right] \left[\begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \end{array}\right] = \lambda \left[\begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \end{array}\right].$$

If $\mathbf{x}_2 \neq \mathbf{0}$, then $B_{22}\mathbf{x}_2 = \lambda\mathbf{x}_2$, and $\lambda \in \lambda(B_{22})$. But if $\mathbf{x}_2 = \mathbf{0}$, then $B_{11}\mathbf{x}_1 = \lambda\mathbf{x}_1$, and $\lambda \in \lambda(B_{11})$. It follows that, $\lambda(B) \subseteq \lambda(B_{11}) \cup \lambda(B_{22})$. However, $\lambda(B)$ and $\lambda(B_{11}) \cup \lambda(B_{22})$ have the same number of elements, so the two sets must be equal. Because A and B are similar, we conclude that

$$\lambda(A) = \lambda(B) = \lambda(B_{11}) \cup \lambda(B_{22}).$$

Therefore, if we can use similarity transformations to reduce A to such a block structure, the problem of computing the eigenvalues of A *decouples* into two smaller problems of computing the eigenvalues of B_{ii} for $i = 1, 2$. Using an inductive argument, it can be shown that if A is block upper-triangular, then the eigenvalues of A are equal to the union of the eigenvalues of the diagonal blocks. If each diagonal block is 1×1, then it follows that the eigenvalues of any upper-triangular matrix are the diagonal elements. The same is true of any lower-triangular matrix; in fact, it can be shown that because $\det(A) = \det(A^T)$, the eigenvalues of A^T are the same as the eigenvalues of A.

Example The matrix

$$A = \left[\begin{array}{rrrrr} 1 & -2 & 3 & -3 & 4 \\ 0 & 4 & -5 & 6 & -5 \\ 0 & 0 & 6 & -7 & 8 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & -8 & 9 \end{array}\right]$$

has eigenvalues 1, 4, 6, 7, and 9. This is because A has a block upper-triangular structure

$$A = \left[\begin{array}{cc} A_{11} & A_{12} \\ 0 & A_{22} \end{array}\right], \quad A_{11} = \left[\begin{array}{rrr} 1 & -2 & 3 \\ 0 & 4 & -5 \\ 0 & 0 & 6 \end{array}\right], \quad A_{22} = \left[\begin{array}{rr} 7 & 0 \\ -8 & 9 \end{array}\right].$$

Because both of these blocks are themselves triangular, their eigenvalues are equal to their diagonal elements, and the spectrum of A is the union of the spectra of these blocks. □

Suppose that $\mathbf{x}$ is a *normalized* eigenvector of A, with eigenvalue λ. Furthermore, suppose that P is a Householder reflection such that $P\mathbf{x} = \mathbf{e}_1$. Because P is symmetric *and* orthogonal, P is its own inverse, so $P\mathbf{e}_1 = \mathbf{x}$. It follows that the matrix P^TAP, which is a similarity transformation of A, satisfies

$$P^TAP\mathbf{e}_1 = P^TA\mathbf{x} = \lambda P^T\mathbf{x} = \lambda P\mathbf{x} = \lambda\mathbf{e}_1.$$

That is, $\mathbf{e}_1$ is an eigenvector of P^TAP with eigenvalue λ, and therefore P^TAP has the block structure

$$P^TAP = \left[\begin{array}{cc} \lambda & \mathbf{v}^T \\ \mathbf{0} & B \end{array}\right].$$

Therefore, $\lambda(A) = \{\lambda\} \cup \lambda(B)$, which means that we can now focus on the $(n-1) \times (n-1)$ matrix B to find the rest of the eigenvalues of A. This process of reducing the eigenvalue problem for A to that of B is called *deflation*.

Continuing this process, we obtain the *Schur Decomposition*

$$A = Q^H T Q$$

where T is an upper-triangular matrix whose diagonal elements are the eigenvalues of A, and Q is a *unitary* matrix, meaning that $Q^H Q = I$. That is, a unitary matrix is the generalization of a real orthogonal matrix to complex matrices. Every square matrix has a Schur decomposition.

The columns of Q are called *Schur vectors*. However, for a general matrix A, there is no relation between Schur vectors of A and eigenvectors of A, as each Schur vector $\mathbf{q}_j$ satisfies $A\mathbf{q}_j = AQ\mathbf{e}_j = QT\mathbf{e}_j$. That is, $A\mathbf{q}_j$ is a linear combination of $\mathbf{q}_1, \ldots, \mathbf{q}_j$. It follows that for $j = 1, 2, \ldots, n$, the first j Schur vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j$ span an invariant subspace of A.

The Schur vectors and eigenvectors of A are the same when A is a *normal* matrix, which means that $A^H A = AA^H$. Any symmetric or skew-symmetric matrix, for example, is normal. It can be shown that in this case, the normalized eigenvectors of A form an orthonormal basis for $\mathbb{R}^n$. It follows that if $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of A, with corresponding (orthonormal) eigenvectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$, then we have

$$AQ = QD, \quad Q = \begin{bmatrix} \mathbf{q}_1 & \cdots & \mathbf{q}_n \end{bmatrix}, \quad D = \operatorname{diag}(\lambda_1, \ldots, \lambda_n).$$

Because Q is a unitary matrix, it follows that

$$Q^H A Q = Q^H Q D = D,$$

and A is similar to a diagonal matrix. We say that A is *diagonalizable*. Furthermore, because D can be obtained from A by a similarity transformation involving a unitary matrix, we say that A is *unitarily diagonalizable*.

Even if A is not a normal matrix, it may be diagonalizable, meaning that there exists an invertible matrix P such that $P^{-1}AP = D$, where D is a diagonal matrix. If this is the case, then, because $AP = PD$, the columns of P are eigenvectors of A, and the rows of P^{-1} are eigenvectors of A^T (as well as the left eigenvectors of A, if P is real).

By definition, an eigenvalue of A corresponds to at least one eigenvector. Because any nonzero scalar multiple of an eigenvector is also an eigenvector, corresponding to the same eigenvalue, an eigenvalue actually corresponds to an *eigenspace*, which is the span of any set of eigenvectors corresponding to the same eigenvalue, and this eigenspace must have a dimension of at least one. Any invariant subspace of a diagonalizable matrix A is a union of eigenspaces.

Now, suppose that λ_1 and λ_2 are *distinct* eigenvalues, with corresponding eigenvectors $\mathbf{x}_1$ and $\mathbf{x}_2$, respectively. Furthermore, suppose that $\mathbf{x}_1$ and $\mathbf{x}_2$ are linearly *dependent*. This means that they must be parallel; that is, there exists a nonzero constant c such that $\mathbf{x}_2 = c\mathbf{x}_1$. However, this implies that $A\mathbf{x}_2 = \lambda_2\mathbf{x}_2$ and $A\mathbf{x}_2 = cA\mathbf{x}_1 = c\lambda_1\mathbf{x}_1 = \lambda_1\mathbf{x}_2$. However, because $\lambda_1 \neq \lambda_2$, this is a contradiction. Therefore, $\mathbf{x}_1$ and $\mathbf{x}_2$ must be linearly independent.

More generally, it can be shown, using an inductive argument, that a set of k eigenvectors corresponding to k distinct eigenvalues must be linearly independent. Suppose that $\mathbf{x}_1, \ldots, \mathbf{x}_k$ are eigenvectors of A, with distinct eigenvalues $\lambda_1, \ldots, \lambda_k$. Trivially, $\mathbf{x}_1$ is linearly independent. Using induction, we assume that we have shown that $\mathbf{x}_1, \ldots, \mathbf{x}_{k-1}$ are linearly independent, and show that $\mathbf{x}_1, \ldots, \mathbf{x}_k$ must be linearly independent as well. If they are not, then there must be constants $c_1, \ldots, c_{k-1}$, not all zero, such that

$$\mathbf{x}_k = c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \cdots + c_{k-1}\mathbf{x}_{k-1}.$$

Multiplying both sides by A yields

$$A\mathbf{x}_k = c_1\lambda_1\mathbf{x}_1 + c_2\lambda_2\mathbf{x}_2 + \cdots + c_{k-1}\lambda_{k-1}\mathbf{x}_{k-1},$$

because $A\mathbf{x}_i = \lambda_i\mathbf{x}_i$ for $i = 1, 2, \ldots, k-1$. However, because both sides are equal to $\mathbf{x}_k$, and $A\mathbf{x}_k = \lambda_k\mathbf{x}_k$, we also have

$$A\mathbf{x}_k = c_1\lambda_k\mathbf{x}_1 + c_2\lambda_k\mathbf{x}_2 + \cdots + c_{k-1}\lambda_k\mathbf{x}_{k-1}.$$

It follows that

$$c_1(\lambda_k - \lambda_1)\mathbf{x}_1 + c_2(\lambda_k - \lambda_2)\mathbf{x}_2 + \cdots + c_{k-1}(\lambda_k - \lambda_{k-1})\mathbf{x}_{k-1} = \mathbf{0}.$$

However, because the eigenvalues $\lambda_1, \ldots, \lambda_k$ are distinct, and not all of the coefficients $c_1, \ldots, c_{k-1}$ are zero, this means that we have a nontrivial linear combination of linearly independent vectors being equal to the zero vector, which is a contradiction. We conclude that eigenvectors corresponding to distinct eigenvalues are linearly independent.

It follows that if A has n distinct eigenvalues, then it has a set of n linearly independent eigenvectors. If X is a matrix whose columns are these eigenvectors, then $AX = XD$, where D is a diagonal matrix of the eigenvectors, and because the columns of X are linearly independent, X is invertible, and therefore $X^{-1}AX = D$, and A is diagonalizable.

Now, suppose that the eigenvalues of A are not distinct; that is, the characteristic polynomial has repeated roots. Then an eigenvalue with multiplicity m does not necessarily correspond to m linearly independent eigenvectors. The *algebraic multiplicity* of an eigenvalue λ is the number of times that λ occurs as a root of the characteristic polynomial. The *geometric multiplicity* of λ is the dimension of the eigenspace corresponding to λ, which is equal to the maximal size of a set of linearly independent eigenvectors corresponding to λ. The geometric multiplicity of an eigenvalue λ is always less than or equal to the algebraic multiplicity. When it is strictly less, then we say that the eigenvalue is *defective*. When both multiplicities are equal to one, then we say that the eigenvalue is *simple*.

The *Jordan canonical form* of an $n \times n$ matrix A is a decomposition that yields information about the eigenspaces of A. It has the form

$$A = XJX^{-1}$$

where J has the block diagonal structure

$$J = \begin{bmatrix} J_1 & 0 & \cdots & 0 \\ 0 & J_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & J_p \end{bmatrix}.$$

Each diagonal block J_p is a *Jordan block* that has the form

$$J_i = \begin{bmatrix} \lambda_i & 1 & & \\ & \lambda_i & \ddots & \\ & & \lambda_i & 1 \\ & & & \lambda_i \end{bmatrix}, \quad i = 1, 2, \ldots, p.$$

The number of Jordan blocks, p, is equal to the number of linearly independent eigenvectors of A. The diagonal element of J_i, λ_i, is an eigenvalue of A. The number of Jordan blocks associated with λ_i is equal to the geometric multiplicity of λ_i. The sum of the sizes of these blocks is equal to the algebraic multiplicity of λ_i. If A is diagonalizable, then each Jordan block is 1×1.

Example Consider a matrix with Jordan canonical form

$$J = \begin{bmatrix} 2 & 1 & 0 & & & \\ 0 & 2 & 1 & & & \\ 0 & 0 & 2 & & & \\ & & & 3 & 1 & \\ & & & 0 & 3 & \\ & & & & & 2 \end{bmatrix}.$$

The eigenvalues of this matrix are 2, with algebraic multiplicity 4, and 3, with algebraic multiplicity 2. The geometric multiplicity of the eigenvalue 2 is 2, because it is associated with two Jordan blocks. The geometric multiplicity of the eigenvalue 3 is 1, because it is associated with only one Jordan block. Therefore, there are a total of three linearly independent eigenvectors, and the matrix is not diagonalizable. □

The Jordan canonical form, while very informative about the eigensystem of A, is not practical to compute using floating-point arithmetic. This is due to the fact that while the eigenvalues of a matrix are continuous functions of its entries, the Jordan canonical form is not. If two computed eigenvalues are nearly equal, and their computed corresponding eigenvectors are nearly parallel, we do not know if they represent two distinct eigenvalues with linearly independent eigenvectors, or a multiple eigenvalue that could be defective.

4.1.3 Perturbation Theory

Just as the problem of solving a system of linear equations $A\mathbf{x} = \mathbf{b}$ can be sensitive to perturbations in the data, the problem of computing the eigenvalues of a matrix can also be sensitive to perturbations in the matrix. We will now obtain some results concerning the extent of this sensitivity.

Suppose that A is obtained by perturbing a diagonal matrix D by a matrix F whose diagonal entries are zero; that is, $A = D + F$. If λ is an eigenvalue of A with corresponding eigenvector $\mathbf{x}$, then we have

$$(D - \lambda I)\mathbf{x} + F\mathbf{x} = \mathbf{0}.$$

If λ is not equal to any of the diagonal entries of A, then $D - \lambda I$ is nonsingular and we have

$$\mathbf{x} = -(D - \lambda I)^{-1} F\mathbf{x}.$$

Taking ∞-norms of both sides, we obtain

$$\|\mathbf{x}\|_\infty = \|(D - \lambda I)^{-1} F\mathbf{x}\|_\infty \leq \|(D - \lambda I)^{-1} F\|_\infty \|\mathbf{x}\|_\infty,$$

which yields

$$\|(D - \lambda I)^{-1} F\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1, j \neq i}^{n} \frac{|f_{ij}|}{|d_{ii} - \lambda|} \geq 1.$$

It follows that for some i, $1 \leq i \leq n$, λ satisfies

$$|d_{ii} - \lambda| \leq \sum_{j=1, j \neq i}^{n} |f_{ij}|.$$

That is, λ lies within one of the *Gerschgorin circles* in the complex plane, that has center a_{ii} and radius

$$r_i = \sum_{j=1, j \neq i}^{n} |a_{ij}|.$$

This is result is known as the *Gerschgorin Circle Theorem*.

Example The eigenvalues of the matrix

$$A = \begin{bmatrix} -5 & -1 & 1 \\ -2 & 2 & -1 \\ 1 & -3 & 7 \end{bmatrix}$$

are

$$\lambda(A) = \{6.4971, 2.7930, -5.2902\}.$$

The Gerschgorin disks are

$$\begin{aligned} D_1 &= \{z \in \mathbb{C} \,|\, |z - 7| \leq 4\}, \\ D_2 &= \{z \in \mathbb{C} \,|\, |z - 2| \leq 3\}, \\ D_3 &= \{z \in \mathbb{C} \,|\, |z + 5| \leq 2\}. \end{aligned}$$

We see that each disk contains one eigenvalue. □

It is important to note that while there are n eigenvalues and n Gerschgorin disks, it is not necessarily true that each disk contains an eigenvalue. The Gerschgorin Circle Theorem only states that all of the eigenvalues are contained within the *union* of the disks.

Another useful sensitivity result that applies to diagonalizable matrices is the *Bauer-Fike Theorem*, which states that if

$$X^{-1}AX = \text{diag}(\lambda_1, \ldots, \lambda_n),$$

and μ is an eigenvalue of a perturbed matrix $A + E$, then

$$\min_{\lambda \in \lambda(A)} |\lambda - \mu| \leq \kappa_p(X) \|E\|_p.$$

That is, μ is within $\kappa_p(X)\|E\|_p$ of an eigenvalue of A. It follows that if A is "nearly non-diagonalizable", which can be the case if eigenvectors are nearly linearly dependent, then a small perturbation in A could still cause a large change in the eigenvalues.

It would be desirable to have a concrete measure of the sensitivity of an eigenvalue, just as we have the condition number that measures the sensitivity of a system of linear equations. To that end, we assume that λ is a simple eigenvalue of an $n \times n$ matrix A that has Jordan canonical form $J = X^{-1}AX$. Then, $\lambda = J_{ii}$ for some i, and $\mathbf{x}_i$, the ith column of X, is a corresponding right eigenvector.

If we define $Y = X^{-H} = (X^{-1})^H$, then $\mathbf{y}_i$ is a left eigenvector of A corresponding to λ. From $Y^H X = I$, it follows that $\mathbf{y}^H\mathbf{x} = 1$. We now let A, λ, and $\mathbf{x}$ be functions of a parameter ϵ that satisfy

$$A(\epsilon)\mathbf{x}(\epsilon) = \lambda(\epsilon)\mathbf{x}(\epsilon), \quad A(\epsilon) = A + \epsilon F, \quad \|F\|_2 = 1.$$

Differentiating with respect to ϵ, and evaluating at $\epsilon = 0$, yields

$$F\mathbf{x} + A\mathbf{x}'(0) = \lambda\mathbf{x}'(0) + \lambda'(0)\mathbf{x}.$$

Taking the inner product of both sides with $\mathbf{y}$ yields

$$\mathbf{y}^H F\mathbf{x} + \mathbf{y}^H A\mathbf{x}'(0) = \lambda\mathbf{y}^H\mathbf{x}'(0) + \lambda'(0)\mathbf{y}^H\mathbf{x}.$$

Because $\mathbf{y}$ is a left eigenvector corresponding to λ, and $\mathbf{y}^H\mathbf{x} = 1$, we have

$$\mathbf{y}^H F\mathbf{x} + \lambda\mathbf{y}^H\mathbf{x}'(0) = \lambda\mathbf{y}^H\mathbf{x}'(0) + \lambda'(0).$$

We conclude that

$$|\lambda'(0)| = |\mathbf{y}^H F\mathbf{x}| \leq \|\mathbf{y}\|_2\|F\|_2\|\mathbf{x}\|_2 \leq \|\mathbf{y}\|_2\|\mathbf{x}\|_2.$$

However, because θ, the angle between $\mathbf{x}$ and $\mathbf{y}$, is given by

$$\cos\theta = \frac{\mathbf{y}^H\mathbf{x}}{\|\mathbf{y}\|_2\|\mathbf{x}\|_2} = \frac{1}{\|\mathbf{y}\|_2\|\mathbf{x}\|_2},$$

it follows that

$$|\lambda'(0)| \leq \frac{1}{|\cos\theta|}.$$

We define $1/|\cos\theta|$ to be the *condition number* of the simple eigenvalue λ. We require λ to be simple because otherwise, the angle between the left and right eigenvectors is not unique, because the eigenvectors themselves are not unique.

It should be noted that the condition number is also defined by $1/|\mathbf{y}^H\mathbf{x}|$, where $\mathbf{x}$ and $\mathbf{y}$ are normalized so that $\|\mathbf{x}\|_2 = \|\mathbf{y}\|_2 = 1$, but either way, the condition number is equal to $1/|\cos\theta|$. The interpretation of the condition number is that an $O(\epsilon)$ perturbation in A can cause an $O(\epsilon/|\cos\theta|)$ perturbation in the eigenvalue λ. Therefore, if $\mathbf{x}$ and $\mathbf{y}$ are nearly orthogonal, a large change in the eigenvalue can occur. Furthermore, if the condition number is large, then A is close to a matrix with a multiple eigenvalue.

Example The matrix

$$A = \begin{bmatrix} 3.1482 & -0.2017 & -0.5363 \\ 0.4196 & 0.5171 & 1.0888 \\ 0.3658 & -1.7169 & 3.3361 \end{bmatrix}$$

has a simple eigenvalue $\lambda = 1.9833$ with left and right eigenvectors

$$\mathbf{x} = \begin{bmatrix} 0.4150 \\ 0.6160 \\ 0.6696 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} -7.9435 \\ 83.0701 \\ -70.0066 \end{bmatrix},$$

such that $\mathbf{y}^H\mathbf{x} = 1$. It follows that the condition number of this eigenvalue is $\|\mathbf{x}\|_2\|\mathbf{y}\|_2 = 108.925$. In fact, the nearby matrix

$$B = \begin{bmatrix} 3.1477 & -0.2023 & -0.5366 \\ 0.4187 & 0.5169 & 1.0883 \\ 0.3654 & -1.7176 & 3.3354 \end{bmatrix}$$

has a double eigenvalue that is equal to 2. □

We now consider the sensitivity of repeated eigenvalues. First, it is important to note that while the eigenvalues of a matrix A are continuous functions of the entries of A, they are not necessarily differentiable functions of the entries. To see this, we consider the matrix

$$A = \begin{bmatrix} 1 & a \\ \epsilon & 1 \end{bmatrix},$$

where $a > 0$. Computing its characteristic polynomial

$$\det(A - \lambda I) = \lambda^2 - 2\lambda + 1 - a\epsilon$$

and computings its roots yields the eigenvalues $\lambda = 1 \pm \sqrt{a\epsilon}$. Differentiating these eigenvalues with respect to ϵ yields

$$\frac{d\lambda}{d\epsilon} = \pm\sqrt{\frac{a}{\epsilon}},$$

which is undefined at $\epsilon = 0$. In general, an $O(\epsilon)$ perturbation in A causes an $O(\epsilon^{1/p})$ perturbation in an eigenvalue associated with a $p \times p$ Jordan block, meaning that the "more defective" an eigenvalue is, the more sensitive it is.

We now consider the sensitivity of eigenvectors, or, more generally, invariant subspaces of a matrix A, such as a subspace spanned by the first k Schur vectors, which are the first k columns in a matrix Q such that Q^HAQ is upper triangular. Suppose that an $n \times n$ matrix A has the Schur decomposition

$$A = QTQ^H, \quad Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}, \quad T = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix},$$

where Q_1 is $n \times r$ and T_{11} is $r \times r$. We define the *separation* between the matrices T_{11} and T_{22} by

$$\text{sep}(T_{11}, T_{22}) = \min_{X \neq 0} \frac{\|T_{11}X - XT_{22}\|_F}{\|X\|_F}.$$

It can be shown that an $O(\epsilon)$ perturbation in A causes a $O(\epsilon/\text{sep}(T_{11}, T_{22}))$ perturbation in the invariant subspace Q_1.

We now consider the case where $r = 1$, meaning that Q_1 is actually a vector $\mathbf{q}_1$, that is also an eigenvector, and T_{11} is the corresponding eigenvalue, λ. Then, we have

$$\begin{aligned} \text{sep}(\lambda, T_{22}) &= \min_{X \neq 0} \frac{\|\lambda X - XT_{22}\|_F}{\|X\|_F} \\ &= \min_{\|\mathbf{y}\|_2 = 1} \|\mathbf{y}^H(T_{22} - \lambda I)\|_2 \\ &= \min_{\|\mathbf{y}\|_2 = 1} \|(T_{22} - \lambda I)^H\mathbf{y}\|_2 \\ &= \sigma_{\min}((T_{22} - \lambda_I)^H) \\ &= \sigma_{\min}(T_{22} - \lambda I), \end{aligned}$$

since the Frobenius norm of a vector is equivalent to the vector 2-norm. Because the smallest singular value indicates the distance to a singular matrix, $\mathrm{sep}(\lambda, T_{22})$ provides a measure of the separation of λ from the other eigenvalues of A. It follows that eigenvectors are more sensitive to perturbation if the corresponding eigenvalues are clustered near one another. That is, eigenvectors associated with nearby eigenvalues are "wobbly".

It should be emphasized that there is no direct relationship between the sensitivity of an eigenvalue and the sensitivity of its corresponding invariant subspace. The sensitivity of a simple eigenvalue depends on the angle between its left and right eigenvectors, while the sensitivity of an invariant subspace depends on the clustering of the eigenvalues. Therefore, a sensitive eigenvalue, that is nearly defective, can be associated with an insensitive invariant subspace, if it is distant from other eigenvalues, while an insensitive eigenvalue can have a sensitive invariant subspace if it is very close to other eigenvalues.

4.2 Power Iterations

4.2.1 The Power Method

We now consider the problem of computing eigenvalues of an $n \times n$ matrix A. For simplicity, we assume that A has eigenvalues $\lambda_1, \ldots, \lambda_n$ such that

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|.$$

We also assume that A is diagonalizable, meaning that it has n linearly independent eigenvectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ such that $A\mathbf{x}_i = \lambda_i \mathbf{x}_i$ for $i = 1, \ldots, n$.

Suppose that we continually multiply a given vector $\mathbf{x}^{(0)}$ by A, generating a sequence of vectors $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$ defined by

$$\mathbf{x}^{(k)} = A\mathbf{x}^{(k-1)} = A^k \mathbf{x}^{(0)}, \quad k = 1, 2, \ldots.$$

Because A is diagonalizable, any vector in $\mathbb{R}^n$ is a linear combination of the eigenvectors, and therefore we can write

$$\mathbf{x}^{(0)} = c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \cdots + c_n\mathbf{x}_n.$$

We then have

$$\begin{aligned} \mathbf{x}^{(k)} &= A^k \mathbf{x}^{(0)} \\ &= \sum_{i=1}^n c_i A^k \mathbf{x}_i \\ &= \sum_{i=1}^n c_i \lambda_i^k \mathbf{x}_i \\ &= \lambda_1^k \left[c_1\mathbf{x}_1 + \sum_{i=2}^n c_i \left(\frac{\lambda_i}{\lambda_1}\right)^k \mathbf{x}_i \right]. \end{aligned}$$

Because $|\lambda_1| > |\lambda_i|$ for $i = 2, \ldots, n$, it follows that the coefficients of $\mathbf{x}_i$, for $i = 2, \ldots, n$, converge to zero as $k \to \infty$. Therefore, the direction of $\mathbf{x}^{(k)}$ converges to that of $\mathbf{x}_1$. This leads to the most basic method of computing an eigenvalue and eigenvector, the *Power Method:*

```
Choose an initial vector $\mathbf{q}_0$ such that $\|\mathbf{q}_0\|_2 = 1$
for $k = 1, 2, \ldots$ do
    $\mathbf{z}_k = A\mathbf{q}_{k-1}$
    $\mathbf{q}_k = \mathbf{z}_k / \|\mathbf{z}_k\|_2$
end
```

This algorithm continues until $\mathbf{q}_k$ converges to within some tolerance. If it converges, it converges to a unit vector that is a scalar multiple of $\mathbf{x}_1$, an eigenvector corresponding to the largest eigenvalue, λ_1. The rate of convergence is $|\lambda_1/\lambda_2|$, meaning that the distance between $\mathbf{q}_k$ and a vector parallel to $\mathbf{x}_1$ decreases by roughly this factor from iteration to iteration.

It follows that convergence can be slow if λ_2 is almost as large as λ_1, and in fact, the power method fails to converge if $|\lambda_2| = |\lambda_1|$, but $\lambda_2 \neq \lambda_1$ (for example, if they have opposite signs). It is worth noting the implementation detail that if λ_1 is negative, for example, it may appear that $\mathbf{q}_k$ is not converging, as it "flip-flops" between two vectors. This is remedied by normalizing $\mathbf{q}_k$ so that it is not only a unit vector, but also a positive number.

Once the normalized eigenvector $\mathbf{x}_1$ is found, the corresponding eigenvalue λ_1 can be computed using a Rayleigh quotient. Then, *deflation* can be carried out by constructing a Householder reflection P_1 so that $P_1\mathbf{x}_1 = \mathbf{e}_1$, as discussed previously, and then P_1AP_1 is a matrix with block upper-triangular structure. This decouples the problem of computing the eigenvalues of A into the (solved) problem of computing λ_1, and then computing the remaining eigenvalues by focusing on the lower right $(n-1) \times (n-1)$ submatrix.

4.2.2 Orthogonal Iteration

This method can be impractical, however, due to the contamination of smaller eigenvalues by roundoff error from computing the larger ones and then deflating. An alternative is to compute several eigenvalues "at once" by using a block version of the Power Method, called *Orthogonal Iteration*. In this method, A is multiplied by an $n \times r$ matrix, with $r > 1$, and then the normalization of the vector computed by the power method is generalized to the *orthogonalization* of the block, through the QR factorization. The method is as follows:

```
Choose an $n \times r$ matrix $Q_0$ such that $Q_0^H Q_0 = I_r$
for $k = 1, 2, \ldots$ do
    $Z_k = AQ_{k-1}$
    $Z_k = Q_k R_k$ (QR Factorization)
end
```

Generally, this method computes a convergent sequence $\{Q_k\}$, as long as Q_0 is not deficient in the directions of certain eigenvectors of A^H, and $|\lambda_r| > |\lambda_{r+1}|$. From the relationship

$$R_k = Q_k^H Z_k = Q_k^H A Q_{k-1},$$

we see that if Q_k converges to a matrix Q, then $Q^H AQ = R$ is upper-triangular, and because $AQ = QR$, the columns of Q span an invariant subspace.

Furthermore, if $Q^\perp$ is a matrix whose columns span $(\text{range}(Q))^\perp$, then

$$\begin{aligned}\begin{bmatrix} Q^H \\ (Q^\perp)^H \end{bmatrix} A \begin{bmatrix} Q & Q^\perp \end{bmatrix} &= \begin{bmatrix} Q^H AQ & Q^H AQ^\perp \\ (Q^\perp)^H AQ & (Q^\perp)^H AQ^\perp \end{bmatrix} \\ &= \begin{bmatrix} R & Q^H AQ^\perp \\ 0 & (Q^\perp)^H AQ^\perp \end{bmatrix}.\end{aligned}$$

That is, $\lambda(A) = \lambda(R) \cup \lambda((Q^\perp)^H AQ^\perp)$, and because R is upper-triangular, the eigenvalues of R are its diagonal elements. We conclude that Orthogonal Iteration, when it converges, yields the largest r eigenvalues of A.

4.2.3 Inverse Iteration

4.3 The QR Algorithm

If we let $r = n$, then, if the eigenvalues of A are separated in magnitude, then generally Orthogonal Iteration converges, yielding the Schur Decomposition of A, $A = QTQ^H$. However, this convergence is generally quite slow. Before determining how convergence can be accelerated, we examine this instance of Orthogonal Iteration more closely.

For each integer k, we define $T_k = Q_k^H AQ_k$. Then, from the algorithm for Orthogonal Iteration, we have

$$T_{k-1} = Q_{k-1}^H AQ_{k-1} = Q_{k-1}^H Z_k = (Q_{k-1}^H Q_k)R_k,$$

and

$$\begin{aligned} T_k &= Q_k^H AQ_k \\ &= Q_k^H AQ_{k-1}Q_{k-1}^H Q_k \\ &= Q_k^H Z_k Q_{k-1}^H Q_k \\ &= Q_k^H Q_k R_k (Q_{k-1}^H Q_k) \\ &= R_k (Q_{k-1}^H Q_k). \end{aligned}$$

That is, T_k is obtained from T_{k-1} by computing the QR factorization of T_{k-1}, and then multiplying the factors in reverse order. Equivalently, T_k is obtained by applying a unitary similarity transformation to T_{k-1}, as

$$T_k = R_k(Q_{k-1}^H Q_k) = (Q_{k-1}^H Q_k)^H T_{k-1}(Q_{k-1}^H Q_k) = U_k^H T_{k-1} U_k.$$

If Orthogonal Iteration converges, then T_k converges to an upper-triangular matrix $T = Q^H AQ$ whose diagonal elements are the eigenvalues of A. This simple process of repeatedly computing the QR factorization and multiplying the factors in reverse order is called the *QR Iteration*, which proceeds as follows:

```
Choose $Q_0$ so that $Q_0^H Q_0 = I_n$ $T_0 = Q_0^H AQ_0$
for $k = 1, 2, \ldots$ do
    $T_{k-1} = U_k R_k$ ($QR$ factorization)
    $T_k = R_k U_k$
end
```

It is this version of Orthogonal Iteration that serves as the cornerstone of an efficient algorithm for computing all of the eigenvalues of a matrix. As described, QR iteration is prohibitively expensive, because $O(n^3)$ operations are required in *each* iteration to compute the QR factorization of T_{k-1}, and typically, many iterations are needed to obtain convergence. However, we will see that with a judicious choice of Q_0, the amount of computational effort can be drastically reduced.

It should be noted that if A is a real matrix with complex eigenvalues, then Orthogonal Iteration or the QR Iteration will not converge, due to distinct eigenvalues having equal magnitude. However, the *structure* of the matrix T_k in QR Iteration generally will converge to "quasi-upper-triangular" form, with 1×1 or 2×2 diagonal blocks corresponding to real eigenvalues or complex-conjugate pairs of eigenvalues, respectively. It is this type of convergence that we will seek in our continued development of the QR Iteration.

4.3.1 Hessenberg Reduction

Let A be a real $n \times n$ matrix. It is possible that A has complex eigenvalues, which must occur in complex-conjugate pairs, meaning that if $a + ib$ is an eigenvalue, where a and b are real, then so is $a - ib$. On the one hand, it is preferable that complex arithmetic be avoided as much as possible when using QR iteration to obtain the Schur Decomposition of A. On the other hand, in the algorithm for QR iteration, if the matrix Q_0 used to compute $T_0 = Q_0^H A Q_0$ is real, then every matrix T_k generated by the iteration will also be real, so it will not be possible to obtain the Schur Decomposition.

We compromise by instead seeking to compute the *Real Schur Decomposition* $A = QTQ^T$ where Q is a real, orthogonal matrix and T is a real, *quasi-upper-triangular* matrix that has a *block* upper-triangular structure

$$T = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1p} \\ 0 & T_{22} & \ddots & \vdots \\ 0 & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & T_{pp} \end{bmatrix},$$

where each diagonal block T_{ii} is 1×1, corresponding to a real eigenvalue, or a 2×2 block, corresponding to a pair of complex eigenvalues that are conjugates of one another.

If QR iteration is applied to such a matrix, then the sequence $\{T_k\}$ will not converge, but a block upper-triangular structure will be obtained, which can then be used to compute all of the eigenvalues. Therefore, the iteration can be terminated when appropriate entries below the diagonal have been made sufficiently small.

However, one significant drawback to the QR iteration is that each iteration is too expensive, as it requires $O(n^3)$ operations to compute the QR factorization, and to multiply the factors in reverse order. Therefore, it is desirable to first use a similarity transformation $H = U^T AU$ to reduce A to a form for which the QR factorization and matrix multiplication can be performed more efficiently.

Suppose that U^T includes a Householder reflection, or a product of Givens rotations, that transforms the first column of A to a multiple of $\mathbf{e}_1$, as in algorithms to compute the QR factorization. Then U operates on all rows of A, so when U is applied to the columns of A, to complete the similarity transformation, it affects all columns. Therefore, the work of zeroing the elements of the first column of A is undone.

Now, suppose that instead, U^T is designed to zero all elements of the first column except the first *two*. Then, U^T affects all rows except the first, meaning that when U^TA is multiplied by U on the right, the first column is *unaffected*. Continuing this reasoning with subsequent columns of A, we see that a sequence of orthogonal transformations can be used to reduce A to an *upper Hessenberg* matrix H, in which $h_{ij} = 0$ whenever $i > j+1$. That is, all entries below the *subdiagonal* are equal to zero.

It is particularly efficient to compute the QR factorization of an upper Hessenberg, or simply Hessenberg, matrix, because it is only necessary to zero one element in each column. Therefore, it can be accomplished with a sequence of $n-1$ Givens row rotations, which requires only $O(n^2)$ operations. Then, these same Givens rotations can be applied, in the same order, to the columns in order to complete the similarity transformation, or, equivalently, accomplish the task of multiplying the factors of the QR factorization.

Specifically, given a Hessenberg matrix H, we apply Givens row rotations $G_1^T, G_2^T, \ldots, G_{n-1}^T$ to H, where G_i^T rotates rows i and $i+1$, to obtain

$$G_{n-1}^T \cdots G_2^T G_1^T H = (G_1 G_2 \cdots G_{n-1})^T H = Q^T H = R,$$

where R is upper-triangular. Then, we compute

$$\tilde{H} = Q^T H Q = RQ = RG_1 G_2 \cdots G_{n-1}$$

by applying column rotations to R, to obtain a new matrix $\tilde{H}$.

By considering which rows or columns the Givens rotations affect, it can be shown that Q is Hessenberg, and therefore $\tilde{H}$ is Hessenberg as well. The process of applying an orthogonal similarity transformation to a Hessenberg matrix to obtain a new Hessenberg matrix with the same eigenvalues that, hopefully, is closer to quasi-upper-triangular form is called a *Hessenberg QR step*. The following algorithm overwrites H with $\tilde{H} = RQ = Q^THQ$, and also computes Q as a product of Givens column rotations, which is only necessary if the full Schur Decomposition of A is required, as opposed to only the eigenvalues.

```
for $j = 1, 2, \ldots, n-1$ do
    $[c, s] = \texttt{givens}(h_{jj}, h_{j+1,j})$
    $G_j = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$
    $H(j : j+1, j : n) = G_j^T H(j : j+1, :)$
end
$Q = I$
for $j = 1, 2, \ldots, n-1$ do
    $H(1 : j+1, j : j+1) = H(1 : j+1, j : j+1) G_j$
    $Q(1 : j+1, j : j+1) = Q(1 : j+1, j : j+1) G_j$
end
```

The function `givens`(a, b) returns c and s such that

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \quad r = \sqrt{a^2 + b^2}.$$

Note that when performing row rotations, it is only necessary to update certain columns, and when performing column rotations, it is only necessary to update certain rows, because of the structure of the matrix at the time the rotation is performed; for example, after the first loop, H is upper-triangular.

Before a Hessenberg QR step can be performed, it is necessary to actually reduce the original matrix A to Hessenberg form $H = U^T AU$. This can be accomplished by performing a sequence of Householder reflections $U = P_1 P_2 \cdots P_{n-2}$ on the columns of A, as in the following algorithm.

```
$U = I$
for $j = 1, 2, \ldots, n-2$ do
    $\mathbf{v} = \texttt{house}(A(j+1:n, j))$, $c = 2/\mathbf{v}^T\mathbf{v}$
    $A(j+1:n, j:n) = A(j+1:n, j:n) - c\mathbf{v}\mathbf{v}^T A(j+1:n, j:n)$
    $A(1:n, j+1:n) = A(1:n, j+1:n) - cA(1:n, j+1:n)\mathbf{v}\mathbf{v}^T$
end
```

The function `house(x)` computes a vector $\mathbf{v}$ such that $P\mathbf{x} = I - c\mathbf{v}\mathbf{v}^T\mathbf{x} = \alpha\mathbf{e}_1$, where $c = 2/\mathbf{v}^T\mathbf{v}$ and $\alpha = \pm\|\mathbf{x}\|_2$. The algorithm for the Hessenberg reduction requires $O(n^3)$ operations, but it is performed only once, before the QR Iteration begins, so it still leads to a substantial reduction in the total number of operations that must be performed to compute the Schur Decomposition.

If a subdiagonal entry $h_{j+1,j}$ of a Hessenberg matrix H is equal to zero, then the problem of computing the eigenvalues of H decouples into two smaller problems of computing the eigenvalues of H_{11} and H_{22}, where

$$H = \begin{bmatrix} H_{11} & H_{12} \\ 0 & H_{22} \end{bmatrix}$$

and H_{11} is $j \times j$. Therefore, an efficient implementation of the QR Iteration on a Hessenberg matrix H focuses on a submatrix of H that is *unreduced*, meaning that all of its subdiagonal entries are nonzero. It is also important to monitor the subdiagonal entries after each iteration, to determine if any of them have become nearly zero, thus allowing further decoupling. Once no further decoupling is possible, H has been reduced to quasi-upper-triangular form and the QR Iteration can terminate.

It is essential to choose an *maximal* unreduced diagonal block of H for applying a Hessenberg QR step. That is, the step must be applied to a submatrix H_{22} such that H has the structure

$$H = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ 0 & H_{22} & H_{23} \\ 0 & 0 & H_{33} \end{bmatrix}$$

where H_{22} is unreduced. This condition ensures that the eigenvalues of H_{22} are also eigenvalues of H, as $\lambda(H) = \lambda(H_{11}) \cup \lambda(H_{22}) \cup \lambda(H_{33})$ when H is structured as above. Note that the size of either H_{11} or H_{33} may be 0×0.

The following property of unreduced Hessenberg matrices is useful for improving the efficiency of a Hessenberg QR step.

Theorem (Implicit Q Theorem) Let A be an $n \times n$ matrix, and let Q and V be $n \times n$ orthgonal matrices such that $Q^T AQ = H$ and $V^T AV = G$ are both upper Hessenberg, and H is unreduced. If $Q = \begin{bmatrix} \mathbf{q}_1 & \cdots & \mathbf{q}_n \end{bmatrix}$ and $V = \begin{bmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_n \end{bmatrix}$, and if $\mathbf{q}_1 = \mathbf{v}_1$, then $\mathbf{q}_i = \pm\mathbf{v}_i$ for $i = 2, \ldots, n$, and $|h_{ij}| = |g_{ij}|$ for $i, j = 1, 2, \ldots, n$.

That is, if two orthogonal similarity transformations that reduce A to Hessenberg form have the same first column, then they are "essentially equal", as are the Hessenberg matrices.

The proof of the Implicit Q Theorem proceeds as follows: From the relations $Q^TAQ = H$ and $V^TAV = G$, we obtain $GW = WH$, where $W = V^TQ$ is orthogonal. Because $\mathbf{q}_1 = \mathbf{v}_1$, we have $W\mathbf{e}_1 = \mathbf{e}_1$. Equating first columns of $GW = WH$, and keeping in mind that G and H are both upper Hessenberg, we find that only the first two elements of $W\mathbf{e}_2$ are nonzero. Proceeding by induction, it follows that W is upper triangular, and therefore W^{-1} is also upper triangular. However, because W is orthogonal, $W^{-1} = W^T$, which means that W^{-1} is lower triangular as well. Therefore, W is a diagonal matrix, so by the orthogonality of W, W must have diagonal entries that are equal to ± 1, and the theorem follows.

Another important property of an unreduced Hessenberg matrix is that all of its eigenvalues have a geometric multiplicity of one. To see this, consider the matrix $H - \lambda I$, where H is an $n \times n$ unreduced Hessenberg matrix and λ is an arbitrary scalar. If λ is not an eigenvalue of H, then H is nonsingular and $\text{rank}(H) = n$. Otherwise, because H is unreduced, from the structure of H it can be seen that the first $n-1$ columns of $H - \lambda I$ must be linearly independent. We conclude that $\text{rank}(H - \lambda I) = n - 1$, and therefore at most one vector $\mathbf{x}$ (up to a scalar multiple) satisfies the equation $H\mathbf{x} = \lambda\mathbf{x}$. That is, there can only be one linearly independent eigenvector. It follows that if any eigenvalue of H repeats, then it is defective.

4.3.2 Shifted QR Iteration

The efficiency of the QR Iteration for computing the eigenvalues of an $n \times n$ matrix A is significantly improved by first reducing A to a Hessenberg matrix H, so that only $O(n^2)$ operations per iteration are required, instead of $O(n^3)$. However, the iteration can still converges very slowly, so additional modifications are needed to make the QR Iteration a practical algorithm for computing the eigenvalues of a general matrix.

In general, the pth subdiagonal entry of H converges to zero at the rate

$$\left|\frac{\lambda_{p+1}}{\lambda_p}\right|,$$

where λ_p is the pth largest eigenvalue of A in magnitude. It follows that convergence can be particularly slow if eigenvalues are very close to one another in magnitude. Suppose that we *shift* H by a scalar μ, meaning that we compute the QR factorization of $H - \mu I$ instead of H, and then update H to obtain a new Hessenberg $\tilde{H}$ by multiplying the QR factors in reverse order as before, but then adding μI. Then, we have

$$\begin{aligned}
\tilde{H} &= RQ + \mu I \\
&= Q^T(H - \mu I)Q + \mu I \\
&= Q^THQ - \mu Q^TQ + \mu I \\
&= Q^THQ - \mu I + \mu I \\
&= Q^THQ.
\end{aligned}$$

So, we are still performing an orthogonal similarity transformation of H, but with a different Q. Then, the convergence rate becomes $|\lambda_{p+1} - \mu|/|\lambda_p - \mu|$. Then, if μ is close to an eigenvalue, convergence of a particular subdiagonal entry will be much more rapid.

In fact, suppose H is *unreduced*, and that μ happens to be an eigenvalue of H. When we compute the QR factorization of $H - \mu I$, which is now *singular*, then, because the first $n-1$ columns of $H - \mu I$ must be linearly independent, it follows that the first $n-1$ columns of R must be linearly independent as well, and therefore the last row of R must be zero. Then, when we compute RQ, which involves rotating columns of R, it follows that the last row of RQ must also be zero. We then add μI, but as this only changes the diagonal elements, we can conclude that $\tilde{h}_{n,n-1} = 0$. In other words, $\tilde{H}$ is not an unreduced Hessenberg matrix, and *deflation has occurred in one step.*

If μ is not an eigenvalue of H, but is still close to an eigenvalue, then $H - \mu I$ is nearly singular, which means that its columns are nearly linearly dependent. It follows that r_{nn} is small, and it can be shown that $\tilde{h}_{n,n-1}$ is also small, and $\tilde{h}_{nn} \approx \mu$. Therefore, the problem is nearly decoupled, and μ is revealed by the structure of $\tilde{H}$ as an approximate eigenvalue of H. This suggests using h_{nn} as the shift μ during each iteration, because if $h_{n,n-1}$ is small compare to h_{nn}, then this choice of shift will drive $h_{n,n-1}$ toward zero. In fact, it can be shown that this strategy generally causes $h_{n,n-1}$ to converge to zero *quadratically*, meaning that only a few similarity transformations are needed to achieve decoupling. This improvement over the linear convergence rate reported earlier is due to the changing of the shift during each step.

Example Consider the 2×2 matrix

$$H = \begin{bmatrix} a & b \\ \epsilon & 0 \end{bmatrix}, \quad \epsilon > 0,$$

that arises naturally when using h_{nn} as a shift. To compute its QR factorization of H, we perform a single Givens rotation to obtain $H = GR$, where

$$G = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}, \quad c = \frac{a}{\sqrt{a^2+\epsilon^2}}, \quad s = \frac{\epsilon}{\sqrt{a^2+\epsilon^2}}.$$

Performing the similarity transformation $\tilde{H} = G^T H G$ yields

$$\begin{aligned} \tilde{H} &= \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a & b \\ \epsilon & 0 \end{bmatrix} \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \\ &= \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} ac+bs & bc-as \\ \epsilon c & -\epsilon s \end{bmatrix} \\ &= \begin{bmatrix} ac^2+bcs+\epsilon cs & bc^2-acs-\epsilon s^2 \\ -acs-bs^2+\epsilon c^2 & -bcs+as^2-\epsilon cs \end{bmatrix} \\ &= \begin{bmatrix} a+bcs & bc^2-\epsilon \\ -bs^2 & -bcs \end{bmatrix}. \end{aligned}$$

We see that the one subdiagonal element is

$$-bs^2 = -b\frac{\epsilon^2}{\epsilon^2+a^2},$$

compared to the original element ϵ. It follows that if ϵ is small compared to a and b, then subsequent QR steps will cause the subdiagonal element to converge to zero quadratically. For example, if

$$H = \begin{bmatrix} 0.6324 & 0.2785 \\ 0.0975 & 0.5469 \end{bmatrix},$$

then the value of h_{21} after each of the first three QR steps is 0.1575, −0.0037, and 2.0876×10^{-5}. □

This shifting strategy is called the *single shift strategy*. Unfortunately, it is not very effective if H has complex eigenvalues. An alternative is the *double shift strategy*, which is used if the two eigenvalues, μ_1 and μ_2, of the lower-right 2×2 block of H are complex. Then, these two eigenvalues are used as shifts in consecutive iterations to achieve quadratic convergence in the complex case as well. That is, we compute

$$\begin{aligned} H - \mu_1 I &= U_1R_1 \\ H_1 &= R_1U_1 + \mu_1 I \\ H_1 - \mu_2 I &= U_2R_2 \\ H_2 &= R_2U_2 + \mu_2 I. \end{aligned}$$

To avoid complex arithmetic when using complex shifts, the *double implicit shift strategy* is used. We first note that

$$\begin{aligned} U_1U_2R_2R_1 &= U_1(H_1 - \mu_2 I)R_1 \\ &= U_1H_1R_1 - \mu_2U_1R_1 \\ &= U_1(R_1U_1 + \mu_1 I)R_1 - \mu_2(H - \mu_1 I) \\ &= U_1R_1U_1R_1 + \mu_1U_1R_1 - \mu_2(H - \mu_1 I) \\ &= (H - \mu_1 I)^2 + \mu_1(H - \mu_1 I) - \mu_2(H - \mu_1 I) \\ &= H^2 - 2\mu_1 H + \mu_1^2 I + \mu_1 H - \mu_1^2 I - \mu_2 H + \mu_1\mu_2 I \\ &= H^2 - (\mu_1 + \mu_2)H + \mu_1\mu_2 I. \end{aligned}$$

Since $\mu_1 = a + bi$ and $\mu_2 = a - bi$ are a complex-conjugate pair, it follows that $\mu_1 + \mu_2 = ab$ and $\mu_1\mu_2 = a^2 + b^2$ are real. Therefore, $U_1U_2R_2R_1 = (U_1U_2)(R_2R_1)$ represents the QR factorization of a real matrix.

Furthermore,

$$\begin{aligned} H_2 &= R_2U_2 + \mu_2 I \\ &= U_2^TU_2R_2U_2 + \mu_2U_2^TU_2 \\ &= U_2^T(U_2R_2 + \mu_2 I)U_2 \\ &= U_2^TH_1U_2 \\ &= U_2^T(R_1U_1 + \mu_1 I)U_2 \\ &= U_2^T(U_1^TU_1R_1U_1 + \mu_1U_1^TU_1)U_2 \\ &= U_2^TU_1^T(U_1R_1 + \mu_1 I)U_1U_2 \\ &= U_2^TU_1^THU_1U_2. \end{aligned}$$

That is, U_1U_2 is the orthogonal matrix that implements the similarity transformation of H to obtain H_2. Therefore, we could use exclusively real arithmetic by forming $M = H^2 - (\mu_1 + \mu_2)H + \mu_1\mu_2 I$, compute its QR factorization to obtain $M = ZR$, and then compute $H_2 = Z^THZ$, since $Z = U_1U_2$, in view of the uniqueness of the QR decomposition. However, M is computed by squaring H, which requires $O(n^3)$ operations. Therefore, this is not a practical approach.

We can work around this difficulty using the Implicit Q Theorem. Instead of forming M in its entirety, we only form its first column, which, being a second-degree polynomial of a Hessenberg matrix, has only three nonzero entries. We compute a Householder transformation P_0 that makes this first column a multiple of $\mathbf{e}_1$. Then, we compute P_0HP_0, which is no longer Hessenberg, because it operates on the first three rows and columns of H. Finally, we apply a series of Householder reflections $P_1, P_2, \ldots, P_{n-2}$ that restore Hessenberg form. Because these reflections are not applied to the first row or column, it follows that if we define $\tilde{Z} = P_0P_1P_2\cdots P_{n-2}$, then Z and $\tilde{Z}$ have the same first column. Since both matrices implement similarity transformations that preserve the Hessenberg form of H, it follows from the Implicit Q Theorem that Z and $\tilde{Z}$ are essentially equal, and that they essentially produce the same updated matrix H_2. This variation of a Hessenberg QR step is called a *Francis QR step.*

A Francis QR step requires $10n^2$ operations, with an additional $10n^2$ operations if orthogonal transformations are being accumulated to obtain the entire real Schur decomposition. Generally, the entire QR algorithm, including the initial reduction to Hessenberg form, requires about $10n^3$ operations, with an additional $15n^3$ operations to compute the orthogonal matrix Q such that $A = QTQ^T$ is the real Schur decomposition of A.

4.3.3 Computation of Eigenvectors

4.4 The Symmetric Eigenvalue Problem

4.4.1 Properties

The eigenvalue problem for a real, symmetric matrix A, or a complex, *Hermitian* matrix A, for which $A = A^H$, is a considerable simplification of the eigenvalue problem for a general matrix. Consider the Schur decomposition $A = QTQ^H$, where T is upper-triangular. Then, if A is Hermitian, it follows that $T = T^H$. But because T is upper-triangular, it follows that T must be diagonal. That is, any symmetric real matrix, or Hermitian complex matrix, is unitarily diagonalizable, as stated previously because A is normal. What's more, because the Hermitian transpose includes complex conjugation, T must equal its complex conjugate, which implies that the eigenvalues of A are real, even if A itself is complex.

Because the eigenvalues are real, we can order them. By convention, we prescribe that if A is an $n \times n$ symmetric matrix, then it has eigenvalues

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n.$$

Furthermore, by the *Courant-Fischer Minimax Theorem*, each of these eigenvalues has the following characterization:

$$\lambda_k = \max_{\dim(S)=k} \min_{\mathbf{y}\in S, \mathbf{y}\neq \mathbf{0}} \frac{\mathbf{y}^H A\mathbf{y}}{\mathbf{y}^H\mathbf{y}}.$$

That is, the kth largest eigenvalue of A is equal to the maximum, over all k-dimensional subspaces of $\mathbb{C}^n$, of the minimum value of the *Rayleigh quotient*

$$r(\mathbf{y}) = \frac{\mathbf{y}^H A\mathbf{y}}{\mathbf{y}^H\mathbf{y}}, \quad \mathbf{y} \neq \mathbf{0},$$

on each subspace. It follows that λ_1, the largest eigenvalue, is the absolute maximum value of the Rayleigh quotient on all of $\mathbb{C}^n$, while λ_n, the smallest eigenvalue, is the absolute minimum value.

In fact, by computing the gradient of $r(\mathbf{y})$, it can be shown that every eigenvector of A is a critical point of $r(\mathbf{y})$, with the corresponding eigenvalue being the value of $r(\mathbf{y})$ at that critical point.

4.4.2 Perturbation Theory

In the symmetric case, the Gerschgorin circles become Gerschgorin intervals, because the eigenvalues of a symmetric matrix are real.

Example The eigenvalues of the 3×3 symmetric matrix

$$A = \begin{bmatrix} -10 & -3 & 2 \\ -3 & 4 & -2 \\ 2 & -2 & 14 \end{bmatrix}$$

are

$$\lambda(A) = \{14.6515, 4.0638, -10.7153\}.$$

The Gerschgorin intervals are

$$\begin{aligned} D_1 &= \{x \in \mathbb{R} \,|\, |x - 14| \leq 4\}, \\ D_2 &= \{x \in \mathbb{R} \,|\, |x - 4| \leq 5\}, \\ D_3 &= \{x \in \mathbb{R} \,|\, |x + 10| \leq 5\}. \end{aligned}$$

We see that each intervals contains one eigenvalue. □

The characterization of the eigenvalues of a symmetric matrix as constrained maxima of the Rayleigh quotient lead to the following results about the eigenvalues of a perturbed symmetric matrix. As the eigenvalues are real, and therefore can be ordered, we denote by $\lambda_i(A)$ the ith largest eigenvalue of A.

Theorem (Wielandt-Hoffman) If A and $A + E$ are $n \times n$ symmetric matrices, then

$$\sum_{i=1}^{n} (\lambda_i(A + E) - \lambda_i(A))^2 \leq \|E\|_F^2.$$

It is also possible to bound the distance between individual eigenvalues of A and $A + E$.

Theorem If A and $A + E$ are $n \times n$ symmetric matrices, then

$$\lambda_n(E) \leq \lambda_k(A + E) - \lambda_k(A) \leq \lambda_1(E).$$

Furthermore,

$$|\lambda_k(A + E) - \lambda_k(A)| \leq \|E\|_2.$$

The second inequality in the above theorem follows directly from the first, as the 2-norm of the symmetric matrix E, being equal to its spectral radius, must be equal to the larger of the absolute value of $\lambda_1(E)$ or $\lambda_n(E)$.

Theorem (Interlacing Property) If A is an $n \times n$ symmetric matrix, and A_r is the $r \times r$ leading principal minor of A, then, for $r = 1, 2, \ldots, n-1$,

$$\lambda_{r+1}(A_{r+1}) \leq \lambda_r(A_r) \leq \lambda_r(A_{r+1}) \leq \cdots \leq \lambda_2(A_{r+1}) \leq \lambda_1(A_r) \leq \lambda_1(A_{r+1}).$$

For a symmetric matrix, or even a more general normal matrix, the left eigenvectors and right eigenvectors are the same, from which it follows that every simple eigenvalue is "perfectly conditioned"; that is, the condition number $1/|\cos\theta|$ is equal to 1 because $\theta = 0$ in this case. However, the same results concerning the sensitivity of invariant subspaces from the nonsymmetric case apply in the symmetric case as well: such sensitivity increases as the eigenvalues become more clustered, even though there is no chance of a defective eigenvalue. This is because for a nondefective, repeated eigenvalue, there are infinitely many possible bases of the corresponding invariant subspace. Therefore, as the eigenvalues approach one another, the eigenvectors become more sensitive to small perturbations, for any matrix.

Let Q_1 be an $n \times r$ matrix with orthonormal columns, meaning that $Q_1^T Q_1 = I_r$. If it spans an invariant subspace of an $n \times n$ symmetric matrix A, then $AQ_1 = Q_1 S$, where $S = Q_1^T A Q_1$. On the other hand, if range(Q_1) is *not* an invariant subspace, but the matrix

$$AQ_1 - Q_1 S = E_1$$

is small for any given $r \times r$ symmetric matrix S, then the columns of Q_1 define an *approximate* invariant subspace.

It turns out that $\|E_1\|_F$ is minimized by choosing $S = Q_1^T A Q_1$. Furthermore, we have

$$\|AQ_1 - Q_1 S\|_F = \|P_1^\perp A Q_1\|_F,$$

where $P_1^\perp = I - Q_1 Q_1^T$ is the orthogonal projection into $(\text{range}(Q_1))^\perp$, and there exist eigenvalues $\mu_1, \ldots, \mu_r \in \lambda(A)$ such that

$$|\mu_k - \lambda_k(S)| \leq \sqrt{2}\|E_1\|_2, \quad k = 1, \ldots, r.$$

That is, r eigenvalues of A are close to the eigenvalues of S, which are known as *Ritz values*, while the corresponding eigenvectors are called *Ritz vectors*. If $(\theta_k, \mathbf{y}_k)$ is an eigenvalue-eigenvector pair, or an *eigenpair* of S, then, because S is defined by $S = Q_1^T A Q_1$, it is also known as a *Ritz pair*. Furthermore, as θ_k is an approximate eigenvalue of A, $Q_1 \mathbf{y}_k$ is an approximate corresponding eigenvector.

To see this, let σ_k (not to be confused with singular values) be an eigenvalue of S, with eigenvector $\mathbf{y}_k$. We multiply both sides of the equation $S\mathbf{y}_k = \sigma_k \mathbf{y}_k$ by Q_1:

$$Q_1 S \mathbf{y}_k = \sigma_k Q_1 \mathbf{y}_k.$$

Then, we use the relation $AQ_1 - Q_1 S = E_1$ to obtain

$$(AQ_1 - E_1)\mathbf{y}_k = \sigma_k Q_1 \mathbf{y}_k.$$

Rearranging yields

$$A(Q_1 \mathbf{y}_k) = \sigma_k (Q_1 \mathbf{y}_k) + E_1 \mathbf{y}_k.$$

If we let $\mathbf{x}_k = Q_1 \mathbf{y}_k$, then we conclude

$$A\mathbf{x}_k = \sigma_k \mathbf{x}_k + E_1 \mathbf{y}_k.$$

Therefore, $\|E_1\|$ is small in some norm, $Q_1 \mathbf{y}_k$ is nearly an eigenvector.

4.4.3 Rayleigh Quotient Iteration

The Power Method, when applied to a symmetric matrix to obtain its largest eigenvalue, is more effective than for a general matrix: its rate of convergence $|\lambda_2/\lambda_1|^2$, meaning that it generally converges twice as rapidly.

Let A be an $n \times n$ symmetric matrix. Even more rapid convergence can be obtained if we consider a variation of the Power Method. *Inverse Iteration* is the Power Method applied to $(A - \mu I)^{-1}$. The algorithm is as follows:

```
Choose $\mathbf{x}_0$ so that $\|\mathbf{x}_0\|_2 = 1$
for $k = 0, 1, 2, \ldots$ do
    Solve $(A - \mu I)\mathbf{z}_k = \mathbf{x}_k$ for $\mathbf{z}_k$
    $\mathbf{x}_{k+1} = \mathbf{z}_k/\|\mathbf{z}_k\|_2$
end
```

Let A have eigenvalues $\lambda_1, \ldots, \lambda_n$. Then, the eigenvalues of $(A - \mu I)^{-1}$ matrix are $1/(\lambda_i - \mu)$, for $i - 1, 2, \ldots, n$. Therefore, this method finds the eigenvalue that is closest to μ.

Now, suppose that we vary μ from iteration to iteration, by setting it equal to the *Rayleigh quotient*

$$r(\mathbf{x}) = \frac{\mathbf{x}^H A\mathbf{x}}{\mathbf{x}^H\mathbf{x}},$$

of which the eigenvalues of A are constrained extrema. We then obtain *Rayleigh Quotient Iteration*:

```
Choose a vector $\mathbf{x}_0$, $\|\mathbf{x}_0\|_2 = 1$
for $k = 0, 1, 2, \ldots$ do
    $\mu_k = \mathbf{x}_k^H A\mathbf{x}_k$
    Solve $(A - \mu_k I)\mathbf{z}_k = \mathbf{x}_k$ for $\mathbf{z}_k$
    $\mathbf{x}_{k+1} = \mathbf{z}_k/\|\mathbf{z}_k\|_2$
end
```

When this method converges, it converges *cubically* to an eigenvalue-eigenvector pair. To see this, consider the diagonal 2×2 matrix

$$A = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}, \quad \lambda_1 > \lambda_2.$$

This matrix has eigenvalues λ_1 and λ_2, with eigenvectors $\mathbf{e}_1$ and $\mathbf{e}_2$. Suppose that $\mathbf{x}_k = \begin{bmatrix} c_k & s_k \end{bmatrix}^T$, where $c_k^2 + s_k^2 = 1$. Then we have

$$\mu_k = r(\mathbf{x}_k) = \begin{bmatrix} c_k & s_k \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} c_k \\ s_k \end{bmatrix} = \lambda_1 c_k^2 + \lambda_2 s_k^2.$$

From

$$\begin{aligned} A - \mu_k I &= \begin{bmatrix} \lambda_1 - (\lambda_1 c_k^2 + \lambda_2 s_k^2) & 0 \\ 0 & \lambda_2 - (\lambda_1 c_k^2 + \lambda_2 s_k^2) \end{bmatrix} \\ &= (\lambda_1 - \lambda_2) \begin{bmatrix} s_k^2 & 0 \\ 0 & -c_k^2 \end{bmatrix}, \end{aligned}$$

we obtain

$$\mathbf{z}_k = \frac{1}{\lambda_1 - \lambda_2} \begin{bmatrix} c_k/s_k^2 \\ -s_k/c_k^2 \end{bmatrix} = \frac{1}{c_k^2 s_k^2(\lambda_1 - \lambda_2)} \begin{bmatrix} c_k^3 \\ -s_k^3 \end{bmatrix}.$$

Normalizing yields

$$\mathbf{x}_{k+1} = \frac{1}{\sqrt{c_k^6 + s_k^6}} \begin{bmatrix} c_k^3 \\ -s_k^3 \end{bmatrix},$$

which indicates cubic convergence to a vector that is parallel to $\mathbf{e}_1$ or $\mathbf{e}_2$, provided $|c_k| \neq |s_k|$.

It should be noted that Inverse Iteration is also useful for a general (unsymmetric) matrix A, for finding *selected* eigenvectors after computing the Schur decomposition $A = QTQ^H$, which reveals the eigenvalues of A, but not the eigenvectors. Then, a computed eigenvalue can be used as the shift μ, causing rapid convergence to a corresponding eigenvector. In fact, in practice a single iteration is sufficient. However, when no such information about eigenvalues is available, Inverse Iteration is far more practical for a symmetric matrix than an unsymmetric matrix, due to the superior convergence of the Power Method in the symmetric case.

4.4.4 The Symmetric QR Algorithm

A symmetric Hessenberg matrix is tridiagonal. Therefore, the same kind of Householder reflections that can be used to reduce a general matrix to Hessenberg form can be used to reduce a symmetric matrix A to a tridiagonal matrix T. However, the symmetry of A can be exploited to reduce the number of operations needed to apply each Householder reflection on the left and right of A.

It can be verified by examining the structure of the matrices involved, and the rows and columns influenced by Givens rotations, that if T is a symmetric tridiagonal matrix, and $T = QR$ is its QR factorization, then Q is upper Hessenberg, and R is *upper-bidiagonal* (meaning that it is upper-triangular, with upper bandwidth 1, so that all entries below the main diagonal and above the superdiagonal are zero). Furthermore, $\tilde{T} = RQ$ is also tridiagonal.

Because each Givens rotation only affects $O(1)$ nonzero elements of a tridiagonal matrix T, it follows that it only takes $O(n)$ operations to compute the QR factorization of a tridiagonal matrix, and to multiply the factors in reverse order. However, to compute the eigenvectors of A as well as the eigenvalues, it is necessary to compute the product of all of the Givens rotations, which still takes $O(n^2)$ operations.

The Implicit Q Theorem applies to symmetric matrices as well, meaning that if two orthogonal similarity transformations reduce a matrix A to *unreduced* tridiagonal form, and they have the same first column, then they are essentially equal, as are the tridiagonal matrices that they produce.

In the symmetric case, there is no need for a double-shift strategy, because the eigenvalues are real. However, the Implicit Q Theorem can be used for a different purpose: computing the similarity transformation to be used during each iteration without explicitly computing $T - \mu I$, where T is the tridiagonal matrix that is to be reduced to diagonal form. Instead, the first column of $T - \mu I$ can be computed, and then a Householder transformation to make it a multiple of $\mathbf{e}_1$. This can then be applied directly to T, followed by a series of Givens rotations to restore tridiagonal form. By the Implicit Q Theorem, this accomplishes the same effect as computing the QR factorization $UR = T - \mu I$ and then computing $\tilde{T} = RU + \mu I$.

While the shift $\mu = t_{nn}$ can always be used, it is actually more effective to use the *Wilkinson shift*, which is given by

$$\mu = t_{nn} + d - \text{sign}(d)\sqrt{d^2 + t_{n,n-1}^2}, \quad d = \frac{t_{n-1,n-1} - t_{nn}}{2}.$$

This expression yields the eigenvalue of the lower 2×2 block of T that is closer to t_{nn}. It can be shown that this choice of shift leads to *cubic* convergence of $t_{n,n-1}$ to zero.

The symmetric QR algorithm is much faster than the unsymmetric QR algorithm. A single QR step requires about $30n$ operations, because it operates on a tridiagonal matrix rather than a Hessenberg matrix, with an additional $6n^2$ operations for accumulating orthogonal transformations. The overall symmetric QR algorithm requires $4n^3/3$ operations to compute only the eigenvalues, and approximately $8n^3$ additional operations to accumulate transformations. Because a symmetric matrix is unitarily diagonalizable, then the columns of the orthogonal matrix Q such that Q^TAQ is diagonal contains the eigenvectors of A.

4.5 The SVD Algorithm

Let A be an $m \times n$ matrix. The *Singular Value Decomposition* (SVD) of A,

$$A = U\Sigma V^T,$$

where U is $m \times m$ and orthogonal, V is $n \times n$ and orthogonal, and Σ is an $m \times n$ diagonal matrix with nonnegative diagonal entries

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p, \quad p = \min\{m, n\},$$

known as the *singular values* of A, is an extremely useful decomposition that yields much information about A, including its range, null space, rank, and 2-norm condition number. We now discuss a practical algorithm for computing the SVD of A, due to Golub and Kahan.

Let U and V have column partitions

$$U = \begin{bmatrix} \mathbf{u}_1 & \cdots & \mathbf{u}_m \end{bmatrix}, \quad V = \begin{bmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_n \end{bmatrix}.$$

From the relations

$$A\mathbf{v}_j = \sigma_j \mathbf{u}_j, \quad A^T\mathbf{u}_j = \sigma_j \mathbf{v}_j, \quad j = 1, \ldots, p,$$

it follows that

$$A^TA\mathbf{v}_j = \sigma_j^2 \mathbf{v}_j.$$

That is, the squares of the singular values are the eigenvalues of A^TA, which is a symmetric matrix.

It follows that one approach to computing the SVD of A is to apply the symmetric QR algorithm to A^TA to obtain a decomposition $A^TA = V\Sigma^T\Sigma V^T$. Then, the relations $A\mathbf{v}_j = \sigma_j\mathbf{u}_j$, $j = 1, \ldots, p$, can be used in conjunction with the QR factorization with column pivoting to obtain U. However, this approach is not the most practical, because of the expense and loss of information incurred from computing A^TA.

Instead, we can *implicitly* apply the symmetric QR algorithm to A^TA. As the first step of the symmetric QR algorithm is to use Householder reflections to reduce the matrix to tridiagonal form, we can use Householder reflections to instead reduce A to *upper bidiagonal form*

$$U_1^TAV_1 = B = \begin{bmatrix} d_1 & f_1 & & & \\ & d_2 & f_2 & & \\ & & \ddots & \ddots & \\ & & & d_{n-1} & f_{n-1} \\ & & & & d_n \end{bmatrix}.$$

It follows that $T = B^TB$ is symmetric and tridiagonal.

We could then apply the symmetric QR algorithm directly to T, but, again, to avoid the loss of information from computing T explicitly, we implicitly apply the QR algorithm to T by performing the following steps during each iteration:

1. Determine the first Givens row rotation G_1^T that *would* be applied to $T - \mu I$, where μ is the Wilkinson shift from the symmetric QR algorithm. This requires only computing the first column of T, which has only two nonzero entries $t_{11} = d_1^2$ and $t_{21} = d_1f_1$.

2. Apply G_1 as a *column* rotation to columns 1 and 2 of B to obtain $B_1 = BG_1$. This introduces an unwanted nonzero in the $(2, 1)$ entry.

3. Apply a Givens row rotation H_1 to rows 1 and 2 to zero the $(2, 1)$ entry of B_1, which yields $B_2 = H_1^TBG_1$. Then, B_2 has an unwanted nonzero in the $(1, 3)$ entry.

4. Apply a Givens column rotation G_2 to columns 2 and 3 of B_2, which yields $B_3 = H_1^TBG_1G_2$. This introduces an unwanted zero in the $(3, 2)$ entry.

5. Continue applying alternating row and column rotations to "chase" the unwanted nonzero entry down the diagonal of B, until finally B is restored to upper bidiagonal form.

By the Implicit Q Theorem, since G_1 is the Givens rotation that would be applied to the first column of T, the column rotations that help restore upper bidiagonal form are essentially equal to those that would be applied to T if the symmetric QR algorithm was being applied to T directly. Therefore, the symmetric QR algorithm is being correctly applied, implicitly, to B.

To detect decoupling, we note that if any superdiagonal entry f_i is small enough to be "declared" equal to zero, then decoupling has been achieved, because the ith subdiagonal entry of T is equal to d_if_i, and therefore such a subdiagonal entry must be zero as well. If a diagonal entry d_i becomes zero, then decoupling is also achieved, because row or column rotations can be used to zero an entire row or column of B. In summary, if any diagonal or superdiagonal entry of B becomes zero, then the tridiagonal matrix $T = B^TB$ is no longer unreduced.

Eventually, sufficient decoupling is achieved so that B is reduced to a diagonal matrix Σ. All Householder reflections that have pre-multiplied A, and all row rotations that have been applied to B, can be accumulated to obtain U, and all Householder reflections that have post-multiplied A, and all column rotations that have been applied to B, can be accumulated to obtain V.

4.6 Jacobi Methods

One of the major drawbacks of the symmetric QR algorithm is that it is not parallelizable. Each orthogonal similarity transformation that is needed to reduce the original matrix A to diagonal form is dependent upon the previous one. In view of the evolution of parallel architectures, it is therefore worthwhile to consider whether there are alternative approaches to reducing an $n \times n$ symmetric matrix A to diagonal form that can exploit these architectures.

4.6.1 The Jacobi Idea

To that end, we consider a rotation matrix, denoted by $J(p, q, \theta)$, of the form

$$\begin{array}{cc} \left[\begin{array}{ccccccc} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{array}\right] & \begin{array}{c} \\ \\ p \\ \\ q \\ \\ \\ \end{array} \\ \begin{array}{ccccccc} & & p & & q & & \end{array} & \end{array}$$

where $c = \cos\theta$ and $s = \sin\theta$. This matrix, when applied as a similarity transformation to a symmetric matrix A, rotates rows and columns p and q of A through the angle θ so that the (p, q) and (q, p) entries are zeroed. We call the matrix $J(p, q, \theta)$ a *Jacobi rotation.* It is actually identical to a Givens rotation, but in this context we call it a Jacobi rotation to acknowledge its inventor.

Let off(A) be the square root of the sum of squares of all off-diagonal elements of A. That is,

$$\text{off}(A)^2 = \|A\|_F^2 - \sum_{i=1}^{n} a_{ii}^2.$$

Furthermore, let

$$B = J(p, q, \theta)^T A J(p, q, \theta).$$

Then, because the Frobenius norm is invariant under orthogonal transformations, and because only rows and columns p and q of A are modified in B, we have

$$\begin{aligned} \text{off}(B)^2 &= \|B\|_F^2 - \sum_{i=1}^{n} b_{ii}^2 \\ &= \|A\|_F^2 - \sum_{i\neq p,q} b_{ii}^2 - (b_{pp}^2 + b_{qq}^2) \\ &= \|A\|_F^2 - \sum_{i\neq p,q} a_{ii}^2 - (a_{pp}^2 + 2a_{pq}^2 + a_{qq}^2) \\ &= \|A\|_F^2 - \sum_{i=1}^{n} a_{ii}^2 - 2a_{pq}^2 \\ &= \text{off}(A)^2 - 2a_{pq}^2 \\ &< \text{off}(A)^2. \end{aligned}$$

We see that the "size" of the off-diagonal part of the matrix is guaranteeed to decrease from such a similarity transformation.

4.6.2 The 2-by-2 Symmetric Schur Decomposition

We now determine the values c and s such that the diagonalization

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{pq} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} b_{pp} & 0 \\ 0 & b_{qq} \end{bmatrix}$$

is achieved. From the above matrix equation, we obtain the equation

$$0 = b_pq = a_pq(c^2 - s^2) + (a_pp - a_qq)cs.$$

If we define

$$\tau = \frac{a_{qq} - a_{pp}}{2a_{pq}}, \quad t = \frac{s}{c},$$

then t satisfies the quadratic equation

$$t^2 + 2\tau t - 1 = 0.$$

Choosing the root that is smaller in absolute value,

$$t = -\tau + / - \sqrt{1+\tau^2} = \tan\theta,$$

and using the relationships

$$\frac{s}{c} = t, \quad s^2 + c^2 = 1,$$

we obtain

$$c = \frac{1}{sqrt1 + t^2}, \quad s = ct.$$

We choose the smaller root to ensure that $|\theta| \leq \pi/4$ and that the difference between A and B, since

$$\|B - A\|_F^2 = 4(1-c)\sum_{i\neq p,q}(a_{ip}^2 + a_{iq})^2 + 2a_pq^2/c^2.$$

Therefore, we want c to be larger, which is accomplished if t is smaller.

4.6.3 The Classical Jacobi Algorithm

The *classical Jacobi algorithm* proceeds as follows: find indices p and q, $p \neq q$, such that $|a_{pq}|$ is maximized. Then, we use a single Jacobi rotation to zero a_{pq}, and then repeat this process until $\text{off}(A)$ is sufficiently small.

Let $N = n(n-1)/2$. A sequence of N Jacobi rotations is called a *sweep*. Because, during each iteration, a_{pq} is the largest off-diagonal entry, it follows that

$$\text{off}(A)^2 \leq 2Na_{pq}^2.$$

Because $\text{off}(B)^2 = \text{off}(A)^2 - 2a_{pq}^2$, we have

$$\text{off}(B)^2 \leq \left(1 - \frac{1}{N}\right) \text{off}(A)^2,$$

which implies linear convergence. However, it has been shown that for sufficiently large k, there exist a constant c such that

$$\text{off}(A^{(k+N)}) \leq c * \text{off}(A^{(k)})^2,$$

where $A^{(k)}$ is the matrix after k Jacobi updates, meaning that the clasical Jacobi algorithm converges quadratically as a function of sweeps. Heuristically, it has been argued that approximatly $\log n$ sweeps are needed in practice.

It is worth noting that the guideline that θ be chosen so that $|\theta| \leq \pi/4$ is actually essential to ensure quadratic convergence, because otherwise it is possible that Jacobi updates may simply interchange nearly converged diagonal entries.

4.6.4 The Cyclic-by-Row Algorithm

The classical Jacobi algorithm is impractical because it requires $O(n^2)$ comparisons to find the largest off-diagonal element. A variation, called the *cyclic-by-row* algorithm, avoids this expense by simply cycling through the rows of A, in order. It can be shown that quadratic convergence is still achieved.

4.6.5 Error Analysis

In terms of floating-point operations and comparisons, the Jacobi method is not competitive with the symmetric QR algorithm, as the expense of two Jacobi sweeps is comparable to that of the entire symmetric QR algorithm, even with the accumulation of transformations to obtain the matrix of eigenvectors. On the other hand, the Jacobi method can exploit a known approximate eigenvector matrix, whereas the symmetric QR algorithm cannot.

The relative error in the computed eigenvalues is quite small if A is positive definite. If λ_i is an exact eigenvalue of A and $\tilde{\lambda}_i$ is the closest computed eigenvalue, then it has been shown by Demmel and Veseliĉ that

$$\frac{|\tilde{\lambda}_i - \lambda_i|}{|\lambda_i|} \approx \mathbf{u}\kappa_2(D^{-1}AD^{-1}) \ll \mathbf{u}\kappa_2(A),$$

where D is a diagonal matrix with diagonal entries $\sqrt{a_{11}}, \sqrt{a_{22}}, \ldots, \sqrt{a_{nn}}$.

4.6.6 Parallel Jacobi

The primary advantage of the Jacobi method over the symmetric QR algorithm is its parallelism. As each Jacobi update consists of a row rotation that affects only rows p and q, and a column rotation that effects only columns p and q, up to $n/2$ Jacobi updates can be performed in parallel. Therefore, a sweep can be efficiently implemented by performing $n-1$ series of $n/2$ parallel updates in which each row i is paired with a different row j, for $i \neq j$.

As the size of the matrix, n, is generally much greater than the number of processors, p, it is common to use a *block* approach, in which each update consists of the computation of a $2r \times 2r$ symmetric Schur decomposition for some chosen block size r. This is accomplished by applying another algorithm, such as the symmetric QR algorithm, on a smaller scale. Then, if $p \geq n/(2r)$, an entire block Jacobi sweep can be parallelized.

4.6.7 Jacobi SVD Procedures

The Jacobi method can be adapted to compute the SVD, just as the symmetric QR algorithm is. Two types of Jacobi SVD procedures are:

- *Two-sided Jacobi*: In each Jacobi update, a 2×2 SVD is computed in place of a 2×2 Schur decomposition, using a pair of rotations to zero out the off-diagonal entries a_{pq} and a_{qp}. This process continues until $\text{off}(A)$, whose square is is reduced by $a_{pq}^2 + a_{qp}^2$, is sufficiently small. The idea is to first use a row rotation to make the block symmetric, and then perform a Jacobi update as in the symmetric eigenvalue problem to diagonalize the symmetrized block.

- *One-sided Jacobi:* This approach, like the Golub-Kahan SVD algorithm, implicitly applies the Jacobi method for the symmetric eigenvalue problem to A^TA. The idea is, within each update, to use a column Jacobi rotation to rotate columns p and q of A so that they are orthogonal, which has the effect of zeroing the (p, q) entry of A^TA. Once all columns of AV are orthogonal, where V is the accumulation of all column rotations, the relation $AV = U\Sigma$ is used to obtain U and Σ by simple column scaling. To find a suitable rotation, we note that if $\mathbf{a}_p$ and $\mathbf{a}_q$, the pth and qth columns of A, are rotated through an angle θ, then the rotated columns satisfy

  $$(c\mathbf{a}_p - s\mathbf{a}_q)^T(s\mathbf{a}_p + c\mathbf{a}_q) = cs(\|\mathbf{a}_p\|^2 - \|\mathbf{a}_q\|^2) + 2(c^2 - s^2)\mathbf{y}^T\mathbf{x},$$

  where $c = \cos\theta$ and $s = \sin\theta$. Dividing by c^2 and defining $t = s/c$, we obtain a quadratic equation for t that can be solved to obtain c and s.

Part III

Data Fitting and Function Approximation

Chapter 5

Polynomial Interpolation

Calculus provides many tools that can be used to understand the behavior of functions, but in most cases it is necessary for these functions to be continuous or differentiable. This presents a problem in most "real" applications, in which functions are used to model relationships between quantities, but our only knowledge of these functions consists of a set of discrete data points, where the data is obtained from measurements. Therefore, we need to be able to construct continuous functions based on discrete data.

The problem of constructing such a continuous function is called **data fitting**. In this lecture, we discuss a special case of data fitting known as **interpolation**, in which the goal is to find a linear combination of n known functions to fit a set of data that imposes n constraints, thus guaranteeing a unique solution that fits the data exactly, rather than approximately. The broader term "constraints" is used, rather than simply "data points", since the description of the data may include additional information such as rates of change or requirements that the fitting function have a certain number of continuous derivatives.

When it comes to the study of functions using calculus, polynomials are particularly simple to work with. Therefore, in this course we will focus on the problem of constructing a polynomial that, in some sense, fits given data. We first discuss some algorithms for computing the unique polynomial $p_n(x)$ of degree n that satisfies $p_n(x_i) = y_i$, $i = 0, \ldots, n$, where the points (x_i, y_i) are given. The points $x_0, x_1, \ldots, x_n$ are called **interpolation points**. The polynomial $p_n(x)$ is called the **interpolating polynomial** of the data (x_0, y_0), (x_1, y_1), …, (x_n, y_n). At first, we will assume that the interpolation points are all distinct; this assumption will be relaxed in a later section.

5.1 Existence and Uniqueness

A straightforward method of computing the interpolating polynomial

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

is to solve the equations $p_n(x_i) = y_i$, for $i = 0, 1, \ldots, n$, for the unknown coefficients a_j, $j = 0, 1, \ldots, n$.

Exercise 5.1.1 *Solve the system of equations*

$$\begin{aligned} a_0 + a_1 x_0 &= y_0, \\ a_0 + a_1 x_1 &= y_1 \end{aligned}$$

for the coefficients of the linear function $p_1(x) = a_0 + a_1 x$ *that interpolates the data* (x_0, y_0), (x_1, y_1). *What is the system of equations that must be solved to compute the coefficients* a_0, a_1 *and* a_2 *of the quadratic function* $p_2(x) = a_0 + a_1 x + a_2 x^2$ *that interpolates the data* (x_0, y_0), (x_1, y_1), (x_2, y_2)? *Express both systems of equations in matrix-vector form.*

For general n, computing the coefficients $a_0, a_1, \ldots, a_n$ of $p_n(x)$ requires solving the system of linear equations $V_n\mathbf{a} = \mathbf{y}$, where the entries of V_n are defined by $[V_n]_{ij} = x_i^j$, $i, j = 0, \ldots, n$, where $x_0, x_1, \ldots, x_n$ are the points at which the data $y_0, y_1, \ldots, y_n$ are obtained. The basis $\{1, x, \ldots, x^n\}$ of the space of polynomials of degree n is called the **monomial basis**, and the corresponding matrix V_n is called the **Vandermonde matrix** for the points $x_0, x_1, \ldots, x_n$.

Unfortunately, this approach to computing $p_n(x)$ is not practical. Solving this system of equations requires $O(n^3)$ floating-point operations; we will see that $O(n^2)$ is possible. Furthermore, the Vandermonde matrix can be ill-conditioned, especially when the interpolation points $x_0, x_1, \ldots, x_n$ are close together. Instead, we will construct $p_n(x)$ using a representation other than the monomial basis. That is, we will represent $p_n(x)$ as

$$p_n(x) = \sum_{i=0}^{n} a_i \varphi_i(x),$$

for some choice of polynomials $\varphi_0(x), \varphi_1(x), \ldots, \varphi_n(x)$. This is equivalent to solving the linear system $P\mathbf{a} = \mathbf{y}$, where the matrix P has entries $p_{ij} = \varphi_j(x_i)$. By choosing the basis functions $\{\varphi_i(x)\}_{i=0}^n$ judiciously, we can obtain a simpler system of linear equations to solve.

Exercise 5.1.2 *Write down the Vandermonde matrix* V_n *for the points* $x_0, x_1, \ldots, x_n$. *Show that*

$$\det V_n = \prod_{i=0}^{n} \prod_{j=0}^{i-1} (x_i - x_j).$$

Conclude that the system of equations $V_n\mathbf{a} = \mathbf{y}$ *has a unique solution.*

Exercise 5.1.3 *In this exercise, we consider another approach to proving the uniqueness of the interpolating polynomial. Let* $p_n(x)$ *and* $q_n(x)$ *be polynomials of degree* n *such that* $p_n(x_i) = q_n(x_i) = y_i$ *for* $i = 0, 1, 2, \ldots, n$. *Prove that* $p_n(x) \equiv q_n(x)$ *for all* x.

Exercise 5.1.4 *Suppose we express the interpolating polynomial of degree one in the form*

$$p_1(x) = a_0(x - x_1) + a_1(x - x_0).$$

What is the matrix of the system of equations $p_1(x_i) = y_i$, *for* $i = 0, 1$? *How should the form of the interpolating polynomial of degree two,* $p_2(x)$, *be chosen to obtain an equally simple system of equations to solve for the coefficients* a_0, a_1, *and* a_2?

5.2 Lagrange Interpolation

In **Lagrange interpolation**, the matrix P is simply the identity matrix, by virtue of the fact that the interpolating polynomial is written in the form

$$p_n(x) = \sum_{j=0}^{n} y_j \mathcal{L}_{n,j}(x),$$

where the polynomials $\{\mathcal{L}_{n,j}\}_{j=0}^{n}$ have the property that

$$\mathcal{L}_{n,j}(x_i) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}.$$

The polynomials $\{\mathcal{L}_{n,j}\}$, $j = 0, \ldots, n$, are called the **Lagrange polynomials** for the interpolation points x_0, x_1, …, x_n.

To obtain a formula for the Lagrange polynomials, we note that the above definition specifies the roots of $\mathcal{L}_{n,j}(x)$: x_i, for $i \neq j$. It follows that $\mathcal{L}_{n,j}(x)$ has the form

$$\mathcal{L}_{n,j}(x) = \beta_j \prod_{i=0, i\neq j}^{n} (x - x_i)$$

for some constant β_j. Substituting $x = x_j$ and requiring $\mathcal{L}_{n,j}(x_j) = 1$ yields $\beta_j = \prod_{i=0,i\neq j}^{n} \frac{1}{(x_j - x_i)}$. We conclude that

$$\mathcal{L}_{n,j}(x) = \prod_{k=0,k\neq j}^{n} \frac{x - x_k}{x_j - x_k}.$$

As the following result indicates, the problem of polynomial interpolation can be solved using Lagrange polynomials.

Theorem 5.2.1 *Let $x_0, x_1, \ldots, x_n$ be $n+1$ distinct numbers, and let $f(x)$ be a function defined on a domain containing these numbers. Then the polynomial defined by*

$$p_n(x) = \sum_{j=0}^{n} f(x_j) \mathcal{L}_{n,j}$$

is the unique polynomial of degree n that satisfies

$$p_n(x_j) = f(x_j), \quad j = 0, 1, \ldots, n.$$

Example 5.2.2 *We will use Lagrange interpolation to find the unique polynomial $p_3(x)$, of degree 3 or less, that agrees with the following data:*

i	x_i	y_i
0	-1	3
1	0	-4
2	1	5
3	2	-6

In other words, we must have $p_3(-1) = 3$, $p_3(0) = -4$, $p_3(1) = 5$, *and* $p_3(2) = -6$.

First, we construct the Lagrange polynomials $\{\mathcal{L}_{3,j}(x)\}_{j=0}^{3}$, *using the formula*

$$\mathcal{L}_{n,j}(x) = \prod_{i=0, i\neq j}^{3} \frac{(x - x_i)}{(x_j - x_i)}.$$

This yields

$$\begin{aligned}
\mathcal{L}_{3,0}(x) &= \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} \\
&= \frac{(x-0)(x-1)(x-2)}{(-1-0)(-1-1)(-1-2)} \\
&= \frac{x(x^2-3x+2)}{(-1)(-2)(-3)} \\
&= -\frac{1}{6}(x^3-3x^2+2x) \\
\mathcal{L}_{3,1}(x) &= \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} \\
&= \frac{(x+1)(x-1)(x-2)}{(0+1)(0-1)(0-2)} \\
&= \frac{(x^2-1)(x-2)}{(1)(-1)(-2)} \\
&= \frac{1}{2}(x^3-2x^2-x+2) \\
\mathcal{L}_{3,2}(x) &= \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} \\
&= \frac{(x+1)(x-0)(x-2)}{(1+1)(1-0)(1-2)} \\
&= \frac{x(x^2-x-2)}{(2)(1)(-1)} \\
&= -\frac{1}{2}(x^3-x^2-2x) \\
\mathcal{L}_{3,3}(x) &= \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} \\
&= \frac{(x+1)(x-0)(x-1)}{(2+1)(2-0)(2-1)} \\
&= \frac{x(x^2-1)}{(3)(2)(1)} \\
&= \frac{1}{6}(x^3-x).
\end{aligned}$$

By substituting x_i *for* x *in each Lagrange polynomial* $\mathcal{L}_{3,j}(x)$, *for* $j = 0, 1, 2, 3$, *it can be verified that*

$$\mathcal{L}_{3,j}(x_i) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}.$$

It follows that the Lagrange interpolating polynomial $p_3(x)$ is given by

$$\begin{aligned}
p_3(x) &= \sum_{j=0}^{3} y_j \mathcal{L}_{3,j}(x) \\
&= y_0\mathcal{L}_{3,0}(x) + y_1\mathcal{L}_{3,1}(x) + y_2\mathcal{L}_{3,2}(x) + y_3\mathcal{L}_{3,3}(x) \\
&= (3)\left(-\frac{1}{6}\right)(x^3 - 3x^2 + 2x) + (-4)\frac{1}{2}(x^3 - 2x^2 - x + 2) + (5)\left(-\frac{1}{2}\right)(x^3 - x^2 - 2x) + \\
&\quad (-6)\frac{1}{6}(x^3 - x) \\
&= -\frac{1}{2}(x^3 - 3x^2 + 2x) + (-2)(x^3 - 2x^2 - x + 2) - \frac{5}{2}(x^3 - x^2 - 2x) - (x^3 - x) \\
&= \left(-\frac{1}{2} - 2 - \frac{5}{2} - 1\right)x^3 + \left(\frac{3}{2} + 4 + \frac{5}{2}\right)x^2 + (-1 + 2 + 5 + 1)\,x - 4 \\
&= -6x^3 + 8x^2 + 7x - 4.
\end{aligned}$$

Substituting each x_i, for $i = 0, 1, 2, 3$, into $p_3(x)$, we can verify that we obtain $p_3(x_i) = y_i$ in each case. □

Exercise 5.2.1 *Write a* MATLAB *function* `L=makelagrange(x)` *that accepts as input a vector* `x` *of length $n+1$ consisting of the points $x_0, x_1, \ldots, x_n$, which must be distinct, and returns a $(n+1)\times(n+1)$ matrix* `L`*, each row of which consists of the coefficients of the Lagrange polynomial $\mathcal{L}_{n,j}$, $j = 0, 1, 2, \ldots, n$, with highest-degree coefficients in the first column. Use the* `conv` *function to multiply polynomials that are represented as row vectors of coefficients.*

Exercise 5.2.2 *Write a* MATLAB *function* `p=lagrangefit(x,y)` *that accepts as input vectors* `x` *and* `y` *of length $n+1$ consisting of the x- and y-coordinates, respectively, of points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, where the x-values must all be distinct, and returns a $(n+1)$-vector* `p` *consisting of the coefficients of the Lagrange interpolating polynomial $p_n(x)$, with highest-degree coefficient in the first position. Use your* `makelagrange` *function from Exercise 5.2.1. Test your function by comparing your output to that of the built-in function* `polyfit`.

The Lagrange interpolating polynomial can be inefficient to evaluate, as written, because it involves $O(n^2)$ subtractions for a polynomial of degree n. We can evaluate the interpolating polynomial more efficiently using a technique called **barycentric interpolation** [5].

Starting with the Lagrange form of an interpolating polynomial $p_n(x)$,

$$p_n(x) = \sum_{j=0}^{n} y_j \mathcal{L}_{n,j}(x),$$

we define the **barycentric weights** w_j, $j = 0, 1, \ldots, n$, by

$$w_j = \prod_{i=0, i\neq j}^{n} \frac{1}{x_j - x_i}. \tag{5.1}$$

Next, we define

$$\pi_n(x) = (x - x_0)(x - x_1) \cdots (x - x_n).$$

Then, each Lagrange polynomial can be rewritten as

$$\mathcal{L}_{n,j}(x) = \frac{\pi_n(x) w_j}{x - x_j}, \quad x \neq x_j,$$

and the interpolant itself can be written as

$$p_n(x) = \pi_n(x) \sum_{j=0}^{n} \frac{y_j w_j}{x - x_j}.$$

However, if we let $y_j = 1$ for $j = 0, 1, \ldots, n$, we have

$$1 = \pi_n(x) = \sum_{j=0}^{n} \frac{w_j}{x - x_j}.$$

Dividing the previous two equations yields

$$p_n(x) = \frac{\displaystyle\sum_{j=0}^{n} \frac{y_j w_j}{x - x_j}}{\displaystyle\sum_{j=0}^{n} \frac{w_j}{x - x_j}}.$$

Although $O(n^2)$ products are needed to compute the barycentric weights, they need only be computed once, and then re-used for each x, which is not the case with the Lagrange form.

Exercise 5.2.3 *Write a* Matlab *function* `w=baryweights(x)` *that accepts as input a vector* `x` *of length* $n + 1$*, consisting of the distinct interpolation points* $x_0, x_1, \ldots, x_n$*, and returns a vector* `w` *of length* $n + 1$ *consisting of the barycentric weights* w_j *as defined in (5.1).*

Exercise 5.2.4 *Write a* Matlab *function* `yy=lagrangeval(x,y,xx)` *that accepts as input vectors* `x` *and* `y`*, both of length* $n + 1$*, representing the data points* $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$*, and a vector of* x*-values* `xx`*. This function uses the barycentric weights (5.1), as computed by the function* `baryweights` *from Exercise 5.2.3, to compute the value of the Lagrange interpolating polynomial for the given data points at each* x*-value in* `xx`*. The corresponding* y*-values must be returned in the vector* `yy`*, which must have the same dimensions as* `xx`*.*

5.3 Divided Differences

While the Lagrange polynomials are easy to compute, they are difficult to work with. Furthermore, if new interpolation points are added, all of the Lagrange polynomials must be recomputed. Unfortunately, it is not uncommon, in practice, to add to an existing set of interpolation points. It may be determined after computing the kth-degree interpolating polynomial $p_k(x)$ of a function $f(x)$ that $p_k(x)$ is not a sufficiently accurate approximation of $f(x)$ on some domain. Therefore, an interpolating polynomial of higher degree must be computed, which requires additional interpolation points.

To address these issues, we consider the problem of computing the interpolating polynomial *recursively.* More precisely, let $k > 0$, and let $p_k(x)$ be the polynomial of degree k that interpolates the function $f(x)$ at the points $x_0, x_1, \ldots, x_k$. Ideally, we would like to be able to obtain $p_k(x)$ from polynomials of degree $k-1$ that interpolate $f(x)$ at points chosen from among $x_0, x_1, \ldots, x_k$. The following result shows that this is possible.

Theorem 5.3.1 *Let n be a positive integer, and let $f(x)$ be a function defined on a domain containing the $n+1$ distinct points $x_0, x_1, \ldots, x_n$, and let $p_n(x)$ be the polynomial of degree n that interpolates $f(x)$ at the points $x_0, x_1, \ldots, x_n$. For each $i = 0, 1, \ldots, n$, we define $p_{n-1,i}(x)$ to be the polynomial of degree $n-1$ that interpolates $f(x)$ at the points $x_0, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$. If i and j are distinct nonnegative integers not exceeding n, then*

$$p_n(x) = \frac{(x - x_j)p_{n-1,j}(x) - (x - x_i)p_{n-1,i}(x)}{x_i - x_j}.$$

This theorem can be proved by substituting $x = x_i$ into the above form for $p_n(x)$, and using the fact that the interpolating polynomial is unique.

Exercise 5.3.1 *Prove Theorem 5.3.1.*

This result leads to an algorithm called **Neville's Method** [26] that computes the value of $p_n(x)$ at a given point using the values of lower-degree interpolating polynomials at x. We now describe this algorithm in detail.

Algorithm 5.3.2 *Let $x_0, x_1, \ldots, x_n$ be distinct numbers, and let $f(x)$ be a function defined on a domain containing these numbers. Given a number x^*, the following algorithm computes $y^* = p_n(x^*)$, where $p_n(x)$ is the nth interpolating polynomial of $f(x)$ that interpolates $f(x)$ at the points $x_0, x_1, \ldots, x_n$.*

```
for $j = 0$ to $n$ do
    $Q_j = f(x_j)$
end
for $j = 1$ to $n$ do
    for $k = n$ to $j$ do
        $Q_k = [(x - x_k)Q_{k-1} - (x - x_{k-j})Q_k]/(x_{k-j} - x_k)$
    end
end
$y^* = Q_n$
```

At the jth iteration of the outer loop, the number Q_k, for $k = n, n-1, \ldots, j$, represents the value at x of the polynomial that interpolates $f(x)$ at the points $x_k, x_{k-1}, \ldots, x_{k-j}$.

The preceding theorem can be used to compute the polynomial $p_n(x)$ itself, rather than its value at a given point. This yields an alternative method of constructing the interpolating polynomial, called **Newton interpolation**, that is more suitable for tasks such as inclusion of additional interpolation points. The basic idea is to represent interpolating polynomials using the **Newton form**, which uses linear factors involving the interpolation points, instead of monomials of the form x^j.

5.3.1 Newton Form

Recall that if the interpolation points $x_0, \ldots, x_n$ are distinct, then the process of finding a polynomial that passes through the points (x_i, y_i), $i = 0, \ldots, n$, is equivalent to solving a system of linear equations $A\mathbf{x} = \mathbf{b}$ that has a unique solution. The matrix A is determined by the choice of basis for the space of polynomials of degree n or less. Each entry a_{ij} of A is the value of the jth polynomial in the basis at the point x_i.

In Newton interpolation, the matrix A is upper triangular, and the basis functions are defined to be the set $\{\mathcal{N}_j(x)\}_{j=0}^n$, where

$$\mathcal{N}_0(x) = 1, \quad \mathcal{N}_j(x) = \prod_{k=0}^{j-1} (x - x_k), \quad j = 1, \ldots, n.$$

The advantage of Newton interpolation is that the interpolating polynomial is easily updated as interpolation points are added, since the basis functions $\{\mathcal{N}_j(x)\}$, $j = 0, \ldots, n$, do not change from the addition of the new points.

Using Theorem 5.3.1, it can be shown that the coefficients c_j of the Newton interpolating polynomial

$$p_n(x) = \sum_{j=0}^n c_j \mathcal{N}_j(x)$$

are given by

$$c_j = f[x_0, \ldots, x_j]$$

where $f[x_0, \ldots, x_j]$ denotes the **divided difference** of $x_0, \ldots, x_j$. The divided difference is defined as follows:

$$\begin{aligned} f[x_i] &= y_i, \\ f[x_i, x_{i+1}] &= \frac{y_{i+1} - y_i}{x_{i+1} - x_i}, \\ f[x_i, x_{i+1}, \ldots, x_{i+k}] &= \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}. \end{aligned}$$

This definition implies that for each nonnegative integer j, the divided difference $f[x_0, x_1, \ldots, x_j]$ only depends on the interpolation points $x_0, x_1, \ldots, x_j$ and the value of $f(x)$ at these points. It follows that the addition of new interpolation points does not change the coefficients $c_0, \ldots, c_n$. Specifically, we have

$$p_{n+1}(x) = p_n(x) + \frac{y_{n+1} - p_n(x_{n+1})}{\mathcal{N}_{n+1}(x_{n+1})} \mathcal{N}_{n+1}(x).$$

This ease of updating makes Newton interpolation the most commonly used method of obtaining the interpolating polynomial.

The following result shows how the Newton interpolating polynomial bears a resemblance to a Taylor polynomial.

Theorem 5.3.3 *Let f be n times continuously differentiable on $[a, b]$, and let $x_0, x_1, \dots, x_n$ be distinct points in $[a, b]$. Then there exists a number $\xi \in [a, b]$ such that*

$$f[x_0, x_1, \dots, x_n] = \frac{f^{(n)}(\xi)}{n!}.$$

Exercise 5.3.2 *Prove Theorem 5.3.3 for the case of $n = 2$, using the definition of divided differences and Taylor's theorem.*

Exercise 5.3.3 *Let $p_n(x)$ be the interpolating polynomial for $f(x)$ at points $x_0, x_1, \dots, x_n \in [a, b]$, and assume that f is n times differentiable. Use Rolle's Theorem to prove that $p_n^{(n)}(\xi) = f^{(n)}(\xi)$ for some point $\xi \in [a, b]$.*

Exercise 5.3.4 *Use Exercise 5.3.3 to prove Theorem 5.3.3.* Hint: *Think of $p_{n-1}(x)$ as an interpolant of $p_n(x)$.*

5.3.2 Computing the Newton Interpolating Polynomial

We now describe in detail how to compute the coefficients $c_j = f[x_0, x_1, \dots, x_j]$ of the Newton interpolating polynomial $p_n(x)$, and how to evaluate $p_n(x)$ efficiently using these coefficients.

The computation of the coefficients proceeds by filling in the entries of a **divided-difference table**. This is a triangular table consisting of $n + 1$ columns, where n is the degree of the interpolating polynomial to be computed. For $j = 0, 1, \dots, n$, the jth column contains $n - j$ entries, which are the divided differences $f[x_k, x_{k+1}, \dots, x_{k+j}]$, for $k = 0, 1, \dots, n - j$.

We construct this table by filling in the $n + 1$ entries in column 0, which are the trivial divided differences $f[x_j] = f(x_j)$, for $j = 0, 1, \dots, n$. Then, we use the recursive definition of the divided differences to fill in the entries of subsequent columns. Once the construction of the table is complete, we can obtain the coefficients of the Newton interpolating polynomial from the first entry in each column, which is $f[x_0, x_1, \dots, x_j]$, for $j = 0, 1, \dots, n$.

In a practical implementation of this algorithm, we do not need to store the entire table, because we only need the first entry in each column. Because each column has one fewer entry than the previous column, we can overwrite all of the other entries that we do not need. The following algorithm implements this idea.

Algorithm 5.3.4 (Divided-Difference Table) *Given n distinct interpolation points $x_0, x_1, \ldots, x_n$, and the values of a function $f(x)$ at these points, the following algorithm computes the coefficients $c_j = f[x_0, x_1, \ldots, x_j]$ of the Newton interpolating polynomial.*

```
for $i = 0, 1, \ldots, n$ do
    $d_{i,0} = f(x_i)$
end
for $j = 1, 2, \ldots, n$ do
    for $i = n, n-1, \ldots, j$ do
        $d_{i,j} = (d_{i,j-1} - d_{i-1,j-1})/(x_i - x_{i-j})$
    end
end
for $j = 0, 1, \ldots, n$ do
    $c_j = d_{j,j}$
end
```

Example 5.3.5 *We will use Newton interpolation to construct the third-degree polynomial $p_3(x)$ that fits the data*

i	x_i	$f(x_i)$
0	-1	3
1	0	-4
2	1	5
3	2	-6

In other words, we must have $p_3(-1) = 3$, $p_3(0) = -4$, $p_3(1) = 5$, and $p_3(2) = -6$.

First, we construct the divided-difference table from this data. The divided differences in the table are computed as follows:

$$f[x_0] = f(x_0) = 3, \quad f[x_1] = f(x_1) = -4, \quad f[x_2] = f(x_2) = 5, \quad f[x_3] = f(x_3) = -6,$$

$$\begin{aligned}
f[x_0,x_1] &= \frac{f[x_1]-f[x_0]}{x_1-x_0} = \frac{-4-3}{0-(-1)} = -7 \\
f[x_1,x_2] &= \frac{f[x_2]-f[x_1]}{x_2-x_1} = \frac{5-(-4)}{1-0} = 9 \\
f[x_2,x_3] &= \frac{f[x_3]-f[x_2]}{x_3-x_2} = \frac{-6-5}{2-1} = -11 \\
f[x_0,x_1,x_2] &= \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0} \\
&= \frac{9-(-7)}{1-(-1)} \\
&= 8 \\
f[x_1,x_2,x_3] &= \frac{f[x_2,x_3]-f[x_1,x_2]}{x_3-x_1} \\
&= \frac{-11-9}{2-0} \\
&= -10 \\
f[x_0,x_1,x_2,x_3] &= \frac{f[x_1,x_2,x_3]-f[x_0,x_1,x_2]}{x_3-x_0} \\
&= \frac{-10-8}{2-(-1)} \\
&= -6
\end{aligned}$$

The resulting divided-difference table is

$x_0=-1$	$f[x_0]=3$			
		$f[x_0,x_1]=-7$		
$x_1=0$	$f[x_1]=-4$		$f[x_0,x_1,x_2]=8$	
		$f[x_1,x_2]=9$		$f[x_0,x_1,x_2,x_3]=-6$
$x_2=1$	$f[x_2]=5$		$f[x_1,x_2,x_3]=-10$	
		$f[x_2,x_3]=-11$		
$x_3=2$	$f[x_3]=-6$			

It follows that the interpolating polynomial $p_3(x)$ *can be expressed in Newton form as follows:*

$$\begin{aligned}
p_3(x) &= \sum_{j=0}^{3} f[x_0,\ldots,x_j]\prod_{i=0}^{j-1}(x-x_i) \\
&= f[x_0]+f[x_0,x_1](x-x_0)+f[x_0,x_1,x_2](x-x_0)(x-x_1)+ \\
&\quad f[x_0,x_1,x_2,x_3](x-x_0)(x-x_1)(x-x_2) \\
&= 3-7(x+1)+8(x+1)x-6(x+1)x(x-1).
\end{aligned}$$

We see that Newton interpolation produces an interpolating polynomial that is in the Newton form, with centers $x_0=-1$, $x_1=0$, *and* $x_2=1$. □

Exercise 5.3.5 *Write a* MATLAB *function* `c=divdiffs(x,y)` *that computes the divided difference table from the given data stored in the input vectors* `x` *and* `y`*, and returns a vector* `c` *consisting of the divided differences* $f[x_0, \ldots, x_j]$, $j = 0, 1, 2, \ldots, n$, *where* $n+1$ *is the length of both* `x` *and* `y`*.*

Once the coefficients have been computed, we can use **nested multiplication** to evaluate the resulting interpolating polynomial, which is represented using the Newton form

$$\begin{aligned} p_n(x) &= \sum_{j=0}^{n} c_j \mathcal{N}_j(x) && (5.2) \\ &= \sum_{j=0}^{n} f[x_0, x_1, \ldots, x_j] \prod_{i=0}^{j-1} (x - x_i) && (5.3) \\ &= f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + && (5.4) \\ & \quad f[x_0, x_1, \ldots, x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1}). && (5.5) \end{aligned}$$

Algorithm 5.3.6 (Nested Multiplication) *Given* n *distinct interpolation points* $x_0, x_1, \ldots, x_n$ *and the coefficients* $c_j = f[x_0, x_1, \ldots, x_j]$ *of the Newton interpolating polynomial* $p_n(x)$*, the following algorithm computes* $y = p_n(x)$ *for a given real number* x*.*

```
$b_n = c_n$
for $i = n-1, n-2, \ldots, 0$ do
    $b_i = c_i + (x - x_i)y$
end
$y = b_0$
```

It can be seen that this algorithm closely resembles **Horner's Method**, which is a special case of nested multiplication that works with the power form of a polynomial, whereas nested multiplication works with the more general Newton form.

Example 5.3.7 *Consider the interpolating polynomial obtained in the previous example,*

$$p_3(x) = 3 - 7(x+1) + 8(x+1)x - 6(x+1)x(x-1).$$

We will use nested multiplication to write this polynomial in the **power form**

$$p_3(x) = b_3x^3 + b_2x^2 + b_1x + b_0.$$

This requires repeatedly applying nested multiplication to a polynomial of the form

$$p(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + c_3(x - x_0)(x - x_1)(x - x_2),$$

and for each application it will perform the following steps,

$$\begin{aligned} b_3 &= c_3 \\ b_2 &= c_2 + (z - x_2)b_3 \\ b_1 &= c_1 + (z - x_1)b_2 \\ b_0 &= c_0 + (z - x_0)b_1, \end{aligned}$$

where, in this example, we will set $z = 0$ *each time.*

The numbers b_0, b_1, b_2 *and* b_3 *computed by the algorithm are the coefficients of* $p(x)$ *in the Newton form, with the centers* x_0, x_1 *and* x_2 *changed to* z, x_0 *and* x_1*; that is,*

$$p(x) = b_0 + b_1(x - z) + b_2(x - z)(x - x_0) + b_3(x - z)(x - x_0)(x - x_1).$$

It follows that $b_0 = p(z)$*, which is why this algorithm is the preferred method for evaluating a polynomial in Newton form at a given point* z*.*

It should be noted that the algorithm can be derived by writing $p(x)$ *in the* **nested form**

$$p(x) = c_0 + (x - x_0)[c_1 + (x - x_1)[c_2 + (x - x_2)c_3]]$$

and computing $p(z)$ *as follows:*

$$\begin{aligned} p(z) &= c_0 + (z - x_0)[c_1 + (z - x_1)[c_2 + (z - x_2)c_3]] \\ &= c_0 + (z - x_0)[c_1 + (z - x_1)[c_2 + (z - x_2)b_3]] \\ &= c_0 + (z - x_0)[c_1 + (z - x_1)b_2] \\ &= c_0 + (z - x_0)b_1 \\ &= b_0. \end{aligned}$$

Initially, we have

$$p(x) = 3 - 7(x + 1) + 8(x + 1)x - 6(x + 1)x(x - 1),$$

so the coefficients of $p(x)$ *in this Newton form are*

$$c_0 = 3, \quad c_1 = -7, \quad c_2 = 8, \quad c_3 = -6,$$

with the centers

$$x_0 = -1, \quad x_1 = 0, \quad x_2 = 1.$$

Applying nested multiplication to these coefficients and centers, with $z = 0$*, yields*

$$\begin{aligned} b_3 &= -6 \\ b_2 &= 8 + (0 - 1)(-6) \\ &= 14 \\ b_1 &= -7 + (0 - 0)(14) \\ &= -7 \\ b_0 &= 3 + (0 - (-1))(-7) \\ &= -4. \end{aligned}$$

It follows that

$$\begin{aligned} p(x) &= -4 + (-7)(x - 0) + 14(x - 0)(x - (-1)) + (-6)(x - 0)(x - (-1))(x - 0) \\ &= -4 - 7x + 14x(x + 1) - 6x^2(x + 1), \end{aligned}$$

and the centers are now 0, -1 *and* 0*.*

For the second application of nested multiplication, we have

$$p(x) = -4 - 7x + 14x(x+1) - 6x^2(x+1),$$

so the coefficients of $p(x)$ *in this Newton form are*

$$c_0 = -4, \quad c_1 = -7, \quad c_2 = 14, \quad c_3 = -6,$$

with the centers

$$x_0 = 0, \quad x_1 = -1, \quad x_2 = 0.$$

Applying nested multiplication to these coefficients and centers, with $z = 0$*, yields*

$$\begin{aligned} b_3 &= -6 \\ b_2 &= 14 + (0-0)(-6) \\ &= 14 \\ b_1 &= -7 + (0-(-1))(14) \\ &= 7 \\ b_0 &= -4 + (0-0)(7) \\ &= -4. \end{aligned}$$

It follows that

$$\begin{aligned} p(x) &= -4 + 7(x-0) + 14(x-0)(x-0) + (-6)(x-0)(x-0)(x-(-1)) \\ &= -4 + 7x + 14x^2 - 6x^2(x+1), \end{aligned}$$

and the centers are now 0*,* 0 *and* -1*.*

For the third and final application of nested multiplication, we have

$$p(x) = -4 + 7x + 14x^2 - 6x^2(x+1),$$

so the coefficients of $p(x)$ *in this Newton form are*

$$c_0 = -4, \quad c_1 = 7, \quad c_2 = 14, \quad c_3 = -6,$$

with the centers

$$x_0 = 0, \quad x_1 = 0, \quad x_2 = -1.$$

Applying nested multiplication to these coefficients and centers, with $z = 0$*, yields*

$$\begin{aligned} b_3 &= -6 \\ b_2 &= 14 + (0-(-1))(-6) \\ &= 8 \\ b_1 &= 7 + (0-0)(8) \\ &= 7 \\ b_0 &= -4 + (0-0)(7) \\ &= 1. \end{aligned}$$

It follows that

$$\begin{aligned} p(x) &= -4 + 7(x-0) + 8(x-0)(x-0) + (-6)(x-0)(x-0)(x-0) \\ &= -4 + 7x + 8x^2 - 6x^3, \end{aligned}$$

and the centers are now 0, 0 *and* 0. *Since all of the centers are equal to zero, the polynomial is now in the power form.* □

Exercise 5.3.6 *Write a* MATLAB *function* `yy=newtonval(x,y,xx)` *that accepts as input vectors* `x` *and* `y`, *both of length* $n+1$, *representing the data points* $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, *and a vector of* x*-values* `xx`. *This function uses Algorithm 5.3.6, as well as the function* `divdiffs` *from Exercise 5.3.5, to compute the value of the Newton interpolating polynomial for the given data points at each* x*-value in* `xx`. *The corresponding* y*-values must be returned in the vector* `yy`, *which must have the same dimensions as* `xx`.

It should be noted that this is not the most efficient way to convert the Newton form of $p(x)$ to the power form. To see this, we observe that after one application of nested multiplication, we have

$$\begin{aligned} p(x) &= b_0 + b_1(x-z) + b_2(x-z)(x-x_0) + b_3(x-z)(x-x_0)(x-x_1) \\ &= b_0 + (x-z)[b_1 + b_2(x-x_0) + b_3(x-x_0)(x-x_1)]. \end{aligned}$$

Therefore, we can apply nested multiplication to the second-degree polynomial

$$q(x) = b_1 + b_2(x-x_0) + b_3(x-x_0)(x-x_1),$$

which is the quotient obtained by dividing $p(x)$ by $(x-z)$. Because

$$p(x) = b_0 + (x-z)q(x),$$

it follows that once we have changed all of the centers of $q(x)$ to be equal to z, then all of the centers of $p(x)$ will be equal to z as well. In summary, we can convert a polynomial of degree n from Newton form to power form by applying nested multiplication n times, where the jth application is to a polynomial of degree $n-j+1$, for $j = 1, 2, \ldots, n$.

Since the coefficients of the appropriate Newton form of each of these polynomials of successively lower degree are computed by the nested multiplication algorithm, it follows that we can implement this more efficient procedure simply by proceeding exactly as before, except that during the jth application of nested multiplication, we do not compute the coefficients $b_0, b_1, \ldots, b_{j-2}$, because they will not change anyway, as can be seen from the previous computations. For example, in the second application, we did not need to compute b_0, and in the third, we did not need to compute b_0 and b_1.

Exercise 5.3.7 *Write a* MATLAB *function* `p=powerform(x,c)` *that accepts as input vectors* `x` *and* `c`, *both of length* $n+1$, *consisting of the interpolation points* x_j *and divided differences* $f[x_0, x_1, \ldots, x_j]$, *respectively,* $j = 0, 1, \ldots, n$. *The output is a* $(n+1)$*-vector consisting of the coefficients of the interpolating polynomial* $p_n(x)$ *in power form, ordered from highest degree to lowest.*

Exercise 5.3.8 *Write a* MATLAB *function* `p=newtonfit(x,y)` *that accepts as input vectors* `x` *and* `y` *of length* $n+1$ *consisting of the* x*- and* y*-coordinates, respectively, of points* $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$*, where the* x*-values must all be distinct, and returns a* $(n+1)$*-vector* `p` *consisting of the coefficients of the Newton interpolating polynomial* $p_n(x)$*, in power form, with highest-degree coefficient in the first position. Use your* `divdiffs` *function from Exercise 5.3.5 and your* `powerform` *function from Exercise 5.3.7. Test your function by comparing your output to that of the built-in function* `polyfit`*.*

5.3.3 Equally Spaced Points

Suppose that the interpolation points $x_0, x_1, \ldots, x_n$ are equally spaced; that is, $x_i = x_0 + ih$ for some positive number h. In this case, the Newton interpolating polynomial can be simplified, since the denominators of all of the divided differences can be expressed in terms of the spacing h. If we define the **forward difference operator** Δ by

$$\Delta x_k = x_{k+1} - x_k,$$

where $\{x_k\}$ is any sequence, then the divided differences $f[x_0, x_1, \ldots, x_k]$ are given by

$$f[x_0, x_1, \ldots, x_k] = \frac{1}{k!h^k} \Delta^k f(x_0). \tag{5.6}$$

The interpolating polynomial can then be described by the **Newton forward-difference formula**

$$p_n(x) = f[x_0] + \sum_{k=1}^{n} \left(\begin{array}{c} s \\ k \end{array} \right) \Delta^k f(x_0), \tag{5.7}$$

where the new variable s is related to x by

$$s = \frac{x - x_0}{h},$$

and the **extended binomial coefficient** $\left(\begin{array}{c} s \\ k \end{array} \right)$ is defined by

$$\left(\begin{array}{c} s \\ k \end{array} \right) = \frac{s(s-1)(s-2)\cdots(s-k+1)}{k!},$$

where k is a nonnegative integer.

Exercise 5.3.9 *Use induction to prove (5.6). Then show that the Newton interpolating polynomial (5.5) reduces to (5.7) in the case of equally spaced interpolation points.*

Example 5.3.8 *We will use the Newton forward-difference formula*

$$p_n(x) = f[x_0] + \sum_{k=1}^{n} \left(\begin{array}{c} s \\ k \end{array} \right) \Delta^k f(x_0)$$

to compute the interpolating polynomial $p_3(x)$ *that fits the data*

i	x_i	$f(x_i)$
0	-1	3
1	0	-4
2	1	5
3	2	-6

In other words, we must have $p_3(-1) = 3$, $p_3(0) = -4$, $p_3(1) = 5$, *and* $p_3(2) = -6$. *Note that the interpolation points* $x_0 = -1$, $x_1 = 0$, $x_2 = 1$ *and* $x_3 = 2$ *are equally spaced, with spacing* $h = 1$.

To apply the forward-difference formula, we define $s = (x - x_0)/h = x + 1$ *and compute the extended binomial coefficients*

$$\begin{pmatrix} s \\ 1 \end{pmatrix} = s = x{+}1, \quad \begin{pmatrix} s \\ 2 \end{pmatrix} = \frac{s(s-1)}{2} = \frac{x(x+1)}{2}, \quad \begin{pmatrix} s \\ 3 \end{pmatrix} = \frac{s(s-1)(s-2)}{6} = \frac{(x+1)x(x-1)}{6},$$

and then the coefficients

$$\begin{aligned}
f[x_0] &= f(x_0) \\
&= 3, \\
\Delta f(x_0) &= f(x_1) - f(x_0) \\
&= -4 - 3 \\
&= -7, \\
\Delta^2 f(x_0) &= \Delta(\Delta f(x_0)) \\
&= \Delta[f(x_1) - f(x_0)] \\
&= [f(x_2) - f(x_1)] - [f(x_1) - f(x_0)] \\
&= f(x_2) - 2f(x_1) + f(x_0) \\
&= 5 - 2(-4) + 3, \\
&= 16 \\
\Delta^3 f(x_0) &= \Delta(\Delta^2 f(x_0)) \\
&= \Delta[f(x_2) - 2f(x_1) + f(x_0)] \\
&= [f(x_3) - f(x_2)] - 2[f(x_2) - f(x_1)] + [f(x_1) - f(x_0)] \\
&= f(x_3) - 3f(x_2) + 3f(x_1) - f(x_0) \\
&= -6 - 3(5) + 3(-4) - 3 \\
&= -36.
\end{aligned}$$

It follows that

$$\begin{aligned}
p_3(x) &= f[x_0] + \sum_{k=1}^{3} \begin{pmatrix} s \\ k \end{pmatrix} \Delta^k f(x_0) \\
&= 3 + \begin{pmatrix} s \\ 1 \end{pmatrix} \Delta f(x_0) + \begin{pmatrix} s \\ 1 \end{pmatrix} \Delta^2 f(x_0) + \begin{pmatrix} s \\ 2 \end{pmatrix} \Delta^3 f(x_0) \\
&= 3 + (x+1)(-7) + \frac{x(x+1)}{2} 16 + \frac{(x+1)x(x-1)}{6}(-36) \\
&= 3 - 7(x+1) + 8(x+1)x - 6(x+1)x(x-1).
\end{aligned}$$

Note that the forward-difference formula computes the same form of the interpolating polynomial as the Newton divided-difference formula. □

Exercise 5.3.10 *Define the* **backward difference operator** ∇ *by*

$$\nabla x_k = x_k - x_{k-1},$$

for any sequence $\{x_k\}$*. Then derive the* **Newton backward-difference formula**

$$p_n(x) = f[x_n] + \sum_{k=1}^{n}(-1)^k \left(\begin{array}{c} -s \\ k \end{array} \right) \nabla^k f(x_n),$$

where $s = (x - x_n)/h$*, and the preceding definition of the extended binomial coefficient applies.*

Exercise 5.3.11 *Look up the documentation for the* Matlab *function* `diff`*. Then write functions* `yy=newtonforwdiff(x,y,xx)` *and* `yy=newtonbackdiff(x,y,xx)` *that use* `diff` *to implement the Newton forward-difference and Newton backward-difference formulas, respectively, and evaluate the interpolating polynomial* $p_n(x)$*, where* $n = \texttt{length}(\texttt{x}) - 1$*, at the elements of* `xx`*. The resulting values must be returned in* `yy`*.*

5.4 Error Analysis

In some applications, the interpolating polynomial $p_n(x)$ is used to fit a known function $f(x)$ at the points $x_0, \ldots, x_n$, usually because $f(x)$ is not feasible for tasks such as differentiation or integration that are easy for polynomials, or because it is not easy to evaluate $f(x)$ at points other than the interpolation points. In such an application, it is possible to determine how well $p_n(x)$ approximates $f(x)$.

5.4.1 Error Estimation

From Theorem 5.3.3, we can obtain the following result.

Theorem 5.4.1 (Interpolation error) *If* f *is* $n+1$ *times continuously differentiable on* $[a,b]$*, and* $p_n(x)$ *is the unique polynomial of degree* n *that interpolates* $f(x)$ *at the* $n+1$ *distinct points* $x_0, x_1, \ldots, x_n$ *in* $[a,b]$*, then for each* $x \in [a,b]$*,*

$$f(x) - p_n(x) = \prod_{j=0}^{n}(x - x_j)\frac{f^{(n+1)}(\xi(x))}{(n+1)!},$$

where $\xi(x) \in [a,b]$*.*

It is interesting to note that the error closely resembles the Taylor remainder $R_n(x)$.

Exercise 5.4.1 *Prove Theorem 5.4.1.* Hint: *work with the Newton interpolating polynomial for the points* $x_0, x_1, \ldots, x_n, x$*.*

Exercise 5.4.2 *Determine a bound on the error* $|f(x) - p_2(x)|$ *for* x *in* $[0,1]$*, where* $f(x) = e^x$*, and* $p_2(x)$ *is the interpolating polynomial of* $f(x)$ *at* $x_0 = 0$*,* $x_1 = 0.5$*, and* $x_2 = 1$*.*

If the number of data points is large, then polynomial interpolation becomes problematic since high-degree interpolation yields oscillatory polynomials, when the data may fit a smooth function.

Example 5.4.2 *Suppose that we wish to approximate the function* $f(x) = 1/(1+x^2)$ *on the interval* $[-5,5]$ *with a tenth-degree interpolating polynomial that agrees with* $f(x)$ *at 11 equally-spaced points* $x_0, x_1, \ldots, x_{10}$ *in* $[-5,5]$*, where* $x_j = -5+j$*, for* $j = 0,1,\ldots,10$*. Figure 5.1 shows that the resulting polynomial is not a good approximation of* $f(x)$ *on this interval, even though it agrees with* $f(x)$ *at the interpolation points. The following MATLAB session shows how the plot in the figure can be created.*

```
>> % create vector of 11 equally spaced points in [-5,5]
>> x=linspace(-5,5,11);
>> % compute corresponding y-values
>> y=1./(1+x.^2);
>> % compute 10th-degree interpolating polynomial
>> p=polyfit(x,y,10);
>> % for plotting, create vector of 100 equally spaced points
>> xx=linspace(-5,5);
>> % compute corresponding y-values to plot function
>> yy=1./(1+xx.^2);
>> % plot function
>> plot(xx,yy)
>> % tell MATLAB that next plot should be superimposed on
>> % current one
>> hold on
>> % plot polynomial, using polyval to compute values
>> % and a red dashed curve
>> plot(xx,polyval(p,xx),'r--')
>> % indicate interpolation points on plot using circles
>> plot(x,y,'o')
>> % label axes
>> xlabel('x')
>> ylabel('y')
>> % set caption
>> title('Runge''s example')
```

The example shown in Figure 5.1 is a well-known example of the difficulty of high-degree polynomial interpolation using equally-spaced points, and it is known as **Runge's example** [33]. □

5.4.2 Chebyshev Interpolation

In general, it is not wise to use a high-degree interpolating polynomial and equally-spaced interpolation points to approximate a function on an interval $[a,b]$ unless this interval is sufficiently small.

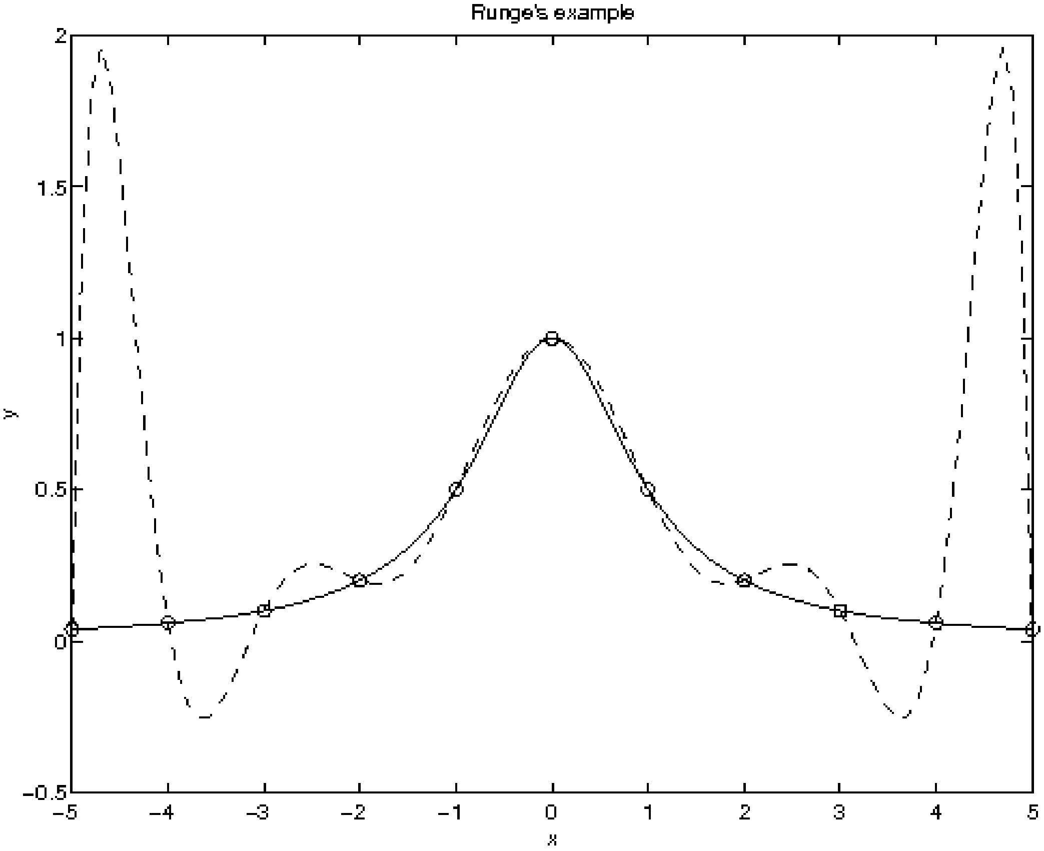


Figure 5.1: The function $f(x) = 1/(1+x^2)$ (solid curve) cannot be interpolated accurately on $[-5, 5]$ using a tenth-degree polynomial (dashed curve) with equally-spaced interpolation points.

Is it possible to choose the interpolation points so that the error is minimized? To answer this question, we introduce the **Chebyshev polynomials**

$$T_k(x) = \cos(k\cos^{-1}(x)), \quad |x| \leq 1, \quad k = 0, 1, 2, \ldots. \tag{5.8}$$

Using (5.8) and the sum and difference formulas for cosine,

$$\begin{aligned} \cos(A+B) &= \cos A\cos B - \sin A\sin B, & (5.9)\\ \cos(A-B) &= \cos A\cos B + \sin A\sin B, & (5.10) \end{aligned}$$

it can be shown that the Chebyshev polynomials satisfy the **three-term recurrence relation**

$$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x), \quad k \geq 1. \tag{5.11}$$

It can easily be seen from this relation, and the first two Chebyshev polynomials, that $T_k(x)$ is in fact a polynomial for all integers $k \geq 0$.

The Chebyshev polynomials have the following properties of interest:

1. The leading coefficient of $T_k(x)$ is 2^{k-1}.
2. $T_k(x)$ is an even function of k is even, and an odd function if k is odd.
3. The zeros of $T_k(x)$, for $k \geq 1$, are
$$x_j = \cos\frac{(2j-1)\pi}{2k}, \quad j = 1, 2, \ldots, k.$$
4. The extrema of $T_k(x)$ on $[-1, 1]$ are
$$\tilde{x}_j = \cos\frac{j\pi}{k}, \quad j = 0, 1, \ldots, k,$$
and the corresponding extremal values are ±1.
5. $|T_k(x)| \leq 1$ on $[-1, 1]$ for all $k \geq 0$.

Exercise 5.4.3 *Use (5.8), (5.9), and (5.10) to prove (5.11).*

Exercise 5.4.4 *Use (5.11) and induction to show that the leading coefficient of* $T_k(x)$ *is* 2^{k-1}*, for* $k \geq 1$*.*

Exercise 5.4.5 *Use the roots of cosine to compute the roots of* $T_k(x)$*. Show that they are real, distinct, and lie within* $(-1, 1)$*. These roots are known as the* **Chebyshev points**.

Let $f(x)$ be a function that is $(n+1)$ times continuously differentiable on $[a, b]$. If we approximate $f(x)$ by a nth-degree polynomial $p_n(x)$ that interpolates $f(x)$ at the $n+1$ roots of the Chebyshev polynomial $T_{n+1}(x)$, mapped from $[-1, 1]$ to $[a, b]$,

$$\xi_j = \frac{1}{2}(b-a)\cos\frac{(2j+1)\pi}{2n+2} + \frac{1}{2}(a+b),$$

then the error in this approximation is

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\left(\frac{b-a}{2}\right)^{n+1} 2^{-n} T_{n+1}(t(x)),$$

where

$$t(x) = -1 + \frac{2}{b-a}(x-a)$$

is the linear map from $[a, b]$ to $[-1, 1]$. This is because

$$\prod_{j=0}^{n}(x - \xi_j) = \left(\frac{b-a}{2}\right)^{n+1} \prod_{j=0}^{n}(t(x) - \tau_j) = \left(\frac{b-a}{2}\right)^{n+1} 2^{-n} T_{n+1}(t(x)),$$

where τ_j is the jth root of $T_{n+1}(t)$. From $|T_{n+1}(t)| \leq 1$, we obtain

$$|f(x) - p_n(x)| \leq \frac{(b-a)^{n+1}}{2^{2n+1}(n+1)!} \max_{\xi \in [a,b]} |f^{(n+1)}(\xi)|.$$

It can be shown that using Chebyshev points leads to much less error in the function $f(x) = 1/(1+x^2)$ from Runge's example [28].

5.5 Osculatory Interpolation

Suppose that the interpolation points are perturbed so that two neighboring points x_i and x_{i+1}, $0 \leq i < n$, approach each other. What happens to the interpolating polynomial? In the limit, as $x_{i+1} \to x_i$, the interpolating polynomial $p_n(x)$ not only satisfies $p_n(x_i) = y_i$, but also the condition

$$p_n'(x_i) = \lim_{x_{i+1} \to x_i} \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

It follows that in order to ensure uniqueness, the data must specify the value of the derivative of the interpolating polynomial at x_i.

In general, the inclusion of an interpolation point x_i k times within the set $x_0, \ldots, x_n$ must be accompanied by specification of $p_n^{(j)}(x_i)$, $j = 0, \ldots, k-1$, in order to ensure a unique solution. These values are used in place of divided differences of identical interpolation points in Newton interpolation.

Interpolation with repeated interpolation points is called **osculatory interpolation**, since it can be viewed as the limit of distinct interpolation points approaching one another, and the term "osculatory" is based on the Latin word for "kiss".

Exercise 5.5.1 *Use a divided-difference table to compute the interpolating polynomial* $p_2(x)$ *for the function* $f(x) = \cos x$ *on* $[0, \pi]$, *with interpolation points* $x_0 = x_1 = 0$ *and* $x_2 = \pi$. *That is,* $p_2(x)$ *must satisfy* $p_2(0) = f(0)$, $p_2'(0) = f'(0)$ *and* $p_2(\pi) = f(\pi)$. *Then, extend* $p_2(x)$ *to a cubic polynomial* $p_3(x)$ *by also requiring that* $p_2'(\pi) = f(\pi)$, *updating the divided-difference table accordingly.*

Exercise 5.5.2 *Suppose that osculatory interpolation is used to construct the polynomial* $p_n(x)$ *that interpolates* $f(x)$ *at only one* x*-value,* x_0, *and satisfies* $p_n(x_0) = f(x_0)$, $p_n'(x_0) = f'(x_0)$, $p_n''(x_0) = f''(x_0)$, *and so on, up to* $p_n^{(n)}(x_0) = f^{(n)}(x_0)$. *What polynomial approximation of* $f(x)$ *is obtained?*

5.5.1 Hermite Interpolation

In the case where each of the interpolation points $x_0, x_1, \ldots, x_n$ is repeated exactly once, the interpolating polynomial for a differentiable function $f(x)$ is called the **Hermite polynomial** of $f(x)$, and is denoted by $p_{2n+1}(x)$, since this polynomial must have degree $2n+1$ in order to satisfy the $2n+2$ constraints

$$p_{2n+1}(x_i) = f(x_i), \quad p_{2n+1}'(x_i) = f'(x_i), \quad i = 0, 1, \ldots, n.$$

To satisfy these constraints, we define, for $i = 0, 1, \ldots, n$,

$$H_i(x) = [L_i(x)]^2(1 - 2L_i'(x_i)(x - x_i)), \tag{5.12}$$

$$K_i(x) = [L_i(x)]^2(x - x_i), \tag{5.13}$$

where, as before, $L_i(x)$ is the ith Lagrange polynomial for the interpolation points $x_0, x_1, \ldots, x_n$.

It can be verified directly that these polynomials satisfy, for $i, j = 0, 1, \ldots, n$,

$$H_i(x_j) = \delta_{ij}, \quad H_i'(x_j) = 0,$$

$$K_i(x_j) = 0, \quad K_i'(x_j) = \delta_{ij},$$

where δ_{ij} is the *Kronecker delta*

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}.$$

It follows that

$$p_{2n+1}(x) = \sum_{i=0}^{n} [f(x_i)H_i(x) + f'(x_i)K_i(x)]$$

is a polynomial of degree $2n+1$ that satisfies the above constraints.

Exercise 5.5.3 *Derive the formulas (5.12), (5.13) for $H_i(x)$ and $K_i(x)$, respectively, using the specified constraints for these polynomials.* Hint: *use an approach similar to that used to derive the formula for Lagrange polynomials.*

To prove that this polynomial is the *unique* polynomial of degree $2n+1$, we assume that there is another polynomial $\tilde{p}_{2n+1}$ of degree $2n+1$ that satisfies the constraints. Because $p_{2n+1}(x_i) = \tilde{p}_{2n+1}(x_i) = f(x_i)$ for $i = 0, 1, \ldots, n$, $p_{2n+1} - \tilde{p}_{2n+1}$ has at least $n+1$ zeros. It follows from Rolle's Theorem that $p_{2n+1}' - \tilde{p}_{2n+1}'$ has n zeros that lie within the intervals (x_{i-1}, x_i) for $i = 0, 1, \ldots, n-1$.

Furthermore, because $p_{2n+1}'(x_i) = \tilde{p}_{2n+1}'(x_i) = f'(x_i)$ for $i = 0, 1, \ldots, n$, it follows that $p_{2n+1}' - \tilde{p}_{2n+1}'$ has $n+1$ additional zeros, for a total of at least $2n+1$. However, $p_{2n+1}' - \tilde{p}_{2n+1}'$ is a polynomial of degree $2n$, and the only way that a polynomial of degree $2n$ can have $2n+1$ zeros is if it is identically zero. Therefore, $p_{2n+1} - \tilde{p}_{2n+1}$ is a constant function, but since this function is known to have at least $n+1$ zeros, that constant must be zero, and the Hermite polynomial is unique.

Using a similar approach as for the Lagrange interpolating polynomial, the following result can be proved.

Theorem 5.5.1 *Let f be $2n+2$ times continuously differentiable on $[a, b]$, and let p_{2n+1} denote the Hermite polynomial of f with interpolation points $x_0, x_1, \ldots, x_n$ in $[a, b]$. Then there exists a point $\xi(x) \in [a, b]$ such that*

$$f(x) - p_{2n+1}(x) = \frac{f^{(2n+2)}(\xi(x))}{(2n+2)!}(x - x_0)^2(x - x_1)^2 \cdots (x - x_n)^2.$$

The proof will be left as an exercise.

5.5.2 Divided Differences

The Hermite polynomial can be described using Lagrange polynomials and their derivatives, but this representation is not practical because of the difficulty of differentiating and evaluating these polynomials. Instead, one can construct the Hermite polynomial using a divided-difference table, as discussed previously, in which each entry corresponding to two identical interpolation points is filled with the value of $f'(x)$ at the common point. Then, the Hermite polynomial can be represented using the Newton divided-difference formula.

Example 5.5.2 *We will use Hermite interpolation to construct the third-degree polynomial $p_3(x)$ that fits $f(x)$ and $f'(x)$ at $x_0 = 0$ and $x_1 = 1$. For convenience, we define new interpolation points z_i that list each (distinct) x-value twice:*

$$z_{2i} = z_{2i+1} = x_i, \quad i = 0, 1, \ldots, n.$$

Our data is as follows:

i	z_i	$f(z_i)$	$f'(z_i)$
0,1	0	0	1
2,3	1	0	1

In other words, we must have $p_3(0) = 0$, $p_3'(0) = 1$, $p_3(1) = 0$, *and* $p_3'(1) = 1$. *To include the values of* $f'(x)$ *at the two distinct interpolation points, we repeat each point once, so that the number of interpolation points, including repetitions, is equal to the number of constraints described by the data.*

First, we construct the divided-difference table from this data. The divided differences in the table are computed as follows:

$$f[z_0] = f(z_0) = 0, \quad f[z_1] = f(z_1) = 0, \quad f[z_2] = f(z_2) = 0, \quad f[z_3] = f(z_3) = 0,$$

$$\begin{aligned}
f[z_0, z_1] &= \frac{f[z_1] - f[z_0]}{z_1 - z_0} \\
&= f'(z_0) \\
&= 1 \\
f[z_1, z_2] &= \frac{f[z_2] - f[z_1]}{z_2 - z_1} \\
&= \frac{0 - 0}{1 - 0} \\
&= 0 \\
f[z_2, z_3] &= \frac{f[z_3] - f[z_2]}{z_3 - z_2} \\
&= f'(z_2) \\
&= 1 \\
f[z_0, z_1, z_2] &= \frac{f[z_1, z_2] - f[z_0, z_1]}{z_2 - z_0} \\
&= \frac{0 - 1}{1 - 0} \\
&= -1 \\
f[z_1, z_2, z_3] &= \frac{f[z_2, z_3] - f[z_1, z_2]}{z_3 - z_1} \\
&= \frac{1 - 0}{1 - 0} \\
&= 1 \\
f[z_0, z_1, z_2, z_3] &= \frac{f[z_1, z_2, z_3] - f[z_0, z_1, z_2]}{z_3 - z_0} \\
&= \frac{1 - (-1)}{1 - 0} \\
&= 2
\end{aligned}$$

Note that the values of the derivative are used whenever a divided difference of the form $f[z_i, z_{i+1}]$

is to be computed, where $z_i = z_{i+1}$*. This makes sense because*

$$\lim_{z_{i+1}\to z_i} f[z_i, z_{i+1}] = \lim_{z_{i+1}\to z_i} \frac{f(z_{i+1}) - f(z_i)}{z_{i+1} - z_i} = f'(z_i).$$

The resulting divided-difference table is

$z_0 = 0$	$f[z_0] = 0$			
		$f[z_0, z_1] = 1$		
$z_1 = 0$	$f[z_1] = 0$		$f[z_0, z_1, z_2] = -1$	
		$f[z_1, z_2] = 0$		$f[z_0, z_1, z_2, z_3] = 2$
$z_2 = 1$	$f[z_2] = 0$		$f[z_1, z_2, z_3] = 1$	
		$f[z_2, z_3] = 1$		
$z_3 = 1$	$f[z_3] = 0$			

It follows that the interpolating polynomial $p_3(x)$ *can be obtained using the Newton divided-difference formula as follows:*

$$\begin{aligned} p_3(x) &= \sum_{j=0}^{3} f[z_0, \ldots, z_j] \prod_{i=0}^{j-1} (x - z_i) \\ &= f[z_0] + f[z_0, z_1](x - z_0) + f[z_0, z_1, z_2](x - z_0)(x - z_1) + \\ &\quad f[z_0, z_1, z_2, z_3](x - z_0)(x - z_1)(x - z_2) \\ &= 0 + (x - 0) + (-1)(x - 0)(x - 0) + 2(x - 0)(x - 0)(x - 1) \\ &= x - x^2 + 2x^2(x - 1). \end{aligned}$$

We see that Hermite interpolation, using divided differences, produces an interpolating polynomial that is in the Newton form, with centers $z_0 = 0$*,* $z_1 = 0$*, and* $z_2 = 1$*.* □

Exercise 5.5.4 *Use the Newton form of the Hermite interpolating polynomial to prove Theorem 5.5.1.*

Exercise 5.5.5 *Write a* MATLAB *function* `c=hermdivdiffs(x,y,yp)` *that computes the divided difference table from the given data stored in the input vectors* `x` *and* `y`*, as well as the derivative values stored in* `yp`*, and returns a vector* `c` *consisting of the divided differences* $f[z_0, \ldots, z_j]$*,* $j = 0, 1, 2, \ldots, 2n + 1$*, where* $n + 1$ *is the length of both* `x` *and* `y`*.*

Exercise 5.5.6 *Write a* MATLAB *function* `p=hermpolyfit(x,y,yp)` *that is similar to the built-in function* `polyfit`*, in that it returns a vector of coefficients, in power form, for the interpolating polynomial corresponding to the given data, except that Hermite interpolation is used instead of Lagrange interpolation. Use the function* `hermdivdiffs` *from Exercise 5.5.5 as well as the function* `powerform` *from Exercise 5.3.7.*

5.6 Piecewise Polynomial Interpolation

We have seen that high-degree polynomial interpolation can be problematic. However, if the fitting function is only required to have a few continuous derivatives, then one can construct a **piecewise polynomial** to fit the data. We now precisely define what we mean by a piecewise polynomial.

Definition 5.6.1 (Piecewise polynomial) *Let $[a,b]$ be an interval that is divided into subintervals $[x_i, x_{i+1}]$, where $i = 0, \ldots, n-1$, $x_0 = a$ and $x_n = b$. A* **piecewise polynomial** *is a function $p(x)$ defined on $[a,b]$ by*

$$p(x) = p_i(x), \quad x_{i-1} \leq x \leq x_i, \quad i = 1, 2, \ldots, n,$$

where, for $i = 1, 2, \ldots, n$, each function $p_i(x)$ is a polynomial defined on $[x_{i-1}, x_i]$. The **degree** *of $p(x)$ is the maximum degree of each polynomial $p_i(x)$, for $i = 1, 2, \ldots, n$.*

It is essential to note that by this definition, a piecewise polynomial defined on $[a,b]$ is equal to some polynomial on each subinterval $[x_{i-1}, x_i]$ of $[a,b]$, for $i = 1, 2, \ldots, n$, but a different polynomial may be used for each subinterval.

To study the accuracy of piecewise polynomials, we need to work with various function spaces, including **Sobolev spaces**; these function spaces are defined in Section B.12.

5.6.1 Piecewise Linear Approximation

We first consider one of the simplest types of piecewise polynomials, a piecewise linear polynomial. Let $f \in C[a,b]$. Given the points $x_0, x_1, \ldots, x_n$ defined as above, the **linear spline** $s_{\mathrm{L}}(x)$ that interpolates f at these points is defined by

$$s_{\mathrm{L}}(x) = f(x_{i-1}) \frac{x - x_i}{x_{i-1} - x_i} + f(x_i) \frac{x - x_{i-1}}{x_i - x_{i-1}}, \quad x \in [x_{i-1}, x_i], \quad i = 1, 2, \ldots, n. \tag{5.14}$$

The points $x_0, x_1, \ldots, x_n$ are the **knots** of the spline.

Exercise 5.6.1 *Given that $s_{\mathrm{L}}(x)$ must satisfy $s_L(x_i) = f(x_i)$ for $i = 0, 1, 2, \ldots, n$, explain how the formula (5.14) can be derived.*

Exercise 5.6.2 *Given the points $(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n))$ plotted in the xy-plane, explain how s_{L} can easily be graphed. How can the graph be produced in a single line of* MATLAB *code, given vectors* `x` *and* `y` *containing the x- and y- coordinates of these points, respectively?*

If $f \in C^2[a,b]$, then by the error in Lagrange interpolation (Theorem 5.4.1), on each subinterval $[x_{i-1}, x_i]$, for $i = 1, 2, \ldots, n$, we have

$$f(x) - s_{\mathrm{L}}(x) = \frac{f''(\xi)}{2}(x - x_{i-1})(x - x_i).$$

This leads to the following Theorem.

Theorem 5.6.2 *Let $f \in C^2[a,b]$, and let s_{L} be the piecewise linear spline defined by (5.14). For $i = 1, 2, \ldots, n$, let $h_i = x_i - x_{i-1}$, and define $h = \max_{1 \leq i \leq n} h_i$. Then*

$$\|f - s_{\mathrm{L}}\|_\infty \leq \frac{M}{8} h^2,$$

where $|f''(x)| \leq M$ on $[a,b]$.

Exercise 5.6.3 *Prove Theorem 5.6.2.*

In Section 5.4, it was observed in Runge's example that even when $f(x)$ is smooth, an interpolating polynomial of $f(x)$ can be highly oscillatory, depending on the number and placement of interpolation points. By contrast, one of the most welcome properties of the linear spline $s_{\rm L}(x)$ is that among all functions in $H^1(a,b)$ that interpolate $f(x)$ at the knots $x_0, x_1, \ldots, x_n$, it is the "flattest". That is, for any function $v \in H^1(a,b)$ that interpolates f at the knots,

$$\|s_{\rm L}'\|_2 \leq \|v'\|_2.$$

To prove this, we first write

$$\|v'\|_2^2 = \|v' - s_{\rm L}'\|_2^2 + 2\langle v' - s_{\rm L}', s_{\rm L}'\rangle + \|s_{\rm L}'\|_2^2.$$

We then note that on each subinterval $[x_{i-1}, x_i]$, since s_L is a linear function, s_L' is a constant function, which we denote by

$$s_L(x) \equiv m_i = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}, \quad i = 1, 2, \ldots, n.$$

We then have

$$\begin{aligned}
\langle v' - s_{\rm L}', s_{\rm L}'\rangle &= \int_a^b [v'(x) - s_{\rm L}'(x)] s_{\rm L}'(x)\, dx \\
&= \sum_{i=1}^n \int_{x_{i-1}}^{x_i} [v'(x) - s_{\rm L}'(x)] s_{\rm L}'(x)\, dx \\
&= \sum_{i=1}^n m_i \int_{x_{i-1}}^{x_i} v'(x) - m_i\, dx \\
&= \sum_{i=1}^n m_i \left[v(x) - m_i x\right]\Big|_{x_{i-1}}^{x_i} \\
&= \sum_{i=1}^n m_i \left[v(x_i) - v(x_{i-1}) - \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}(x_i - x_{i-1})\right] \\
&= 0,
\end{aligned}$$

because by assumption, $v(x)$ interpolates $f(x)$ at the knots. This leaves us with

$$\|v'\|_2^2 = \|v' - s_L'\|_2^2 + \|s_L'\|_2^2,$$

which establishes the result.

5.6.2 Cubic Spline Interpolation

A major drawback of the linear spline is that it does not have any continuous derivatives. This is significant when the function to be approximated, $f(x)$, is a smooth function. Therefore, it is desirable that a piecewise polynomial approximation possess a certain number of continuous derivatives. This requirement imposes additional constraints on the piecewise polynomial, and therefore the degree of the polynomials used on each subinterval must be chosen sufficiently high to ensure that these constraints can be satisfied.

We therefore define a **spline** of degree k to be a piecewise polynomial of degree k that has $k-1$ continuous derivatives. The most commonly used spline is a **cubic spline**, which we now define.

Definition 5.6.3 (Cubic Spline) *Let* $f(x)$ *be function defined on an interval* $[a, b]$*, and let* $x_0, x_1, \ldots, x_n$ *be* $n+1$ *distinct points in* $[a, b]$*, where* $a = x_0 < x_1 < \cdots < x_n = b$*. A* **cubic spline***, or* **cubic spline interpolant***, is a piecewise polynomial* $s(x)$ *that satisifes the following conditions:*

1. *On each interval* $[x_{i-1}, x_i]$*, for* $i = 1, \ldots, n$*,* $s(x) = s_i(x)$*, where* $s_i(x)$ *is a cubic polynomial.*
2. $s(x_i) = f(x_i)$ *for* $i = 0, 1, \ldots, n$*.*
3. $s(x)$ *is twice continuously differentiable on* (a, b)*.*
4. *Either of the following boundary conditions are satisfied:*
    (a) $s''(a) = s''(b) = 0$*, which is called* **free** *or* **natural boundary conditions***, and*
    (b) $s'(a) = f'(a)$ *and* $s'(b) = f'(b)$*, which is called* **clamped boundary conditions**.

If $s(x)$ *satisfies free boundary conditions, we say that* $s(x)$ *is a* **natural spline**. *The points* $x_0, x_1, \ldots, x_n$ *are called the* **nodes** *of* $s(x)$*.*

Clamped boundary conditions are often preferable because they use more information about $f(x)$, which yields a spline that better approximates $f(x)$ on $[a, b]$. However, if information about $f'(x)$ is not available, then natural boundary conditions must be used instead.

5.6.2.1 Constructing Cubic Splines

Suppose that we wish to construct a cubic spline interpolant $s(x)$ that fits the given data (x_0, y_0), $(x_1, y_1), \ldots, (x_n, y_n)$, where $a = x_0 < x_1 < \cdots < x_n = b$, and $y_i = f(x_i)$, for some known function $f(x)$ defined on $[a, b]$. From the preceding discussion, this spline is a piecewise polynomial of the form

$$s(x) = s_i(x) = d_i(x - x_{i-1})^3 + c_i(x - x_{i-1})^2 + b_i(x - x_{i-1}) + a_i, \quad i = 1, 2, \ldots, n, \quad x_{i-1} \leq x \leq x_i. \tag{5.15}$$

That is, the value of $s(x)$ is obtained by evaluating a different cubic polynomial for each subinterval $[x_{i-1}, x_i]$, for $i = 1, 2, \ldots, n$.

We now use the definition of a cubic spline to construct a system of equations that must be satisfied by the coefficients a_i, b_i, c_i and d_i for $i = 1, 2, \ldots, n$. We can then compute these coefficients by solving the system. Because $s(x)$ must fit the given data, we have

$$s(x_{i-1}) = a_i = y_{i-1} = f(x_{i-1}), \quad i = 1, 2, \ldots, n. \tag{5.16}$$

If we define $h_i = x_i - x_{i-1}$, for $i = 1, 2, \ldots, n$, and define $a_{n+1} = y_n$, then the requirement that $s(x)$ is continuous at the interior nodes implies that we must have $s_i(x_i) = s_{i+1}(x_i)$ for $i = 1, 2, \ldots, n-1$.

Furthermore, because $s(x)$ must fit the given data, we must also have $s(x_n) = s_n(x_n) = y_n$. These conditions lead to the constraints

$$s_i(x_i) = d_i h_i^3 + c_i h_i^2 + b_i h_i + a_i = a_{i+1} = s_{i+1}(x_i), \quad i = 1, 2, \ldots, n. \tag{5.17}$$

To ensure that $s(x)$ has a continuous first derivative at the interior nodes, we require that $s_i'(x_i) = s_{i+1}'(x_i)$ for $i = 1, 2 \ldots, n-1$, which imposes the constraints

$$s_i'(x_i) = 3d_i h_i^2 + 2c_i h_i + b_i = b_{i+1} = s_{i+1}'(x_i), \quad i = 1, 2, \ldots, n-1. \tag{5.18}$$

Similarly, to enforce continuity of the second derivative at the interior nodes, we require that $s_i''(x_i) = s_{i+1}''(x_i)$ for $i = 1, 2, \ldots, n-1$, which leads to the constraints

$$s_i''(x_i) = 6d_i h_i + 2c_i = 2c_{i+1} = s_{i+1}''(x_i), \quad i = 1, 2, \ldots, n-1. \tag{5.19}$$

There are $4n$ coefficients to determine, since there are n cubic polynomials, with 4 coefficients each. However, we have only prescribed $4n - 2$ constraints, so we must specify 2 more in order to determine a unique solution. If we use natural boundary conditions, then these constraints are

$$\begin{aligned} s_1''(x_0) = 2c_1 &= 0, & (5.20) \\ s_n''(x_n) = 3d_n h_n + c_n &= 0. & (5.21) \end{aligned}$$

On the other hand, if we use clamped boundary conditions, then our additional constraints are

$$\begin{aligned} s_1'(x_0) = b_1 &= z_0, & (5.22) \\ s_n'(x_n) = 3d_n h_n^2 + 2c_n h_n + b_n &= z_n, & (5.23) \end{aligned}$$

where $z_i = f'(x_i)$ for $i = 0, 1, \ldots, n$.

Having determined our constraints that must be satisfied by $s(x)$, we can set up a system of $4n$ linear equations based on these constraints, and then solve this system to determine the coefficients a_i, b_i, c_i, d_i for $i = 1, 2 \ldots, n$.

However, it is not necessary to construct the matrix for such a system, because it is possible to instead solve a smaller system of only $O(n)$ equations obtained from the continuity conditions (5.18) and the boundary conditions (5.20), (5.21) or (5.22), (5.23), depending on whether natural or clamped boundary conditions, respectively, are imposed. This reduced system is accomplished by using equations (5.16), (5.17) and (5.19) to eliminate the a_i, b_i and d_i, respectively.

Exercise 5.6.4 *Show that under natural boundary conditions, the coefficients $c_2, \ldots, c_n$ of the cubic spline (5.15) satisfy the system of equations $A\mathbf{c} = \mathbf{b}$, where*

$$A = \begin{bmatrix} 2(h_1 + h_2) & h_2 & 0 & \cdots & 0 \\ h_2 & 2(h_2 + h_3) & h_3 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & h_{n-1} \\ 0 & \cdots & 0 & h_{n-1} & 2(h_{n-1} + h_n) \end{bmatrix},$$

$$\mathbf{c} = \begin{bmatrix} c_2 \\ \vdots \\ c_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \frac{3}{h_2}(a_3 - a_2) - \frac{3}{h_1}(a_2 - a_1) \\ \vdots \\ \frac{3}{h_n}(a_{n+1} - a_n) - \frac{3}{h_{n-1}}(a_n - a_{n-1}) \end{bmatrix}.$$

Exercise 5.6.5 *Show that under clamped boundary conditions, the coefficients* $c_1, \ldots, c_{n+1}$ *of the cubic spline (5.15) satisfy the system of equations* $A\mathbf{c} = \mathbf{b}$*, where*

$$A = \begin{bmatrix} 2h_1 & h_1 & 0 & \cdots & \cdots & 0 \\ h_1 & 2(h_1+h_2) & h_2 & \ddots & & \vdots \\ 0 & h_2 & 2(h_2+h_3) & h_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & h_{n-1} & 2(h_{n-1}+h_n) & h_n \\ 0 & \cdots & \cdots & 0 & h_n & 2h_n \end{bmatrix},$$

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n+1} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \frac{3}{h_1}(a_2 - a_1) - 3z_0 \\ \frac{3}{h_2}(a_3 - a_2) - \frac{3}{h_1}(a_2 - a_1) \\ \vdots \\ \frac{3}{h_n}(a_{n+1} - a_n) - \frac{3}{h_{n-1}}(a_n - a_{n-1}) \\ 3z_n - \frac{3}{h_n}(a_{n+1} - a_n) \end{bmatrix},$$

and $c_{n+1} = s_n''(x_n)$.

Example 5.6.4 *We will construct a cubic spline interpolant for the following data on the interval* $[0, 2]$.

j	x_j	y_j
0	0	3
1	1/2	−4
2	1	5
3	3/2	−6
4	2	7

The spline, $s(x)$*, will consist of four pieces* $\{s_j(x)\}_{j=1}^4$*, each of which is a cubic polynomial of the form*

$$s_j(x) = a_j + b_j(x - x_{j-1}) + c_j(x - x_{j-1})^2 + d_j(x - x_{j-1})^3, \quad j = 1, 2, 3, 4.$$

We will impose natural boundary conditions on this spline, so it will satisfy the conditions $s''(0) = s''(2) = 0$*, in addition to the "essential" conditions imposed on a spline: it must fit the given data and have continuous first and second derivatives on the interval* $[0, 2]$.

These conditions lead to the following system of equations that must be solved for the coefficients c_1, c_2, c_3, c_4*, and* c_5*, where* $c_j = s''(x_{j-1})/2$ *for* $j = 1, 2, \ldots, 5$*. We define* $h = (2 - 0)/4 = 1/2$ *to be the spacing between the interpolation points.*

$$\begin{aligned} c_1 &= 0 \\ \frac{h}{3}(c_1 + 4c_2 + c_3) &= \frac{y_2 - 2y_1 + y_0}{h} \\ \frac{h}{3}(c_2 + 4c_3 + c_4) &= \frac{y_3 - 2y_2 + y_1}{h} \\ \frac{h}{3}(c_3 + 4c_4 + c_5) &= \frac{y_4 - 2y_3 + y_2}{h} \\ c_5 &= 0. \end{aligned}$$

Substituting $h = 1/2$ *and the values of* y_j*, and also taking into account the boundary conditions, we obtain*

$$\begin{aligned} \frac{1}{6}(4c_2 + c_3) &= 32 \\ \frac{1}{6}(c_2 + 4c_3 + c_4) &= -40 \\ \frac{1}{6}(c_3 + 4c_4) &= 48 \end{aligned}$$

This system has the solutions

$$c_2 = 516/7, \quad c_3 = -720/7, \quad c_4 = 684/7.$$

Using (5.16), (5.17), and (5.19), we obtain

$$a_1 = 3, \quad a_2 = -4, \quad a_3 = 5, \quad a_4 = -6.$$

$$b_1 = -184/7, \quad b_2 = 74/7, \quad b_3 = -4, \quad b_4 = -46/7,$$

and

$$d_1 = 344/7, \quad d_2 = -824/7, \quad d_3 = 936/7, \quad d_4 = -456/7.$$

We conclude that the spline $s(x)$ *that fits the given data, has two continuous derivatives on* $[0, 2]$*, and satisfies natural boundary conditions is*

$$s(x) = \begin{cases} \frac{344}{7}x^3 - \frac{184}{7}x^2 + 3 & \text{if } x \in [0, 0.5] \\ -\frac{824}{7}(x - 1/2)^3 + \frac{516}{7}(x - 1/2)^2 + \frac{74}{7}(x - 1/2) - 4 & \text{if } x \in [0.5, 1] \\ \frac{936}{7}(x - 1)^3 - \frac{720}{7}(x - 1)^2 - 4(x - 1) + 5 & \text{if } x \in [1, 1.5] \\ -\frac{456}{7}(x - 3/2)^3 + \frac{684}{7}(x - 3/2)^2 - \frac{46}{7}(x - 3/2) - 6 & \text{if } x \in [1.5, 2] \end{cases}.$$

The graph of the spline is shown in Figure 5.2. □

The Matlab function `spline` can be used to construct cubic splines satisfying both natural (also known as "not-a-knot") and clamped boundary conditions. The following exercises require reading the documentation for this function.

Exercise 5.6.6 *Use* `spline` *to construct a cubic spline for the data from Example 5.6.4. First, use the interface* `pp=spline(x,y)`*, where* `x` *and* `y` *are vectors consisting of the* x*- and* y*-coordinates, respectively, of the given data points, and* `pp` *is a structure that represents the cubic spline* $s(x)$*. Examine the members of* `p` *and determine how to interpret them. Where do you see the coefficients computed in Example 5.6.4?*

Exercise 5.6.7 *The interface* `yy=spline(x,y,xx)`*, where* `xx` *is a vector of* x*-values at which the spline constructed from* `x` *and* `y` *should be evaluated, produces a vector* `yy` *of corresponding* y*-values. Use this interface on the data from Example 5.6.4 to reproduce Figure 5.2.*

Exercise 5.6.8 *If the input argument* `y` *in the function call* `pp=spline(x,y)` *has two components more than* `x`*, it is assumed that the first and last components are the slopes* $z_0 = s'(x_0)$ *and* $z_n = s'(x_n)$ *imposed by clamped boundary conditions. Use the given data from Example 5.6.4, with various values of* z_0 *and* z_n*, and construct the clamped cubic spline using this interface to* `spline`*. Compare the coefficients and graphs to that of the natural cubic spline from Exercises 5.6.6 and 5.6.7.*

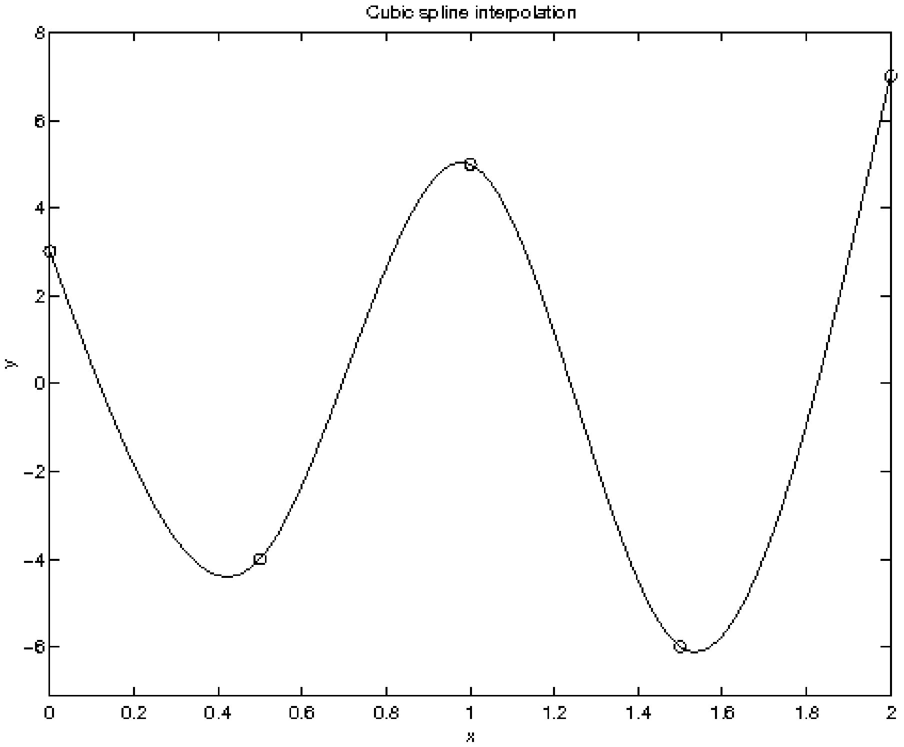


Figure 5.2: Cubic spline that passing through the points $(0, 3)$, $(1/2, -4)$, $(1, 5)$, $(2, -6)$, and $(3, 7)$.

5.6.2.2 Well-Posedness and Accuracy

For both sets of boundary conditions, the system $A\mathbf{c} = \mathbf{b}$ has a unique solution, because the matrix A is strictly row diagonally dominant. This property guarantees that A is invertible, due to Gerschgorin's Circle Theorem. We therefore have the following results.

Theorem 5.6.5 *Let $x_0, x_1, \ldots, x_n$ be $n+1$ distinct points in the interval $[a,b]$, where $a = x_0 < x_1 < \cdots < x_n = b$, and let $f(x)$ be a function defined on $[a,b]$. Then f has a unique cubic spline interpolant $s(x)$ that is defined on the nodes $x_0, x_1, \ldots, x_n$ that satisfies the natural boundary conditions $s''(a) = s''(b) = 0$.*

Theorem 5.6.6 *Let $x_0, x_1, \ldots, x_n$ be $n+1$ distinct points in the interval $[a,b]$, where $a = x_0 < x_1 < \cdots < x_n = b$, and let $f(x)$ be a function defined on $[a,b]$ that is differentiable at a and b. Then f has a unique cubic spline interpolant $s(x)$ that is defined on the nodes $x_0, x_1, \ldots, x_n$ that satisfies the clamped boundary conditions $s'(a) = f'(a)$ and $s'(b) = f'(b)$.*

Just as the linear spline is the "flattest" interpolant, in an average sense, the natural cubic spline has the least "average curvature". Specifically, if $s_2(x)$ is the natural cubic spline for $f \in C[a,b]$ on $[a,b]$ with knots $a = x_0 < x_1 < \cdots < x_n = b$, and $v \in H^2(a,b)$ is any interpolant of f with these knots, then

$$\|s_2''\|_2 \leq \|v''\|_2.$$

This can be proved in the same way as the corresponding result for the linear spline. It is this property of the natural cubic spline, called the **smoothest interpolation property**, from which splines were named.

The following result, proved in [34, p. 57-58], provides insight into the accuracy with which a cubic spline interpolant $s(x)$ approximates a function $f(x)$.

Theorem 5.6.7 *Let f be four times continuously differentiable on $[a,b]$, and assume that $\|f^{(4)}\|_\infty = M$. Let $s(x)$ be the unique clamped cubic spline interpolant of $f(x)$ on the nodes $x_0, x_1, \ldots, x_n$, where $a = x_0 < x_1 < \cdots < x_n < b$. Then for $x \in [a,b]$,*

$$\|f(x) - s(x)\|_\infty \leq \frac{5M}{384} \max_{1 \leq i \leq n} h_i^4,$$

where $h_i = x_i - x_{i-1}$.

A similar result applies in the case of natural boundary conditions [6].

5.6.2.3 Hermite Cubic Splines

We have seen that it is possible to construct a piecewise cubic polynomial that interpolates a function $f(x)$ at knots $a = x_0 < x_1 < \cdots < x_n = b$, that belongs to $C^2[a,b]$. Now, suppose that we also know the values of $f'(x)$ at the knots. We wish to construct a piecewise cubic polynomial $s(x)$ that agrees with $f(x)$, and whose derivative agrees with $f'(x)$ at the knots. This piecewise polynomial is called a **Hermite cubic spline**.

Because $s(x)$ is cubic on each subinterval $[x_{i-1}, x_i]$ for $i = 1, 2, \ldots, n$, there are $4n$ coefficients, and therefore $4n$ degrees of freedom, that can be used to satisfy any criteria that are imposed on

$s(x)$. Requiring that $s(x)$ interpolates $f(x)$ at the knots, and that $s'(x)$ interpolates $f'(x)$ at the knots, imposes $2n+2$ constraints on the coefficients. We can then use the remaining $2n-2$ degrees of freedom to require that $s(x)$ belong to $C^1[a,b]$; that is, it is continuously differentiable on $[a,b]$. Note that unlike the cubic spline interpolant, the Hermite cubic spline does *not* have a continuous second derivative.

The following result provides insight into the accuracy with which a Hermite cubic spline interpolant $s(x)$ approximates a function $f(x)$.

Theorem 5.6.8 *Let f be four times continuously differentiable on $[a,b]$, and assume that $\|f^{(4)}\|_\infty = M$. Let $s(x)$ be the unique Hermite cubic spline interpolant of $f(x)$ on the nodes $x_0, x_1, \ldots, x_n$, where $a = x_0 < x_1 < \cdots < x_n < b$. Then*

$$\|f(x) - s(x)\|_\infty \leq \frac{M}{384} \max_{1\leq i\leq n} h_i^4,$$

where $h_i = x_i - x_{i-1}$.

This can be proved in the same way as the error bound for the linear spline, except that the error formula for Hermite interpolation is used instead of the error formula for Lagrange interpolation.

Exercise 5.6.9 *Prove Theorem 5.6.8.*

An advantage of Hermite cubic splines over cubic spline interpolants is that they are *local* approximations rather than *global*; that is, if the values of $f(x)$ and $f'(x)$ change at some knot x_i, only the polynomials defined on the pieces containing x_i need to be changed. In cubic spline interpolation, all pieces are coupled, so a change at one point changes the polynomials for all pieces.

To see this, we represent the Hermite cubic spline using the same form as in the cubic spline interpolant,

$$s_i(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3, \quad x \in [x_{i-1}, x_i], \tag{5.24}$$

for $i = 1, 2, \ldots, n$. Then, the coefficients a_i, b_i, c_i, d_i can be determined explicitly in terms of only $f(x_{i-1})$, $f'(x_{i-1})$, $f(x_i)$ and $f'(x_i)$.

Exercise 5.6.10 *Use the conditions*

$$s_i(x_{i-1}) = f(x_{i-1}), \quad s_i(x_i) = f(x_i), \quad s_i'(x_{i-1}) = f'(x_{i-1}), \quad s_i'(x_i) = f'(x_i)$$

to obtain the values of the coefficients a_i, b_i, c_i, d_i in (5.24).

Exercise 5.6.11 *Write a* MATLAB *function* `hp=hermitespline(x,y)` *that constructs a Hermite cubic spline for the data given in the vectors* `x` *and* `y`*. The output* `hp` *should be a structure that contains enough information so that the spline can be evaluated at any x-value without having to specify any additional parameters. Write a second function* `y=hsplineval(hp,x)` *that performs this evaluation.*

Chapter 6

Approximation of Functions

Previously we have considered the problem of polynomial *interpolation*, in which a function $f(x)$ is approximated by a polynomial $p_n(x)$ that agrees with $f(x)$ at $n + 1$ distinct points, based on the assumption that $p_n(x)$ will be, in some sense, a good approximation of $f(x)$ at other points. As we have seen, however, this assumption is not always valid, and in fact, such an approximation can be quite poor, as demonstrated by Runge's example.

Therefore, we consider an alternative approach to approximation of a function $f(x)$ on an interval $[a, b]$ by a polynomial, in which the polynomial is not required to agree with f at any specific points, but rather approximate f well in an "overall" sense, by not deviating much from f at *any* point in $[a, b]$. This requires that we define an appropriate notion of "distance" between functions that is, intuitively, consistent with our understanding of distance between numbers or points in space.

To that end, we can use *vector norms*, as defined in Section B.11, where the vectors in question consist of the values of functions at selected points. In this case, the problem can be reduced to a *least squares problem*, as discussed in Chapter 6. This is discussed in Section 6.1.

Still, finding an approximation of $f(x)$ that is accurate with respect to any discrete, finite subset of the domain cannot guarantee that it accurately approximates $f(x)$ on the entire domain. Therefore, in Section 6.2 we generalize least squares approximations to a *continuous* setting by working with with norms on **function spaces**, which are vector spaces in which the vectors are functions. Such function spaces and norms are reviewed in Section B.12.

In the remainder of the chapter, we consider approximating $f(x)$ by functions other than polynomials. Section 6.3 presents an approach to approximating $f(x)$ with a *rational* function, to overcome the limitations of polynomial approximation, while Section 6.4 explores approximation through *trigonometric polynomials*, or sines and cosines, to capture the frequency content of $f(x)$.

6.1 Discrete Least Squares Approximations

As stated previously, one of the most fundamental problems in science and engineering is *data fitting*–constructing a function that, in some sense, conforms to given data points. So far, we have discussed two data-fitting techniques, polynomial interpolation and piecewise polynomial interpolation. Interpolation techniques, of any kind, construct functions that agree *exactly* with the data. That is, given points (x_1, y_1), (x_2, y_2), $\ldots$, (x_m, y_m), interpolation yields a function $f(x)$ such that

$f(x_i) = y_i$ for $i = 1, 2, \ldots, m$.

However, fitting the data exactly may not be the best approach to describing the data with a function. We have seen that high-degree polynomial interpolation can yield oscillatory functions that behave very differently than a smooth function from which the data is obtained. Also, it may be pointless to try to fit data exactly, for if it is obtained by previous measurements or other computations, it may be erroneous. Therefore, we consider revising our notion of what constitutes a "best fit" of given data by a function.

Let $\mathbf{f} = \left[\begin{array}{cccc} f(x_1) & f(x_2) & \cdots & f(x_n) \end{array}\right]$ and let $\mathbf{y} = \left[\begin{array}{cccc} y_1 & y_2 & \cdots & y_n \end{array}\right]$. One alternative approach to data fitting is to solve the **minimax** problem, which is the problem of finding a function $f(x)$ of a given form for which

$$\|\mathbf{f} - \mathbf{y}\|_\infty = \max_{1 \leq i \leq n} |f(x_i) - y_i|$$

is minimized. However, this is a very difficult problem to solve.

Another approach is to minimize the total **absolute deviation** of $f(x)$ from the data. That is, we seek a function $f(x)$ of a given form for which

$$\|\mathbf{f} - \mathbf{y}\|_1 = \sum_{i=1}^m |f(x_i) - y_i|$$

is minimized. However, we cannot apply standard minimization techniques to this function, because, like the absolute value function that it employs, it is not differentiable.

This defect is overcome by considering the problem of finding $f(x)$ of a given form for which

$$\|\mathbf{f} - \mathbf{y}\|_2^2 = \sum_{i=1}^m [f(x_i) - y_i]^2$$

is minimized. This is known as the **least squares problem**. We will first show how this problem is solved for the case where $f(x)$ is a *linear* function of the form $f(x) = a_1 x + a_0$, and then generalize this solution to other types of functions.

When $f(x)$ is linear, the least squares problem is the problem of finding constants a_0 and a_1 such that the function

$$E(a_0, a_1) = \sum_{i=1}^m (a_1 x_i + a_0 - y_i)^2$$

is minimized. In order to minimize this function of a_0 and a_1, we must compute its partial derivatives with respect to a_0 and a_1. This yields

$$\frac{\partial E}{\partial a_0} = \sum_{i=1}^m 2(a_1 x_i + a_0 - y_i), \quad \frac{\partial E}{\partial a_1} = \sum_{i=1}^m 2(a_1 x_i + a_0 - y_i) x_i.$$

At a minimum, both of these partial derivatives must be equal to zero. This yields the system of linear equations

$$\begin{aligned} m a_0 + \left(\sum_{i=1}^m x_i\right) a_1 &= \sum_{i=1}^m y_i, \\ \left(\sum_{i=1}^m x_i\right) a_0 + \left(\sum_{i=1}^m x_i^2\right) a_1 &= \sum_{i=1}^m x_i y_i. \end{aligned}$$

These equations are called the **normal equations**.

Using the formula for the inverse of a 2×2 matrix,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix},$$

we obtain the solutions

$$\begin{aligned} a_0 &= \frac{\left(\sum_{i=1}^m x_i^2\right)\left(\sum_{i=1}^m y_i\right) - \left(\sum_{i=1}^m x_i\right)\left(\sum_{i=1}^m x_i y_i\right)}{m\sum_{i=1}^m x_i^2 - \left(\sum_{i=1}^m x_i\right)^2}, \\ a_1 &= \frac{m\sum_{i=1}^m x_i y_i - \left(\sum_{i=1}^m x_i\right)\left(\sum_{i=1}^m y_i\right)}{m\sum_{i=1}^m x_i^2 - \left(\sum_{i=1}^m x_i\right)^2}. \end{aligned}$$

Example 6.1.1 *We wish to find the linear function $y = a_1 x + a_0$ that best approximates the data shown in Table 6.1, in the least-squares sense. Using the summations*

Table 6.1: Data points (x_i, y_i), for $i = 1, 2, \ldots, 10$, to be fit by a linear function

i	x_i	y_i
1	2.0774	3.3123
2	2.3049	3.8982
3	3.0125	4.6500
4	4.7092	6.5576
5	5.5016	7.5173
6	5.8704	7.0415
7	6.2248	7.7497
8	8.4431	11.0451
9	8.7594	9.8179
10	9.3900	12.2477

$$\sum_{i=1}^m x_i = 56.2933, \quad \sum_{i=1}^m x_i^2 = 380.5426, \quad \sum_{i=1}^m y_i = 73.8373, \quad \sum_{i=1}^m x_i y_i = 485.9487,$$

we obtain

$$\begin{aligned} a_0 &= \frac{380.5426 \cdot 73.8373 - 56.2933 \cdot 485.9487}{10 \cdot 380.5426 - 56.2933^2} = \frac{742.5703}{636.4906} = 1.1667, \\ a_1 &= \frac{10 \cdot 485.9487 - 56.2933 \cdot 73.8373}{10 \cdot 380.5426 - 56.2933^2} = \frac{702.9438}{636.4906} = 1.1044. \end{aligned}$$

We conclude that the linear function that best fits this data in the least-squares sense is

$$y = 1.1044x + 1.1667.$$

The data, and this function, are shown in Figure 6.1. □

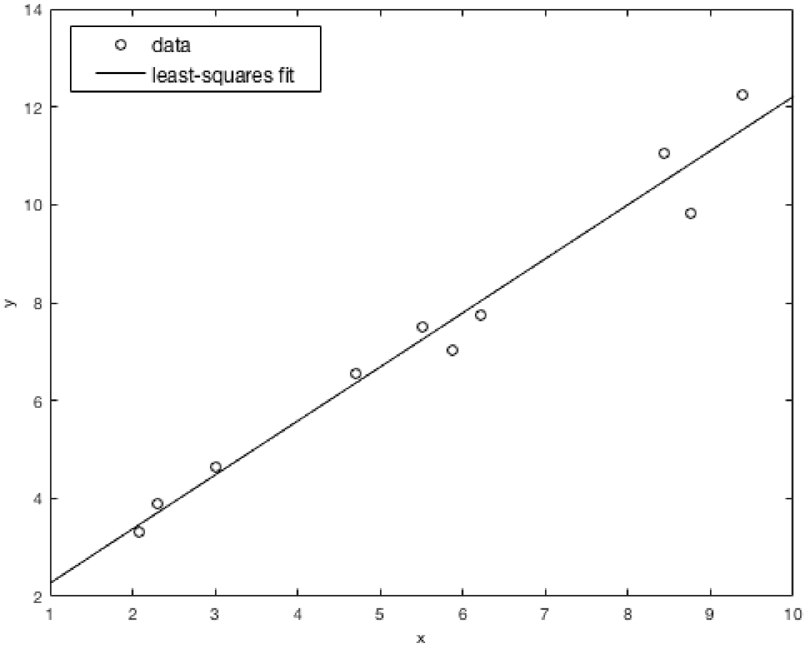


Figure 6.1: Data points (x_i, y_i) (circles) and least-squares line (solid line)

Exercise 6.1.1 *Write a* MATLAB *function* `[m,b]=leastsqline(x,y)` *that computes the slope* $a_1 =$ `m` *and* y*-intercept* $a_0 =$ `b` *of the line* $y = a_1x + a_0$ *that best fits the data* (x_i, y_i)*,* $i = 1, 2, \ldots, m$ *where* $m =$ `length`(x)*, in the least-squares sense.*

Exercise 6.1.2 *Generalize the above derivation of the coefficients* a_0 *and* a_1 *of the least-squares line to obtain formulas for the coefficients* a*,* b *and* c *of the quadratic function* $y = ax^2 + bx + c$ *that best fits the data* (x_i, y_i)*,* $i = 1, 2, \ldots, m$*, in the least-squares sense. Then generalize your function* `leastsqline` *from Exercise 6.1.1 to obtain a new function* `leastsqquad` *that computes these coefficients.*

It is interesting to note that if we define the $m \times 2$ matrix A, the 2-vector $\mathbf{a}$, and the m-vector $\mathbf{y}$ by

$$A = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix}, \quad \mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix},$$

then $\mathbf{a}$ is the solution to the system of equations

$$A^TA\mathbf{a} = A^T\mathbf{y}.$$

These equations are the normal equations defined earlier, written in matrix-vector form. They arise from the problem of finding the vector $\mathbf{a}$ such that

$$\|A\mathbf{a} - \mathbf{y}\|_2$$

is minimized, where, for any vector $\mathbf{u}$, $|\mathbf{u}|$ is the magnitude, or length, of $\mathbf{u}$.

In this case, this expression is equivalent to the square root of the expression we originally intended to minimize,

$$\sum_{i=1}^{m}(a_1x_i + a_0 - y_i)^2,$$

but the normal equations also characterize the solution $\mathbf{a}$, an n-vector, to the more general linear least squares problem of minimizing $|A\mathbf{a} - \mathbf{y}|$ for any matrix A that is $m \times n$, where $m \geq n$, whose columns are linearly independent.

We now consider the problem of finding a polynomial of degree n that gives the best least-squares fit. As before, let (x_1, y_1), (x_2, y_2), …, (x_m, y_m) be given data points that need to be approximated by a polynomial of degree n. We assume that $n < m - 1$, for otherwise, we can use polynomial interpolation to fit the points exactly.

Let the least-squares polynomial have the form

$$p_n(x) = \sum_{j=0}^{n} a_j x^j.$$

Our goal is to minimize the sum of squares of the deviations in $p_n(x)$ from each y-value,

$$E(\mathbf{a}) = \sum_{i=1}^{m}[p_n(x_i) - y_i]^2 = \sum_{i=1}^{m}\left[\sum_{j=0}^{n} a_j x_i^j - y_i\right]^2,$$

where $\mathbf{a}$ is a column vector of the unknown coefficients of $p_n(x)$,

$$\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}.$$

Differentiating this function with respect to each a_k yields

$$\frac{\partial E}{\partial a_k} = \sum_{i=1}^{m} 2\left[\sum_{j=0}^{n} a_j x_i^j - y_i\right] x_i^k, \quad k = 0, 1, \ldots, n.$$

Setting each of these partial derivatives equal to zero yields the system of equations

$$\sum_{j=0}^{n}\left(\sum_{i=1}^{m} x_i^{j+k}\right) a_j = \sum_{i=1}^{m} x_i^k y_i, \quad k = 0, 1, \ldots, n.$$

These are the **normal equations**. They are a generalization of the normal equations previously defined for the linear case, where $n = 1$. Solving this system yields the coefficients $\{a_j\}_{j=0}^n$ of the least-squares polynomial $p_n(x)$.

As in the linear case, the normal equations can be written in matrix-vector form

$$A^TA\mathbf{a} = A^T\mathbf{y},$$

where

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix}, \quad \mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}. \tag{6.1}$$

The matrix A is called a *Vandermonde matrix* for the points $x_0, x_1, \ldots, x_m$.

The normal equations equations can be used to compute the coefficients of *any* linear combination of functions $\{\phi_j(x)\}_{j=0}^n$ that best fits data in the least-squares sense, provided that these functions are linearly independent. In this general case, the entries of the matrix A are given by $a_{ij} = \phi_i(x_j)$, for $i = 1, 2, \ldots, m$ and $j = 0, 1, \ldots, n$.

Example 6.1.2 *We wish to find the quadratic function $y = a_2x^2 + a_1x + a_0$ that best approximates the data shown in Table 6.2, in the least-squares sense. By defining*

Table 6.2: Data points (x_i, y_i), for $i = 1, 2, \ldots, 10$, to be fit by a quadratic function

i	x_i	y_i
1	2.0774	2.7212
2	2.3049	3.7798
3	3.0125	4.8774
4	4.7092	6.6596
5	5.5016	10.5966
6	5.8704	9.8786
7	6.2248	10.5232
8	8.4431	23.3574
9	8.7594	24.0510
10	9.3900	27.4827

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_{10} & x_{10}^2 \end{bmatrix}, \quad \mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{10} \end{bmatrix},$$

and solving the normal equations

$$A^TA\mathbf{a} = A^T\mathbf{y},$$

we obtain the coefficients

$$c_0 = 4.7681, \quad c_1 = -1.5193, \quad c_2 = 0.4251,$$

and conclude that the quadratic function that best fits this data in the least-squares sense is

$$y = 0.4251x^2 - 1.5193x + 4.7681.$$

The data, and this function, are shown in Figure 6.2. □

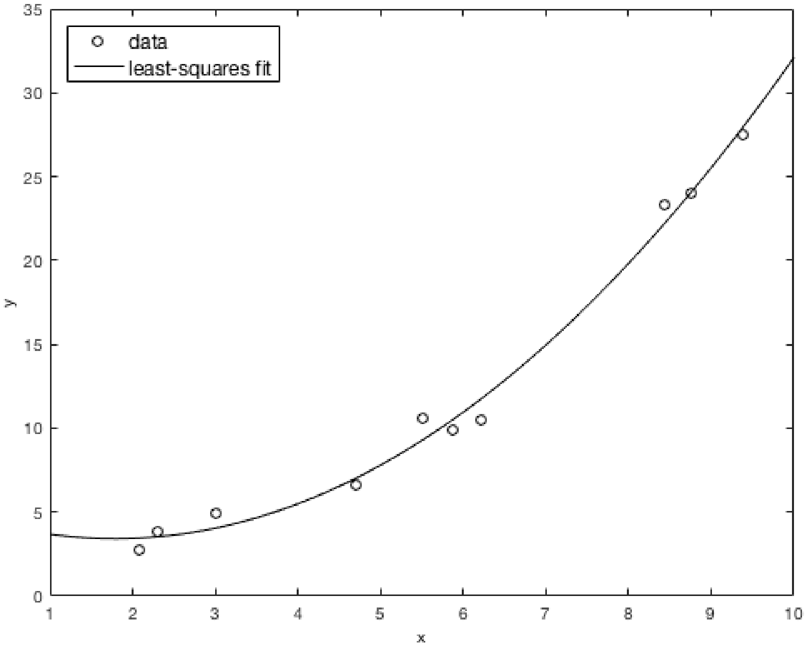


Figure 6.2: Data points (x_i, y_i) (circles) and quadratic least-squares fit (solid curve)

Exercise 6.1.3 *Write a* MATLAB *function* `a=leastsqpoly(x,y,n)` *that computes the coefficients* a_j*,* $j = 0, 1, \ldots, n$ *of the polynomial of degree* n *that best fits the data* (x_i, y_i) *in the least-squares sense. Use the* `vander` *function to easily construct the Vandermonde matrix* A *used in the normal equations. Make sure you solve the normal equations* without *explicitly computing* A^TA*. Test your function on the data from Example 6.1.2, with different values of* n*, but with* $n < 10$*. How does the residual* $\|A\mathbf{a} - bfy\|_2$ *behave as* n *increases?*

Exercise 6.1.4 *Test your function* `leastsqpoly` *from Exercise 6.1.3 to approximate the function* $y = e^{-cx}$ *on the interval* $[0, 1]$ *where* c *is a chosen positive constant. Experiment with different values of* c*, as well as* m *and* n*, the number of data points and degree of the approximating polynomial, respectively. What combination yields the smallest relative residual* $\|A\mathbf{a} - \mathbf{y}\|_2/\|\mathbf{y}\|_2$*?*

Least-squares fitting can also be used to fit data with functions that are not linear combinations of functions such as polynomials. Suppose we believe that given data points can best be matched to an exponential function of the form $y = be^{ax}$, where the constants a and b are unknown. Taking the natural logarithm of both sides of this equation yields

$$\ln y = \ln b + ax.$$

If we define $z = \ln y$ and $c = \ln b$, then the problem of fitting the original data points $\{(x_i, y_i)\}_{i=1}^m$

with an exponential function is transformed into the problem of fitting the data points $\{(x_i, z_i)\}_{i=1}^m$ with a linear function of the form $c + ax$, for unknown constants a and c.

Similarly, suppose the given data is believed to approximately conform to a function of the form $y = bx^a$, where the constants a and b are unknown. Taking the natural logarithm of both sides of this equation yields

$$\ln y = \ln b + a \ln x.$$

If we define $z = \ln y$, $c = \ln b$ and $w = \ln x$, then the problem of fitting the original data points $\{(x_i, y_i)\}_{i=1}^m$ with a constant times a power of x is transformed into the problem of fitting the data points $\{(w_i, z_i)\}_{i=1}^m$ with a linear function of the form $c + aw$, for unknown constants a and c.

Example 6.1.3 *We wish to find the exponential function $y = be^{ax}$ that best approximates the data shown in Table 6.3, in the least-squares sense. By defining*

Table 6.3: Data points (x_i, y_i), for $i = 1, 2, \ldots, 5$, to be fit by an exponential function

i	x_i	y_i
1	2.0774	1.4509
2	2.3049	2.8462
3	3.0125	2.1536
4	4.7092	4.7438
5	5.5016	7.7260

$$A = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots & \vdots \\ 1 & x_5 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c \\ a \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_5 \end{bmatrix},$$

where $c = \ln b$ and $z_i = \ln y_i$ for $i = 1, 2, \ldots, 5$, and solving the normal equations

$$A^T A\mathbf{c} = A^T \mathbf{z},$$

we obtain the coefficients

$$a = 0.4040, \quad b = e^c = e^{-0.2652} = 0.7670,$$

and conclude that the exponential function that best fits this data in the least-squares sense is

$$y = 0.7670e^{0.4040x}.$$

□

Exercise 6.1.5 *Write a* MATLAB *function* `[a,b]=leastsqexp(x,y)` *that computes the coefficients a and b of a function $y = be^{ax}$ that fits the given data (x_i, y_i), $i = 1, 2, \ldots, m$ where $m =$* `length(x)`*, in the least squares sense.*

Exercise 6.1.6 *Write a* MATLAB *function* `[a,b]=leastsqpower(x,y)` *that computes the coefficients a and b of a function $y = bx^a$ that fits the given data (x_i, y_i), $i = 1, 2, \ldots, m$ where $m =$* `length(x)`*, in the least squares sense.*

6.2 Continuous Least Squares Approximation

Now, suppose we have a *continuous* set of data. That is, we have a function $f(x)$ defined on an interval $[a, b]$, and we wish to approximate it as closely as possible, in some sense, by a function $f_n(x)$ that is a linear combination of given real-valued functions $\{\phi_j(x)\}_{j=0}^n$. If we choose m equally spaced points $\{x_i\}_{i=1}^m$ in $[a, b]$, and let $m \to \infty$, we obtain the **continuous least-squares problem** of finding the function

$$f_n(x) = \sum_{j=0}^n c_j \phi_j(x)$$

that minimizes

$$E(c_0, c_1, \ldots, c_n) = \|f_n - f\|_2^2 = \int_a^b [f_n(x) - f(x)]^2 \, dx = \int_a^b \left[\sum_{j=0}^n c_j \phi_j(x) - f(x) \right]^2 dx,$$

where

$$\|f_n - f\|_2 = \left(\int_a^b [f_n(x) - f(x)]^2 \, dx \right)^{1/2}.$$

We refer to f_n as the **best approximation in span**$(\phi_0, \phi_1, \ldots, \phi_n)$ **to** f **in the 2-norm on** (a, b).

This minimization can be performed for $f \in C[a, b]$, the space of functions that are continuous on $[a, b]$, but it is not necessary for a function $f(x)$ to be continuous for $\|f\|_2$ to be defined. Rather, we consider the space $L^2(a, b)$, the space of real-valued functions such that $|f(x)|^2$ is *integrable* over (a, b). Both of these spaces, in addition to being *normed* spaces, are also *inner product spaces*, as they are equipped with an inner product

$$\langle f, g \rangle = \int_a^b f(x) g(x) \, dx.$$

Such spaces are reviewed in Section B.13.

To obtain the coefficients $\{c_j\}_{j=0}^n$, we can proceed as in the discrete case. We compute the partial derivatives of $E(c_0, c_1, \ldots, c_n)$ with respect to each c_k and obtain

$$\frac{\partial E}{\partial c_k} = \int_a^b \phi_k(x) \left[\sum_{j=0}^n c_j \phi_j(x) - f(x) \right] dx,$$

and requiring that each partial derivative be equal to zero yields the **normal equations**

$$\sum_{j=0}^n \left[\int_a^b \phi_k(x) \phi_j(x) \, dx \right] c_j = \int_a^b \phi_k(x) f(x) \, dx, \quad k = 0, 1, \ldots, n.$$

We can then solve this system of equations to obtain the coefficients $\{c_j\}_{j=0}^n$. This system can be solved as long as the functions $\{\phi_j(x)\}_{j=0}^n$ are *linearly independent*. That is, the condition

$$\sum_{j=0}^n c_j \phi_j(x) \equiv 0, \quad x \in [a, b],$$

is only true if $c_0 = c_1 = \cdots = c_n = 0$. In particular, this is the case if, for $j = 0, 1, \ldots, n$, $\phi_j(x)$ is a polynomial of degree j.

Exercise 6.2.1 *Prove that the functions $\{\phi_j(x)\}_{j=0}^n$ are linearly independent in $C[a,b]$ if, for $j = 0, 1, \ldots, n$, $\phi_j(x)$ is a polynomial of degree j.*

Exercise 6.2.2 *Let A be the $(n+1) \times (n+1)$ matrix defined by*

$$a_{ij} = \int_a^b \phi_i(x)\phi_j(x)\,dx,$$

where the functions $\{\phi_j(x)\}_{j=0}^n$ are real-valued functions that are linearly independent in $C[a,b]$. Prove that A is symmetric positive definite. Why is the assumption of linear independence essential? How does this guarantee that the solution of the normal equations yields a minimum rather than a maximum or saddle point?

Example 6.2.1 *We approximate $f(x) = e^x$ on the interval $[0,5]$ by a fourth-degree polynomial*

$$f_4(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + c_4x^4.$$

The normal equations have the form

$$\sum_{j=0}^n a_{ij}c_j = b_i, \quad i = 0, 1, \ldots, 4,$$

or, in matrix-vector form, $A\mathbf{c} = \mathbf{b}$, where

$$a_{ij} = \int_0^5 x^i x^j\,dx = \int_0^5 x^{i+j}\,dx = \frac{5^{i+j+1}}{i+j+1}, \quad i, j = 0, 1, \ldots, 4,$$

$$b_i = \int_0^5 x^i e^x\,dx, \quad i = 0, 1, \ldots, 4.$$

Integration by parts yields the relation

$$b_i = 5^i e^5 - ib_{i-1}, \quad b_0 = e^5 - 1.$$

Solving this system of equations yields the polynomial

$$f_4(x) = 2.3002 - 6.226x + 9.5487x^2 - 3.86x^3 + 0.6704x^4.$$

As Figure 6.3 shows, this polynomial is barely distinguishable from e^x on $[0,5]$.

However, it should be noted that the matrix A is closely related to the $n \times n$ **Hilbert matrix** *H_n, which has entries*

$$[H_n]_{ij} = \frac{1}{i+j-1}, \quad 1 \leq i, j \leq n.$$

This matrix is famous for being highly ill-conditioned, *meaning that solutions to systems of linear equations involving this matrix that are computed using floating-point arithmetic are highly sensitive to roundoff error. In fact, the matrix A in this example has a* condition number *of* 1.56×10^7, *which means that a change of size ϵ in the right-hand side vector $\mathbf{b}$, with entries b_i, can cause a change of size $1.56\epsilon \times 10^7$ in the solution $\mathbf{c}$.* □

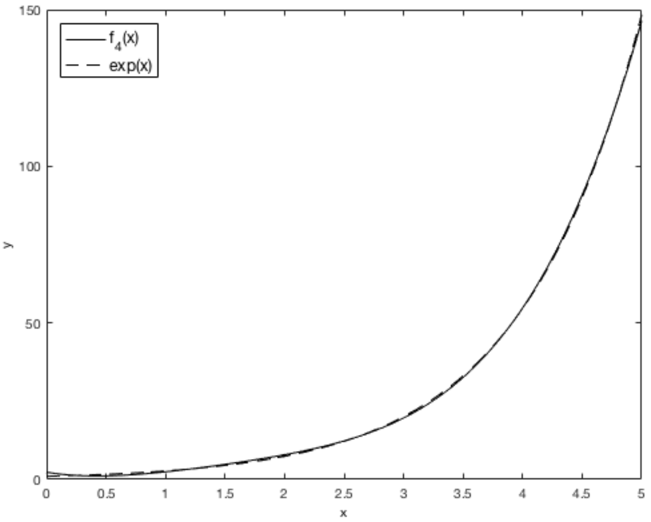


Figure 6.3: Graphs of $f(x) = e^x$ (red dashed curve) and 4th-degree continuous least-squares polynomial approximation $f_4(x)$ on $[0, 5]$ (blue solid curve)

Exercise 6.2.3 *Repeat Example 6.2.1 with* $f(x) = x^7$. *What happens to the coefficients* $\{c_j\}_{j=0}^4$ *if the right-hand side vector* $\mathbf{b}$ *is perturbed?*

For the remainder of this section, we restrict ourselves to the case where the functions $\{\phi_j(x)\}_{j=0}^n$ are polynomials. These polynomials form a *basis* of $\mathcal{P}_n$, the vector space of polynomials of degree at most n. Then, for $f \in L^2(a, b)$, we refer to the polynomial f_n that minimizes $\|f - p\|_2$ over all $p \in \mathcal{P}_n$ as the **best 2-norm approximating polynomial**, or **least-squares approximating polynomial, of degree n to f on** (a, b).

6.2.1 Orthogonal Polynomials

As the preceding example shows, it is important to choose the polynomials $\{\phi_j(x)\}_{j=0}^n$ wisely, so that the resulting system of normal equations is not unduly sensitive to round-off errors. An even better choice is one for which this system can be solved analytically, with relatively few computations. An ideal choice of polynomials is one for which the task of computing $f_{n+1}(x)$ can reuse the computations needed to compute $f_n(x)$.

Suppose that we can construct a set of polynomials $\{\phi_j(x)\}_{j=0}^n$ that is **orthogonal** with respect to the inner product of functions on (a, b). That is,

$$\langle \phi_k, \phi_j \rangle = \int_a^b \phi_k(x)\phi_j(x)\,dx = \left\{ \begin{array}{ll} 0 & k \neq j \\ \alpha_k > 0 & k = j \end{array} \right. .$$

Then, the normal equations simplify to a trivial system

$$\left[\int_a^b [\phi_k(x)]^2\,dx \right] c_k = \int_a^b \phi_k(x) f(x)\,dx, \quad k = 0, 1, \ldots, n,$$

or, in terms of norms and inner products,

$$\|\phi_k\|_2^2 c_k = \langle \phi_k, f\rangle, \quad k = 0, 1, \ldots, n.$$

It follows that the coefficients $\{c_j\}_{j=0}^n$ of the least-squares approximation $f_n(x)$ are simply

$$c_k = \frac{\langle \phi_k, f\rangle}{\|\phi_k\|_2^2}, \quad k = 0, 1, \ldots, n.$$

If the constants $\{\alpha_k\}_{k=0}^n$ above satisfy $\alpha_k = 1$ for $k = 0, 1, \ldots, n$, then we say that the orthogonal set of functions $\{\phi_j(x)\}_{j=0}^n$ is **orthonormal**. In that case, the solution to the continuous least-squares problem is simply given by

$$c_k = \langle \phi_k, f\rangle, \quad k = 0, 1, \ldots, n. \tag{6.2}$$

Next, we will learn how sets of orthogonal polynomials can be constructed.

6.2.2 Construction of Orthogonal Polynomials

Recall the process known as *Gram-Schmidt orthogonalization* for obtaining a set of orthogonal vectors $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n$ from a set of linearly independent vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$:

$$\begin{aligned} \mathbf{p}_1 &= \mathbf{a}_1 \\ \mathbf{p}_2 &= \mathbf{a}_2 - \frac{\mathbf{p}_1 \cdot \mathbf{a}_2}{\mathbf{p}_1 \cdot \mathbf{p}_1}\mathbf{p}_1 \\ &\vdots \\ \mathbf{p}_n &= \mathbf{a}_n - \sum_{j=0}^{n-1} \frac{\mathbf{p}_j \cdot \mathbf{a}_n}{\mathbf{p}_j \cdot \mathbf{p}_j}\mathbf{p}_j. \end{aligned}$$

By *normalizing* each vector $\mathbf{p}_j$, we obtain a unit vector

$$\mathbf{q}_j = \frac{1}{|\mathbf{p}_j|}\mathbf{p}_j,$$

and a set of *orthonormal* vectors $\{\mathbf{q}_j\}_{j=1}^n$, in that they are orthogonal ($\mathbf{q}_k \cdot \mathbf{q}_j = 0$ for $k \neq j$), and unit vectors ($\mathbf{q}_j \cdot \mathbf{q}_j = 1$).

We can use a similar process to compute a set of orthogonal polynomials $\{p_j(x)\}_{j=0}^n$. For convenience, we will require that all polynomials in the set be *monic*; that is, their leading (highest-degree) coefficient must be equal 1. We then define $p_0(x) = 1$. Then, because $p_1(x)$ is supposed to be of degree 1, it must have the form $p_1(x) = x - \alpha_1$ for some constant α_1. To ensure that $p_1(x)$ is orthogonal to $p_0(x)$, we compute their inner product, and obtain

$$0 = \langle p_0, p_1\rangle = \langle 1, x - \alpha_1\rangle,$$

so we must have

$$\alpha_1 = \frac{\langle 1, x\rangle}{\langle 1, 1\rangle}.$$

For $j > 1$, we start by setting $p_j(x) = xp_{j-1}(x)$, since p_j should be of degree one greater than that of p_{j-1}, and this satisfies the requirement that p_j be monic. Then, we need to subtract polynomials of lower degree to ensure that p_j is orthogonal to p_i, for $i < j$. To that end, we apply Gram-Schmidt orthogonalization and obtain

$$p_j(x) = xp_{j-1}(x) - \sum_{i=0}^{j-1} \frac{\langle p_i, xp_{j-1}\rangle}{\langle p_i, p_i\rangle} p_i(x).$$

However, by the definition of the inner product, $\langle p_i, xp_{j-1}\rangle = \langle xp_i, p_{j-1}\rangle$. Furthermore, because xp_i is of degree $i+1$, and p_{j-1} is orthogonal to *all* polynomials of degree less than j, it follows that $\langle p_i, xp_{j-1}\rangle = 0$ whenever $i < j-1$.

We have shown that sequences of orthogonal polynomials satisfy a **three-term recurrence relation**

$$p_j(x) = (x - \alpha_j)p_{j-1}(x) - \beta_{j-1}^2 p_{j-2}(x), \quad j > 1,$$

where the **recursion coefficients** α_j and β_{j-1}^2 are defined to be

$$\alpha_j = \frac{\langle p_{j-1}, xp_{j-1}\rangle}{\langle p_{j-1}, p_{j-1}\rangle}, \quad j > 1,$$

$$\beta_j^2 = \frac{\langle p_{j-1}, xp_j\rangle}{\langle p_{j-1}, p_{j-1}\rangle} = \frac{\langle xp_{j-1}, p_j\rangle}{\langle p_{j-1}, p_{j-1}\rangle} = \frac{\langle p_j, p_j\rangle}{\langle p_{j-1}, p_{j-1}\rangle} = \frac{\|p_j\|_2^2}{\|p_{j-1}\|_2^2}, \quad j \geq 1.$$

Note that $\langle xp_{j-1}, p_j\rangle = \langle p_j, p_j\rangle$ because xp_{j-1} differs from p_j by a polynomial of degree at most $j-1$, which is orthogonal to p_j. The recurrence relation is also valid for $j = 1$, provided that we define $p_{j-1}(x) \equiv 0$, and α_1 is defined as above. That is,

$$p_1(x) = (x - \alpha_1)p_0(x), \quad \alpha_1 = \frac{\langle p_0, xp_0\rangle}{\langle p_0, p_0\rangle}.$$

If we also define the recursion coefficient β_0 by

$$\beta_0^2 = \langle p_0, p_0\rangle,$$

and then define

$$q_j(x) = \frac{p_j(x)}{\beta_0\beta_1\cdots\beta_j},$$

then the polynomials $q_0, q_1, \ldots, q_n$ are also orthogonal, and

$$\langle q_j, q_j\rangle = \frac{\langle p_j, p_j\rangle}{\beta_0^2\beta_1^2\cdots\beta_j^2} = \langle p_j, p_j\rangle \frac{\langle p_{j-1}, p_{j-1}\rangle}{\langle p_j, p_j\rangle} \cdots \frac{\langle p_0, p_0\rangle}{\langle p_1, p_1\rangle} \frac{1}{\langle p_0, p_0\rangle} = 1.$$

That is, these polynomials are *orthonormal.*

Exercise 6.2.4 *Compute the first three monic orthogonal polynomials with respect to the inner product*

$$\langle f, g\rangle = \int_0^1 f(x)g(x)w(x)\,dx,$$

with weight functions $w(x) = 1$ and $w(x) = x$.

Exercise 6.2.5 *Write a* MATLAB *function* `P=orthpoly(a,b,w,n)` *that computes the coefficients of monic orthogonal polynomials on the interval* (a, b)*, up to and including degree* n*, and stores their coefficients in the rows of the* $(n + 1) \times (n + 1)$ *matrix* P*. The vector* `w` *stores the coefficients of a polynomial* $w(x)$ *that serves as the weight function. Use* MATLAB*'s polynomial functions to evaluate the required inner products. How can you ensure that the weight function does not change sign on* (a, b)*?*

6.2.3 Legendre Polynomials

If we consider the inner product

$$\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)\,dx,$$

then by Gram-Schmidt Orthogonalization, a sequence of orthogonal polynomials, with respect to this inner product, can be defined as follows:

$$L_0(x) = 1, \tag{6.3}$$

$$L_1(x) = x, \tag{6.4}$$

$$L_{j+1}(x) = \frac{2j+1}{j+1} x L_j(x) - \frac{j}{j+1} L_{j-1}(x), \quad j = 1, 2, \ldots \tag{6.5}$$

These are known as the **Legendre polynomials** [23]. One of their most important applications is in the construction of Gaussian quadrature rules (see Section 7.5). Specifically, the roots of $L_n(x)$, for $n \geq 1$, are the nodes of a Gaussian quadrature rule for the interval $(-1, 1)$. However, they can also be used to easily compute continuous least-squares polynomial approximations, as the following example shows.

Example 6.2.2 *We will use Legendre polynomials to approximate* $f(x) = \cos x$ *on* $[-\pi/2, \pi/2]$ *by a quadratic polynomial. First, we note that the first three Legendre polynomials, which are the ones of degree 0, 1 and 2, are*

$$L_0(x) = 1, \quad L_1(x) = x, \quad L_2(x) = \frac{1}{2}(3x^2 - 1).$$

However, it is not practical to use these polynomials directly to approximate $f(x)$*, because they are orthogonal with respect to the inner product defined on the interval* $(-1, 1)$*, and we wish to approximate* $f(x)$ *on* $(-\pi/2, \pi/2)$*.*

To obtain orthogonal polynomials on $(-\pi/2, \pi/2)$*, we replace* x *by* $2t/\pi$*, where* t *belongs to* $[-\pi/2, \pi/2]$*, in the Legendre polynomials, which yields*

$$\tilde{L}_0(t) = 1, \quad \tilde{L}_1(t) = \frac{2t}{\pi}, \quad \tilde{L}_2(t) = \frac{1}{2}\left(\frac{12}{\pi^2}t^2 - 1\right).$$

Then, we can express our quadratic approximation $f_2(x)$ *of* $f(x)$ *by the linear combination*

$$f_2(x) = c_0\tilde{L}_0(x) + c_1\tilde{L}_1(x) + c_2\tilde{L}_2(x),$$

where

$$c_j = \frac{\langle f, \tilde{L}_j\rangle}{\langle \tilde{L}_j, \tilde{L}_j\rangle}, \quad j = 0, 1, 2.$$

Computing these inner products yields

$$\begin{aligned}
\langle f, \tilde{L}_0 \rangle &= \int_{-\pi/2}^{\pi/2} \cos t\, dt \\
&= 2, \\
\langle f, \tilde{L}_1 \rangle &= \int_{-\pi/2}^{\pi/2} \frac{2t}{\pi} \cos t\, dt \\
&= 0, \\
\langle f, \tilde{L}_2 \rangle &= \int_{-\pi/2}^{\pi/2} \frac{1}{2}\left(\frac{12}{\pi^2}t^2 - 1\right) \cos t\, dt \\
&= \frac{2}{\pi^2}(\pi^2 - 12), \\
\langle \tilde{L}_0, \tilde{L}_0 \rangle &= \int_{-\pi/2}^{\pi/2} 1\, dt \\
&= \pi, \\
\langle \tilde{L}_1, \tilde{L}_1 \rangle &= \int_{-\pi/2}^{\pi/2} \left(\frac{2t}{\pi}\right)^2 dt \\
&= \frac{8\pi}{3}, \\
\langle \tilde{L}_2, \tilde{L}_2 \rangle &= \int_{-\pi/2}^{\pi/2} \left[\frac{1}{2}\left(\frac{12}{\pi^2}t^2 - 1\right)\right]^2 dt \\
&= \frac{\pi}{5}.
\end{aligned}$$

It follows that

$$c_0 = \frac{2}{\pi}, \quad c_1 = 0, \quad c_2 = \frac{2}{\pi^2}\frac{5}{\pi}(\pi^2 - 12) = \frac{10}{\pi^3}(\pi^2 - 12),$$

and therefore

$$f_2(x) = \frac{2}{\pi} + \frac{5}{\pi^3}(\pi^2 - 12)\left(\frac{12}{\pi^2}x^2 - 1\right) \approx 0.98016 - 0.4177x^2.$$

This approximation is shown in Figure 6.4. □

Exercise 6.2.6 *Write a* MATLAB *script that computes the coefficients of the Legendre polynomials up to a given degree n, using the recurrence relation (6.5) and the function* `conv` *for multiplying polynomials. Then, plot the graphs of these polynomials on the interval $(-1, 1)$. What properties can you observe in these graphs? Is there any symmetry to them?*

Exercise 6.2.7 *Prove that the Legendre polynomial $L_j(x)$ is an odd function if j is odd, and an even function if j is even.* Hint: *use mathematical induction.*

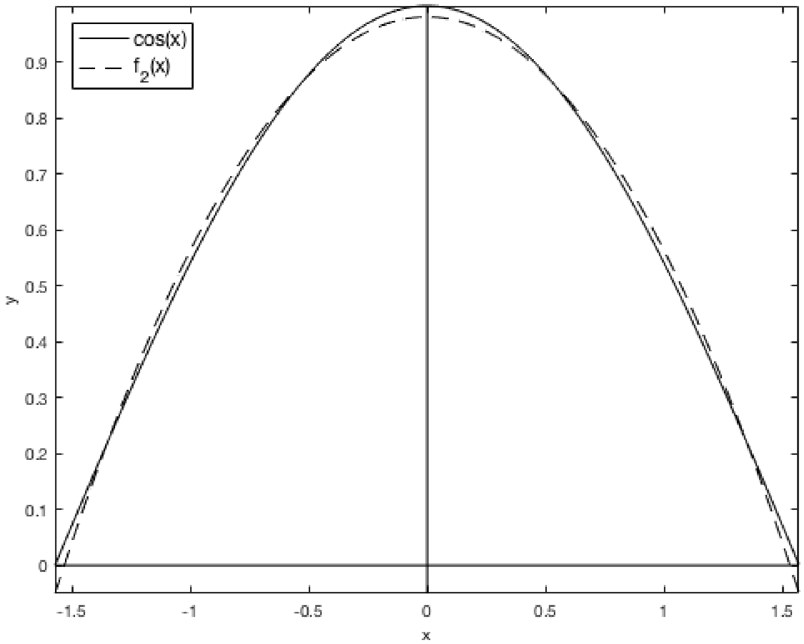


Figure 6.4: Graph of $\cos x$ (solid blue curve) and its continuous least-squares quadratic approximation (red dashed curve) on $(-\pi/2, \pi/2)$

Exercise 6.2.8 *Let A be the Vandermonde matrix from (6.1), where the points $x_1, x_2, \ldots, x_m$ are equally spaced points in the interval $(-1, 1)$. Construct this matrix in* MATLAB *for a small chosen value of n and a large value of m, and then compute the QR factorization of A (See Chapter 6). How do the columns of Q relate to the Legendre polynomials?*

6.2.4 Chebyshev Polynomials

It is possible to compute sequences of orthogonal polynomials with respect to other inner products. A generalization of the inner product that we have been using is defined by

$$\langle f, g \rangle = \int_a^b f(x)g(x)w(x)\,dx,$$

where $w(x)$ is a **weight function**. To be a weight function, it is required that $w(x) \geq 0$ on (a, b), and that $w(x) \neq 0$ on any subinterval of (a, b). So far, we have only considered the case of $w(x) \equiv 1$.

Exercise 6.2.9 *Prove that the discussion of Section 6.2.2 also applies when using the inner product*

$$\langle f, g\rangle = \int_a^b f(x)g(x)w(x)\,dx,$$

where $w(x)$ is a weight function *that satisfies $w(x) \geq 0$ on (a, b). That is, polynomials orthogonal with respect to this inner product also satisfy a three-term recurrence relation, with analogous definitions of the recursion coefficients α_j and β_j.*

Another weight function of interest is

$$w(x) = \frac{1}{\sqrt{1-x^2}}, \quad -1 < x < 1.$$

A sequence of polynomials that is orthogonal with respect to this weight function, and the associated inner product

$$\langle f, g\rangle = \int_{-1}^1 f(x)g(x)\frac{1}{\sqrt{1-x^2}}\,dx$$

is the sequence of *Chebyshev polynomials*, previously introduced in Section 5.4.2:

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_{j+1}(x) &= 2xT_j(x) - T_{j-1}(x), \quad j = 1, 2, \ldots \end{aligned}$$

which can also be defined by

$$T_j(x) = \cos(j\cos^{-1}x), \quad -1 \leq x \leq 1.$$

It is interesting to note that if we let $x = \cos\theta$, then

$$\begin{aligned} \langle f, T_j\rangle &= \int_{-1}^1 f(x)\cos(j\cos^{-1}x)\frac{1}{\sqrt{1-x^2}}\,dx \\ &= \int_0^\pi f(\cos\theta)\cos j\theta\,d\theta. \end{aligned}$$

In Section 6.4, we will investigate continuous and discrete least-squares approximation of functions by linear combinations of *trigonometric polynomials* such as $\cos j\theta$ or $\sin j\theta$, which will reveal how these coefficients $\langle f, T_j\rangle$ can be computed very rapidly.

Exercise 6.2.10 *Write a* MATLAB *function* `fn=best2normapprox(f,a,b,n,w)` *that computes the coefficients of $f_n(x)$, the best 2-norm approximating polynomial of degree n to the given function $f(x)$ on (a, b), with respect to the inner product*

$$\langle f, g\rangle = \int_a^b f(x)g(x)w(x)\,dx,$$

where `w` *is a function handle for the weight function $w(x)$. Use the built-in function* `integral` *to evaluate the required inner products. Make the fifth argument* `w` *an optional argument, using $w(x) \equiv 1$ as a default.*

Exercise 6.2.11 *Compute the best 2-norm approximating polynomial of degree 3 to the functions* $f(x) = e^x$ *and* $g(x) = \sin \pi x$ *on* $(-1, 1)$*, using both Legendre and Chebyshev polynomials. Comment on the accuracy of these approximations.*

6.2.5 Error Analysis

Let $p \in \mathcal{P}_n$, where $\mathcal{P}_n$ is the space of polynomials of degree at most n, and let f_n be the best 2-norm approximating polynomial of degree n to $f \in L^2(a, b)$. As before, we assume the polynomials $q_0(x), q_1(x), \ldots, q_n(x)$ are *orthonormal*, in the sense that

$$\langle q_j, q_k \rangle = \int_a^b q_j(x) q_k(x) w(x)\, dx = \delta_{jk}, \quad j, k = 0, 1, \ldots, n.$$

Then, from (6.2) we have

$$f_n(x) = \sum_{j=0}^{n} \langle q_j, f \rangle q_j(x). \tag{6.6}$$

This form of $f_n(x)$ can be used to prove the following result.

Theorem 6.2.3 *The polynomial* $f_n \in \mathcal{P}_n$ *is the best 2-norm approximating polynomial of degree* n *to* $f \in L^2(a, b)$ *if and only if*

$$\langle f - f_n, p \rangle = 0$$

for all $p \in \mathcal{P}_n$*.*

Exercise 6.2.12 *Use (6.6) to prove one part of Theorem 6.2.3: assume* f_n *is the best 2-norm approximating polynomial to* $f \in L^2(a, b)$*, and show that* $\langle f - f_n, p \rangle = 0$ *for any* $p \in \mathcal{P}_n$*.*

Exercise 6.2.13 *Use the Cauchy-Schwarz inequality to prove the converse of Exercise 6.2.12: that if* $f \in L^2(a, b)$ *and* $\langle f - f_n, p \rangle = 0$ *for an arbitrary* $p \in \mathcal{P}_n$*, then* f_n *is the best 2-norm approximating polynomial of degree* n *to* f*; that is,*

$$\|f - f_n\|_2 \leq \|f - p\|_2.$$

Hint: *by the assumptions,* $f - f_n$ *is orthogonal to* any *polynomial in* $\mathcal{P}_n$*.*

6.2.6 Roots of Orthogonal Polynomials

Finally, we prove one property of orthogonal polynomials that will prove useful in our upcoming discussion of the role of orthogonal polynomials in numerical integration. Let $\varphi_j(x)$ be a polynomial of degree $j \geq 1$ that is orthogonal to all polynomials of lower degree, with respect to the inner product

$$\langle f, g \rangle = \int_a^b f(x) g(x) w(x)\, dx,$$

and let the points $\xi_1, \xi_2, \ldots, \xi_k$ be the points in (a, b) at which $\varphi_j(x)$ changes sign. This set of points cannot be empty, because φ_j, being a polynomial of degree at least one, is orthogonal a constant function, which means

$$\int_a^b \varphi_j(x)w(x)\,dx = 0.$$

Because $w(x)$ is a weight function, it does not change sign. Therefore, in order for the integral to be zero, $\varphi_j(x)$ must change sign at least once in (a, b).

If we define

$$\pi_k(x) = (x - \xi_1)(x - \xi_2)\cdots(x - \xi_k),$$

then $\varphi_j(x)\pi_k(x)$ does not change sign on (a, b), because π_k changes sign at exactly the same points in (a, b) as φ_j. Because both polynomials are also nonzero on (a, b), we must have

$$\langle \varphi_j, \pi_k \rangle = \int_a^b \varphi_j(x)\pi_k(x)w(x)\,dx \neq 0.$$

If $k < j$, then we have a contradiction, because φ_j is orthogonal to *any* polynomial of lesser degree. Therefore, $k \geq j$. However, if $k > j$, we also have a contradiction, because a polynomial of degree j cannot change sign more than j times on the entire real number line, let alone an interval. We conclude that $k = j$, which implies that all of the roots of φ_j are real and distinct, and lie in (a, b).

Exercise 6.2.14 *Use your function* `orthpoly` *from Exercise 6.2.5 to generate orthogonal polynomials of a fixed degree n for various weight functions. How does the distribution of the roots of $p_n(x)$ vary based on where the weight function has smaller or larger values?* Hint: *consider the distribution of the roots of Chebyshev polynomials, and their weight function $w(x) = (1 - x^2)^{-1/2}$.*

6.3 Rational Approximation

In some cases, it is not practical to approximate a given function $f(x)$ by a polynomial, because it simply cannot capture the behavior of $f(x)$, regardless of the degree. This is because higher-degree polynomials tend to be oscillatory, so if $f(x)$ is mostly smooth, the degree n of an approximating polynomial $f_n(x)$ must be unreasonably high. Therefore, in this section we consider an alternative to polynomial approximation.

Specifically, we seek a *rational function* of the form

$$r_{m,n}(x) = \frac{p_m(x)}{q_n(x)} = \frac{a_0 + a_1x + a_2x^2 + \cdots + a_mx^m}{b_0 + b_1x + b_2x^2 + \cdots + b_nx^n},$$

where $p_m(x)$ and $q_n(x)$ are polynomials of degree m and n, respectively. For convenience, we impose $b_0 = 1$, since otherwise the other coefficients can simply be scaled.

To construct $p_m(x)$ and $q_n(x)$, we generalize approximation of $f(x)$ by a Taylor polynomial of degree n. Consider the error

$$E(x) = f(x) - r_{m,n}(x) = \frac{f(x)q_n(x) - p_m(x)}{q_n(x)}.$$

As in Taylor polynomial approximation, our goal is to choose the coefficients of p_m and q_n so that

$$E(0) = E'(0) = E''(0) = \cdots = E^{(m+n)} = 0.$$

That is, 0 is a root of multiplicity $m+n+1$. It follows that x^{m+n+1} is included in the factorization of the numerator of $E(x)$.

For convenience, we express p and q as polynomials of degree $m+n$, by padding them with coefficients thath are zero: $a_{m+1} = a_{m+2} = \cdots = a_{m+n} = 0$ and $b_{n+1} = b_{n+2} = \cdots = b_{n+m} = 0$. Taking a Maclaurin expansion of $f(x)$,

$$f(x) = \sum_{i=0}^{\infty} c_i x^i, \quad c_i = \frac{f^{(i)}(0)}{i!},$$

we obtain the following expression for this numerator:

$$\begin{aligned}
f(x)q_n(x) - p_m(x) &= \sum_{i=0}^{\infty} c_i x^i \sum_{j=0}^{m+n} b_j x^j - \sum_{i=0}^{m+n} a_i x^i \\
&= \sum_{i=0}^{\infty} \sum_{j=0}^{m+n} c_i b_j x^{i+j} - \sum_{i=0}^{m+n} a_i x^i \\
&= \sum_{i=0}^{\infty} \sum_{j=0}^{\min(m+n,i)} b_j c_{i-j} x^i - \sum_{i=0}^{m+n} a_i x^i \\
&= \sum_{i=0}^{m+n} \left[\sum_{j=0}^{i} b_j c_{i-j} - a_i \right] x_i + \sum_{i=m+n+1}^{\infty} \sum_{j=0}^{m+n} b_j c_{i-j} x^i.
\end{aligned}$$

We can then ensure that 0 is a root of multiplicity $m+n+1$ if the numerator has no terms of degree $m+n$ or less. That is, each coefficient of x^i in the first summation must equal zero.

This entails solving the system of $m+n+1$ equations

$$\begin{aligned}
c_0 &= a_0 \\
c_1 + b_1 c_0 &= a_1 \\
c_2 + b_1 c_1 + b_2 c_0 &= a_2 \\
&\vdots \\
c_{m+n} + b_1 c_{m+n-1} + \cdots + b_{m+n} c_0 &= a_{m+n}.
\end{aligned} \tag{6.7}$$

This is a system of $m+n+1$ linear equations in the $m+n+1$ unknowns $b_1, b_2, \ldots, b_n, a_0, a_1, \ldots, a_m$. The resulting rational function $r_{m,n}(x)$ is called a **Padé approximant** of $f(x)$ [27].

While this system can certainly be solved using Gaussian elimination with partial pivoting, we would like to find out if the structure of this system can somehow be exploited to solve it more efficiently than is possible for a general system of linear equations. To that end, we consider a simple example.

Example 6.3.1 *We consider the approximation of* $f(x) = e^{-x}$ *by a rational function of the form*

$$r_{2,3}(x) = \frac{a_0 + a_1 x + a_2 x^2}{1 + b_1 x + b_2 x^2 + b_3 x^3}.$$

The Maclaurin series for $f(x)$ *has coefficients* $c_j = (-1)^j/j!$. *The system of equations (6.7) becomes*

$$\begin{aligned} c_0 &= a_0 \\ c_1 + b_1 c_0 &= a_1 \\ c_2 + b_1 c_1 + b_2 c_0 &= a_2 \\ c_3 + b_1 c_2 + b_2 c_1 + b_3 c_0 &= 0 \\ c_4 + b_1 c_3 + b_2 c_2 + b_3 c_1 &= 0 \\ c_5 + b_1 c_4 + b_2 c_3 + b_3 c_2 &= 0. \end{aligned}$$

This can be written as $A\mathbf{x} = \mathbf{b}$, *where*

$$A = \begin{bmatrix} -1 & & & & & \\ & -1 & & c_0 & & \\ & & -1 & c_1 & c_0 & \\ & & & c_2 & c_1 & c_0 \\ & & & c_3 & c_2 & c_1 \\ & & & c_4 & c_3 & c_2 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -c_0 \\ -c_1 \\ -c_2 \\ -c_3 \\ -c_4 \\ -c_5 \end{bmatrix}.$$

If possible, we would like to work with a structure that facilitates Gaussian elimination. To that end, we can reverse the rows and columns of A, $\mathbf{x}$ *and* $\mathbf{b}$ *to obtain the system*

$$A = \begin{bmatrix} c_2 & c_3 & c_4 & & & \\ c_1 & c_2 & c_3 & & & \\ c_0 & c_1 & c_2 & & & \\ & c_0 & c_1 & -1 & & \\ & & c_0 & & -1 & \\ & & & & & -1 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -c_5 \\ -c_4 \\ -c_3 \\ -c_2 \\ -c_1 \\ -c_0 \end{bmatrix}.$$

It follows that Gaussian elimination can be carried out by eliminating $m = 2$ *entries in each of the first* $n = 3$ *columns. After that, the matrix will be reduced to upper triangular form so that back substitution can be carried out. If pivoting is required, it can be carried out on only the first* n *rows, because due to the block lower triangular structure of* A, *it follows that* A *is nonsingular if and only if the upper left* $n \times n$ *block is.*

After carrying out Gaussian elimination for this example, with Maclaurin series coefficients $c_j = (-1)^j/j!$, *we obtain the rational approximation*

$$e^{-x} \approx r_{2,3}(x) = \frac{p_2(x)}{q_3(x)} = \frac{\frac{1}{20}x^2 - \frac{2}{5}x + 1}{\frac{1}{60}x^3 + \frac{3}{20}x^2 + \frac{3}{5}x + 1}.$$

Plotting the error in this approximation on the interval $[0, 1]$, *we see that the error is maximum at* $x = 1$, *at roughly* 4.5×10^{-5}. □

Exercise 6.3.1 *Write a* MATLAB *function* `[p,q]=padeapprox(c,m,n)` *that computes the coefficients of the polynomials* $p_m(x)$ *and* $q_n(x)$ *such that* $r_{m,n}(x) = p_m(x)/q_n(x)$ *is the Padé approximant of degree* m, n *for the function* $f(x)$ *with Maclaurin series coefficients* $c_0, c_1, \ldots, c_{m+n}$ *stored in the vector* `c`.

6.3.1 Continued Fraction Form

We now examine the process of efficiently evaluating $r_{m,n}(x)$. A natural approach is to simply apply nested multiplication to $p_m(x)$ and $q_n(x)$ individually.

Example 6.3.2 *If we apply nested multiplication to* $p_2(x)$ *and* $q_3(x)$ *from Example 6.3.1, we obtain*

$$p_2(x) = 1 + x\left(-\frac{2}{5} + \frac{1}{20}x\right), \quad q_3(x) = 1 + x\left(\frac{3}{5} + x\left(\frac{3}{20} + \frac{1}{60}x\right)\right).$$

It follows that evaluating $r_{2,3}(x)$ *requires 5 multiplications, 5 additions, and one division.*

An alternative approach is to represent $r_{2,3}(x)$ *as a* **continued fraction** *[29, p. 285-322]. We have*

$$\begin{aligned} r_{2,3}(x) &= \frac{p_2(x)}{q_3(x)} = \frac{\frac{1}{20}x^2 - \frac{2}{5}x + 1}{\frac{1}{60}x^3 + \frac{3}{20}x^2 + \frac{3}{5}x + 1} \\ &= \frac{3}{\frac{x^3 + 9x^2 + 36x + 60}{x^2 - 8x + 20}} \\ &= \frac{3}{x + 17 + \frac{152x - 280}{x^2 - 8x + 20}} \\ &= \frac{3}{x + 17 + \frac{8}{\frac{x^2 - 8x + 20}{19x - 35}}} \\ &= \frac{3}{x + 17 + \frac{152}{x - \frac{117}{19} + \frac{3125/361}{x - 35/19}}} \end{aligned}$$

In this form, evaluation of $r_{2,3}(x)$ *requires three divisions, no multiplications, and five additions, resulting in significantly more efficiency than using nested multiplication on* $p_2(x)$ *and* $q_3(x)$. □

It is important to note that the efficiency of this approach comes from the ability to make the polynomial in each denominator monic–that is, having a leading coefficient of one–to remove the need for a multiplication.

Exercise 6.3.2 *Write a* MATLAB *function* `y=contfrac(p,q,x)` *that takes as input polynomials* $p(x)$ *and* $q(x)$*, represented as vectors of coefficients* `p` *and* `q`*, respectively, and outputs* $y = p(x)/q(x)$ *by evaluating* $p(x)/q(x)$ *as a continued fraction.* Hint: *use the* MATLAB *function* `deconv` *to divide polynomials.*

6.3.2 Chebyshev Rational Approximation

One drawback of the Padé approximant is that while it is highly accurate near $x = 0$, it loses accuracy as x moves away from zero. Certainly it is straightforward to perform Taylor expansion around a different center x_0, which ensures similar accuracy near x_0, but it would be desirable to instead compute an approximation that is accurate on an entire interval $[a, b]$.

To that end, we can employ the *Chebyshev polynomials*, previously discussed in Section 5.4.2. Just as they can help reduce the error in polynomial interpolation over an interval, they can also improve the accuracy of a rational approximation over an interval. For simplicity, we consider the interval $(-1, 1)$, on which each Chebyshev polynomial $T_k(x)$ satisfies $|T_k(x)| \leq 1$, but the approach described here can readily be applied to an arbitrary interval through shifting and scaling as needed.

The main idea is to use $T_k(x)$ in place of x^k in constructing our rational approximation. That is, our rational approximation now has the form

$$r_{m,n}(x) = \frac{p_m(x)}{q_n(x)} = \frac{\sum_{k=0}^{m} a_k T_k(x)}{\sum_{k=0}^{n} b_k T_k(x)}.$$

If we also expand $f(x)$ in a series of Chebyshev polynomials,

$$f(x) = \sum_{k=0}^{\infty} c_k T_k(x), \tag{6.8}$$

then the error in our approximation is

$$E(x) = f(x) - r_{m,n}(x) = \frac{f(x)q_n(x) - p_m(x)}{q_n(x)} = \frac{1}{q_n(x)} \left[\sum_{i=0}^{\infty} \sum_{j=0}^{n} c_i b_j T_i(x) T_j(x) - \sum_{i=0}^{m} a_i T_i(x) \right].$$

By applying the identity

$$T_i(x)T_j(x) = \frac{1}{2}[T_{i+j}(x) + T_{|i-j|}(x)], \tag{6.9}$$

we obtain the error

$$\begin{aligned} E(x) &= \frac{1}{q_n(x)} \left[\frac{1}{2} \sum_{i=0}^{\infty} \sum_{j=0}^{n} c_i b_j [T_{i+j}(x) + T_{|i-j|}(x)] - \sum_{i=0}^{m} a_i T_i(x) \right] \\ &= \frac{1}{q_n(x)} \left[\sum_{i=0}^{\infty} c_i T_i(x) + \frac{1}{2} \sum_{j=1}^{n} b_j \left[\sum_{i=j}^{\infty} c_{i-j} T_i(x) + \sum_{i=1}^{j} c_{j-i} T_i(x) + \sum_{i=0}^{\infty} c_{i+j} T_i(x) \right] - \right. \\ &\qquad \left. \sum_{i=0}^{m} a_i T_i(x) \right]. \end{aligned} \tag{6.10}$$

The coefficients $\{a_j\}_{i=0}^{m}$, $\{b_j\}_{j=1}^{n}$ are then determined by requiring that the coefficient of $T_i(x)$ in $E(x)$ vanishes, for $i = 0, 1, 2, \ldots, m + n$.

To obtain the coefficients $\{c_j\}_{j=0}^{\infty}$ in the series expansion of $f(x)$ from (6.8), we use the fact that the Chebyshev polynomials are orthogonal on $(-1, 1)$ with respect to the weight function $w(x) = (1 - x^2)^{-1/2}$. By taking the inner product of both sides of (6.8), formulas for c_j can be obtained.

Exercise 6.3.3 *Derive a formula for the coefficients* c_j, $j = 0, 1, 2, \ldots,$ *of the expansion of* $f(x)$ *in a series of Chebyshev polynomials in (6.8).*

Example 6.3.3 *We consider the approximation of* $f(x) = e^{-x}$ *by a rational function of the form*

$$r_{2,3}(x) = \frac{a_0T_0(x) + a_1T_1(x) + a_2T_2(x)}{1 + b_1T_1(x) + b_2T_2(x) + b_3T_3(x)}.$$

The Chebyshev series (6.8) for $f(x)$ *has coefficients* c_j *that can be obtained using the result of Exercise 6.3.3. The system of equations implied by (6.10) becomes*

$$\begin{aligned} c_0 + \frac{1}{2}(b_1c_1 + b_2c_2 + b_3c_3) &= a_0 \\ c_1 + b_1c_0 + \frac{1}{2}(b_1c_2 + b_2c_1 + b_2c_3 + b_3c_2 + b_3c_4) &= a_1 \\ c_2 + b_2c_0 + \frac{1}{2}(b_1c_1 + b_1c_3 + b_2c_4 + b_3c_1 + b_3c_5) &= a_2 \\ c_3 + b_3c_0 + \frac{1}{2}(b_1c_2 + b_1c_4 + b_2c_1 + b_2c_5 + b_3c_6) &= 0 \\ c_4 + \frac{1}{2}(b_1c_3 + b_1c_5 + b_2c_2 + b_2c_6 + b_3c_1 + b_3c_7) &= 0 \\ c_5 + \frac{1}{2}(b_1c_4 + b_1c_6 + b_2c_3 + b_2c_7 + b_3c_2 + c_3c_8) &= 0. \end{aligned}$$

This can be written as $A\mathbf{x} = \mathbf{b}$, *where*

$$A = \begin{bmatrix} -1 & & & & & \\ & -1 & & & & \\ & & -1 & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix} + \frac{1}{2}\begin{bmatrix} & & & & & \\ & & & c_0 & & \\ & & & & c_0 & \\ & & & & & c_0 \\ & & & & & \\ & & & & & \end{bmatrix} +$$

$$\frac{1}{2}\begin{bmatrix} & & & & & \\ & & & c_0 & & \\ & & & c_1 & c_0 & \\ & & & c_2 & c_1 & c_0 \\ & & & c_3 & c_2 & c_1 \\ & & & c_4 & c_3 & c_2 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} & & & c_1 & c_2 & c_3 \\ & & & c_2 & c_3 & c_4 \\ & & & c_3 & c_4 & c_5 \\ & & & c_4 & c_5 & c_6 \\ & & & c_5 & c_6 & c_7 \\ & & & c_6 & c_7 & c_8 \end{bmatrix},$$

$$\mathbf{x} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -c_0 \\ -c_1 \\ -c_2 \\ -c_3 \\ -c_4 \\ -c_5 \end{bmatrix}.$$

After carrying out Gaussian elimination for this example, we obtain the rational approximation

$$e^{-x} \approx r_{2,3}(x) = \frac{p_2(x)}{q_3(x)} \approx \frac{0.0231x^2 - 0.3722x + 0.9535}{0.0038x^3 + 0.0696x^2 + 0.5696x + 1}.$$

Plotting the error in this approximation on the interval $(-1, 1)$*, we see that the error is maximum at* $x = -1$*, at roughly* 1.1×10^{-5}*, which is less than one-fourth of the error in the Padé approximant on* $[0, 1]$*. In fact, on* $[0, 1]$*, the error is maximum at* $x = 0$ *and is only* 4.1×10^{-6}*.* □

Exercise 6.3.4 *Write a* MATLAB *function* `[p,q]=chebyrat(c,m,n)` *that accepts as inputs a vector* `c` *consisting of the coefficients* $c_0, c_1, \ldots, c_{m+n}$ *in the expansion of a given function* $f(x)$ *in a series of Chebyshev polynomials as in (6.8), along with the degrees* `m` *and* `n` *of the numerator and denominator, respectively, of a rational Chebyshev interpolant* $r_{m,n}(x)$ *of* $f(x)$*. The output must be row vectors* `p` *and* `q` *containing the coefficients of the polynomials* $p_m(x)$ *and* $q_n(x)$ *for the numerator and denominator, respectively, of* $r_{m,n}(x)$*.*

6.4 Trigonometric Interpolation

In many application areas, such as differential equations and signal processing, it is more useful to express a given function $u(x)$ as a linear combination of sines and cosines, rather than polynomials. In differential equations, this form of approximation is beneficial due to the simplicity of differentiating sines and cosines, and in signal processing, one can readily analyze the frequency content of $u(x)$. In this section, we develop efficient algorithms for approximation of functions with such trigonometric functions.

6.4.1 Fourier Series

Suppose that a function $u(x)$ defined on the interval $[0, L]$ is intended to satisfy **periodic boundary conditions** $u(0) = u(L)$. Then, since $\sin x$ and $\cos x$ are both 2π-periodic, $u(x)$ can be expressed in terms of both sines and cosines, as follows:

$$u(x) = \frac{a_0}{2} + \sum_{j=1}^{\infty} a_j \cos\frac{2\pi jx}{L} + b_j \sin\frac{2\pi jx}{L}, \tag{6.11}$$

where, for $j = 0, 1, 2, \ldots$, the coefficients a_j and b_j are defined by

$$a_j = \frac{2}{L}\int_0^L u(x)\cos\frac{2\pi jx}{L}\,dx, \quad b_j = \frac{2}{L}\int_0^L u(x)\sin\frac{2\pi jx}{L}\,dx. \tag{6.12}$$

This series representation of $u(x)$ is called the **Fourier series** of $u(x)$.

The formulas for the coefficients $\{a_j\}$, $\{b_j\}$ in (6.12) are obtained using the fact that the functions $\{\cos(2\pi jx/L)\}$, $\{\sin(2\pi jx/L)\}$ are *orthogonal* with respect to the inner product

$$(f, g) = \int_0^L \overline{f(x)}g(x)\,dx, \tag{6.13}$$

which can be established using trigonometric identities. The complex conjugation of $f(x)$ in (6.13) is necessary to ensure that the *norm* $\|\cdot\|$ defined by

$$\|u\| = \sqrt{(u, u)} \tag{6.14}$$

satisfies one of the essential properties of norms, that the norm of a function must be nonnegative.

Exercise 6.4.1 *Prove that if m, n are integers, then*

$$\left(\cos\frac{2\pi mx}{L}, \cos\frac{2\pi nx}{L}\right) = \begin{cases} 0 & m \neq n, \\ L/2 & m = n, n \neq 0, \\ L & m = n = 0 \end{cases},$$

$$\left(\sin\frac{2\pi mx}{L}, \sin\frac{2\pi nx}{L}\right) = \begin{cases} 0 & m \neq n, \\ L/2 & m = n \end{cases},$$

$$\left(\cos\frac{2\pi mx}{L}, \sin\frac{2\pi nx}{L}\right) = 0,$$

where the inner product (f, g) is as defined in (6.13).

Alternatively, we can use the relation $e^{i\theta} = \cos\theta + i\sin\theta$ to express the solution in terms of complex exponentials,

$$u(x) = \frac{1}{\sqrt{L}} \sum_{\omega=-\infty}^{\infty} \hat{u}(\omega) e^{2\pi i\omega x/L}, \tag{6.15}$$

where

$$\hat{u}(\omega) = \frac{1}{\sqrt{L}} \int_0^L e^{-2\pi i\omega x/L} u(x)\, dx. \tag{6.16}$$

Like the sines and cosines in (6.11), the functions $e^{2\pi i\omega x/L}$ are orthogonal with respect to the inner product (6.13). Specifically, we have

$$\left(e^{2\pi i\omega x/L}, e^{2\pi i\eta x/L}\right) = \begin{cases} L & \omega = \eta \\ 0 & \omega \neq \eta \end{cases}. \tag{6.17}$$

This explains the presence of the scaling constant $1/\sqrt{L}$ in (6.15). It normalizes the functions $e^{2\pi i\omega x/L}$ so that they form an orthonormal set, meaning that they are orthogonal to one another, and have unit norm.

Exercise 6.4.2 *Prove (6.17).*

We say that $f(x)$ is **square-integrable** on $(0, L)$ if

$$\int_0^L |f(x)|^2\, dx < \infty. \tag{6.18}$$

That is, the above integral must be finite; we also say that $f \in L^2(0, L)$. If such a function is also piecewise continuous, the following identity, known as **Parseval's identity**, is satisfied:

$$\sum_{\omega=-\infty}^{\infty} |\hat{f}(\omega)|^2 = \|f\|^2, \tag{6.19}$$

where the norm $\|\cdot\|$ is as defined in (6.14).

Exercise 6.4.3 *Prove (6.19).*

6.4.2 The Discrete Fourier Transform

Suppose that we define a grid on the interval $[0, L]$, consisting of the N points $x_j = j\Delta x$, where $\Delta x = L/N$, for $j = 0, \ldots, N-1$. Given an L-periodic function $f(x)$, we would like to compute an approximation to its Fourier series of the form

$$f_N(x) = \frac{1}{\sqrt{L}} \sum_{\omega=-N/2+1}^{N/2} e^{2\pi i\omega x/L} \tilde{f}(\omega), \tag{6.20}$$

where each $\tilde{f}(\omega)$ approximates the corresponding coefficient $\hat{f}(\omega)$ of the true Fourier series. Ideally, this approximate series should satisfy

$$f_N(x_j) = f(x_j), \quad j = 0, 1, \ldots, N-1. \tag{6.21}$$

That is, $f_N(x)$ should be an *interpolant* of $f(x)$, with the N points x_j, $j = 0, 1, \ldots, N-1$, as the interpolation points.

6.4.2.1 Fourier Interpolation

The problem of finding this interpolant, called the **Fourier interpolant** of f, has a unique solution that can easily be computed. The coefficients $\tilde{f}(\omega)$ are obtained by approximating the integrals that defined the coefficients of the the Fourier series:

$$\tilde{f}(\omega) = \frac{1}{\sqrt{L}} \sum_{j=0}^{N-1} e^{-2\pi i\omega x_j/L} f(x_j)\Delta x, \quad \omega = -N/2+1, \ldots, N/2. \tag{6.22}$$

Because the functions $e^{2\pi i\omega x/L}$ are orthogonal with respect to the discrete inner product

$$(u, v)_N = \Delta x \sum_{j=0}^{N-1} \overline{u(x_j)} v(x_j), \tag{6.23}$$

it is straightforward to verify that $f_N(x)$ does in fact satisfy the conditions (6.21). Note that the discrete inner product is an approximation of the continuous inner product.

From (6.21), we have

$$f(x_j) = \frac{1}{\sqrt{L}} \sum_{\eta=-N/2+1}^{N/2} e^{2\pi i\eta x_j/L} \tilde{f}(\eta). \tag{6.24}$$

Multiplying both sides by $\Delta x e^{-2\pi i\omega x_j/L}$, and summing from $j = 0$ to $j = N-1$ yields

$$\Delta x \sum_{j=0}^{N-1} e^{-2\pi i\omega x_j/L} f(x_j) = \Delta x \frac{1}{\sqrt{L}} \sum_{j=0}^{N-1} \sum_{\eta=-N/2+1}^{N/2} e^{-2\pi i\omega x_j/L} e^{2\pi i\eta x_j/L} \tilde{f}(\eta), \tag{6.25}$$

or

$$\Delta x \sum_{j=0}^{N-1} e^{-2\pi i\omega x_j/L} f(x_j) = \frac{1}{\sqrt{L}} \sum_{\eta=-N/2+1}^{N/2} \tilde{f}(\eta) \left[\Delta x \sum_{j=0}^{N-1} e^{-2\pi i\omega x_j/L} e^{2\pi i\eta x_j/L} \right]. \tag{6.26}$$

Because

$$\left(e^{2\pi i\omega x/L}, e^{2\pi i\eta x/L}\right)_N = \begin{cases} L & \omega = \eta \\ 0 & \omega \neq \eta \end{cases}, \tag{6.27}$$

all terms in the outer sum on the right side of (6.26) vanish except for $\eta = \omega$, and we obtain the formula (6.22). It should be noted that the algebraic operations performed on (6.24) are equivalent to taking the discrete inner product of both sides of (6.24) with $e^{2\pi i\omega x/L}$.

Exercise 6.4.4 *Prove (6.27).* Hint: *use formulas associated with geometric series.*

The process of obtaining the approximate Fourier coefficients as in (6.22) is called the **discrete Fourier transform** (DFT) of $f(x)$. The discrete inverse Fourier transform is given by (6.20). As at the beginning of this section, we can also work with the real form of the Fourier interpolant,

$$f_N(x) = \frac{\tilde{a}_0}{2} + \sum_{j=1}^{N/2-1} \tilde{a}_j \cos\frac{2\pi jx}{L} + \tilde{b}_j \sin\frac{2\pi jx}{L} + \tilde{a}_{N/2}\cos\frac{\pi Nx}{L}, \tag{6.28}$$

where the coefficients $\tilde{a}_j$, $\tilde{b}_j$ are approximations of the coefficients a_j, b_j from (6.12).

Exercise 6.4.5 *Express the coefficients $\tilde{a}_j$, $\tilde{b}_j$ of the real form of the Fourier interpolant (6.28) in terms of the coefficients $\tilde{f}(\omega)$ from the complex exponential form (6.20).*

Exercise 6.4.6 *Why is there no need for a coefficient $\tilde{b}_{N/2}$ in (6.28)?*

Exercise 6.4.7 *Use the result of Exercise 6.4.4 to prove the following discrete orthogonality relations:*

$$\left(\cos\frac{2\pi mx}{L}, \cos\frac{2\pi nx}{L}\right)_N = \begin{cases} 0 & m \neq n, \\ L/2 & m = n, n \neq 0, \\ L & m = n = 0 \end{cases},$$

$$\left(\sin\frac{2\pi mx}{L}, \sin\frac{2\pi nx}{L}\right)_N = \begin{cases} 0 & m \neq n, \\ L/2 & m = n \end{cases},$$

$$\left(\cos\frac{2\pi mx}{L}, \sin\frac{2\pi nx}{L}\right)_N = 0,$$

where m and n are integers, and the discrete inner product $(f, g)_N$ is as defined in (6.23).

6.4.2.2 De-Noising and Aliasing

Suppose we have $N = 128$ data points sampled from the following function over $[0, 2\pi]$:

$$f(x) = \sin(10x) + \text{noise}. \tag{6.29}$$

The function $f(x)$, shown in Figure 6.5(a), is quite noisy. However, by taking the discrete Fourier transform (Figure 6.5(b)), we can extract the original sine wave quite easily. The DFT shows two distinct spikes, corresponding to frequencies of $\omega = \pm 10$, that is, the frequencies of the original sine wave. The first $N/2 + 1$ values of the Fourier transform correspond to frequencies of $0 \leq \omega \leq \omega_{max}$,

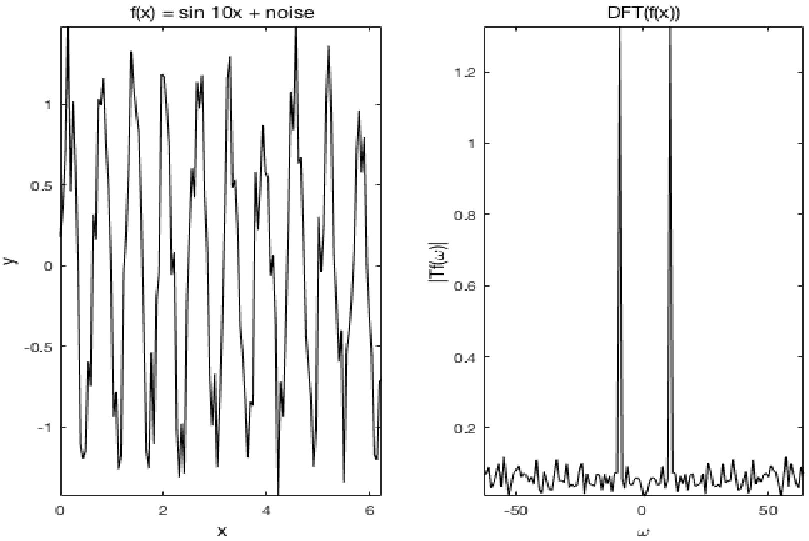


Figure 6.5: (a) Left plot: noisy signal (b) Right plot: discrete Fourier transform

where $\omega_{max} = N/2$. The remaining $N/2 - 1$ values of the Fourier transform correspond to the frequencies $-\omega_{max} < \omega < 0$.

The DFT only considers a finite range of frequencies. If there are frequencies beyond this present in the Fourier series, an effect known as **aliasing** occurs. The effect of aliasing is shown in Figure 6.6: it “folds” these frequencies back into the computed DFT. Specifically,

$$\tilde{f}(\omega) = \sum_{\ell=-\infty}^{\infty} \hat{f}(\omega + \ell N), \quad -N/2 + 1 \leq \omega \leq N/2. \tag{6.30}$$

Aliasing can be avoided by filtering the function before the DFT is applied, to prevent high-frequency components from “contaminating” the coefficients of the DFT.

Exercise 6.4.8 *Use (6.18) and (6.20) to prove (6.30).* Hint: *Let $x = x_j$ for some j.*

6.4.3 The Fast Fourier Transform

The discrete Fourier transform, as it was presented in the previous lecture, requires $O(N^2)$ operations to compute. In fact the discrete Fourier transform can be computed much more efficiently than that ($O(N \log_2 N)$ operations) by using the fast Fourier transform (FFT). The FFT arises by noting that a DFT of length N can be written as the sum of two Fourier transforms each of length $N/2$. One of these transforms is formed from the even-numbered points of the original N, and the other from the odd-numbered points.

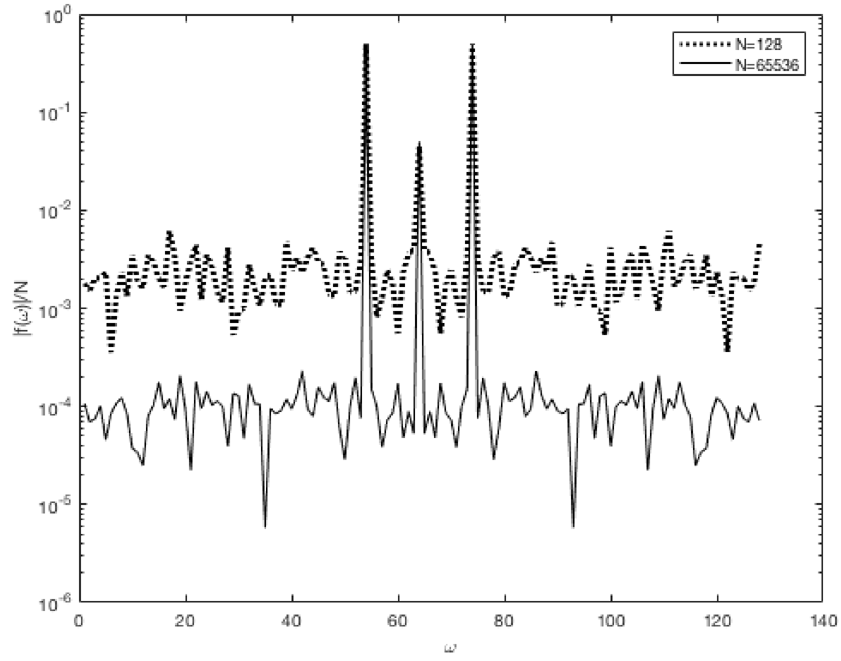


Figure 6.6: Aliasing effect on noisy signal: coefficients $\hat{f}(\omega)$, for ω outside $(-63, 64)$, are added to coefficients inside this interval.

We have

$$\begin{aligned}\tilde{f}(\omega) &= \frac{\Delta x}{\sqrt{L}} \sum_{j=0}^{N-1} e^{-2\pi i j\omega/N} f(x_j) \\ &= \frac{\Delta x}{\sqrt{L}} \sum_{j=0}^{N/2-1} e^{-2\pi i\omega(2j)/N} f(x_{2j}) + \frac{\Delta x}{\sqrt{L}} \sum_{j=0}^{N/2-1} e^{-2\pi i\omega(2j+1)/N} f(x_{2j+1}) \\ &= \frac{\Delta x}{\sqrt{L}} \sum_{j=0}^{N/2-1} e^{-2\pi i\omega j/(N/2)} f(x_{2j}) + \frac{\Delta x}{\sqrt{L}} W^{\omega} \sum_{j=0}^{N/2-1} e^{-2\pi i\omega j/(N/2)} f(x_{2j+1}) \qquad (6.31)\end{aligned}$$

where

$$W = e^{-2\pi i/N}. \qquad (6.32)$$

It follows that

$$\tilde{f}(\omega) = \frac{1}{2}\tilde{f}^e(\omega) + \frac{1}{2}W^{\omega}\tilde{f}^o(\omega), \quad \omega = -N/2+1, \ldots, N/2, \qquad (6.33)$$

where $\tilde{f}^e(\omega)$ is the DFT of f obtained from its values at the even-numbered points of the N-point grid on which f is defined, and $\tilde{f}^o(\omega)$ is the DFT of f obtained from its values at the the odd-numbered points. Because the coefficients of a DFT of length N are N-periodic, in view of the identity $e^{2\pi i} = 1$, evaluation of $\tilde{f}^e$ and $\tilde{f}^o$ at ω between $-N/2+1$ and $N/2$ is valid, even though they are transforms of length $N/2$ instead of N.

This reduction to half-size transforms can be performed recursively; i.e. a transform of length $N/2$ can be written as the sum of two transforms of length $N/4$, etc. Because only $O(N)$ operations are needed to construct a transform of length N from two transforms of length $N/2$, the entire process requires only $O(N \log_2 N)$ operations.

Exercise 6.4.9 *Write two functions to compute the DFT of a function* $f(x)$ *defined on* $[0, L]$*, represented by a* N*-vector* `f` *that contains its values at* $x_j = j\Delta x$*,* $j = 0, 1, 2, \ldots, N-1$*, where* $j = L/N$*. For the first function, use the formula (6.20), and for the second, use recursion and the formula (6.33) for the FFT. Compare the efficiency of your functions for different values of* N*. How does the running time increase as* N *increases?*

6.4.4 Convergence and Gibbs' Phenomenon

The Fourier series for an L-periodic function $f(x)$ will converge to $f(x)$ at any point in $[0, L]$ at which f is continuously differentiable. If f has a jump discontinuity at a point c, then the series will converge to $\frac{1}{2}[f(c^+) + f(c^-)]$, where

$$f(c^+) = \lim_{x \to c^+} f(x), \quad f(c^-) = \lim_{x \to c^-} f(x). \tag{6.34}$$

If $f(x)$ is *not* L-periodic, then there is a jump discontinuity in the L-periodic extension of $f(x)$ beyond $[0, L]$, and the Fourier series will again converge to the average of the values of $f(x)$ on either side of this discontinuity.

Such discontinuities pose severe difficulties for trigonometric interpolation, because the basis functions $e^{i\omega x}$ grow more oscillatory as $|\omega|$ increases. In particular, the truncated Fourier series of a function $f(x)$ with a jump discontinuity at $x = c$ exhibits what is known as **Gibbs' phenomenon**, first discussed in [37], in which oscillations appear on either side of $x = c$, even if $f(x)$ itself is smooth there.

Convergence of the Fourier series of f is more rapid when f is smooth. In particular, if f is p-times differentiable and its pth derivative is at least *piecewise* continuous (that is, continuous except possibly for jump discontinuities), then the coefficients of the complex exponential form of the Fourier series satisfy

$$|\hat{f}(\omega)| \leq \frac{C}{|\omega|^{p+1} + 1} \tag{6.35}$$

for some constant C that is independent of ω [17].

Exercise 6.4.10 *Generate a random vector of DFT coefficients that satisfy the decay rate (6.35), for some value of* p*. Then, perform an inverse FFT to obtain the truncated Fourier series (6.20), and plot the resulting function* $f_N(x)$*. How does the behavior of the function change as* p *decreases?*

Exercise 6.4.11 *Demonstrate Gibbs' phenomenon by plotting truncated Fourier series of the function* $f(x) = x$ *on* $[0, 2\pi]$*. Use the formula (6.20), evaluated on a finer grid (that is, using* $\tilde{N}$ *equally spaced points in* $[0, 2\pi]$*, where* $\tilde{N} \gg N$*). What happens as* N *increases?*

Chapter 7

Differentiation and Integration

The solution of many mathematical models requires performing the basic operations of calculus, differentiation and integration. In this chapter, we will learn several techniques for approximating a derivative of a function at a point, and a definite integral of a function over an interval. As we will see, our previous discussion of polynomial interpolation will play an essential role, as polynomials are the easiest functions on which to perform these operations.

7.1 Numerical Differentiation

We first discuss how Taylor series and polynomial interpolation can be applied to help solve a fundamental problem from calculus that frequently arises in scientific applications, the problem of computing the derivative of a given function $f(x)$ at a given point $x = x_0$. The basics of derivatives are reviewed in Section A.2.

7.1.1 Taylor Series

Recall that the derivative of $f(x)$ at a point x_0, denoted $f'(x_0)$, is defined by

$$f'(x_0) = \lim_{h\to 0} \frac{f(x_0+h) - f(x_0)}{h}.$$

This definition suggests a method for approximating $f'(x_0)$. If we choose h to be a small positive constant, then

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0)}{h}.$$

This approximation is called the **forward difference formula**.

To estimate the accuracy of this approximation, we note that if $f''(x)$ exists on $[x_0, x_0 + h]$, then, by Taylor's Theorem, $f(x_0+h) = f(x_0)+f'(x_0)h+f''(\xi)h^2/2$, where $\xi \in [x_0, x_0+h]$. Solving for $f'(x_0)$, we obtain

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - \frac{f''(\xi)}{2}h,$$

so the error in the forward difference formula is $O(h)$. We say that this formula is **first-order accurate**.

The forward-difference formula is called a **finite difference approximation** to $f'(x_0)$, because it approximates $f'(x)$ using values of $f(x)$ at points that have a small, but finite, distance between them, as opposed to the definition of the derivative, that takes a limit and therefore computes the derivative using an "infinitely small" value of h. The forward-difference formula, however, is just one example of a finite difference approximation. If we replace h by $-h$ in the forward-difference formula, where h is still positive, we obtain the **backward-difference formula**

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - h)}{h}.$$

Like the forward-difference formula, the backward difference formula is first-order accurate.

If we average these two approximations, we obtain the **centered difference formula**

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h}.$$

To determine the accuracy of this approximation, we assume that $f'''(x)$ exists on the interval $[x_0 - h, x_0 + h]$, and then apply Taylor's Theorem again to obtain

$$\begin{aligned} f(x_0 + h) &= f(x_0) + f'(x_0)h + \frac{f''(x_0)}{2}h^2 + \frac{f'''(\xi_+)}{6}h^3, \\ f(x_0 - h) &= f(x_0) - f'(x_0)h + \frac{f''(x_0)}{2}h^2 - \frac{f'''(\xi_-)}{6}h^3, \end{aligned}$$

where $\xi_+ \in [x_0, x_0 + h]$ and $\xi_- \in [x_0 - h, x_0]$. Subtracting the second equation from the first and solving for $f'(x_0)$ yields

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} - \frac{f'''(\xi_+) + f'''(\xi_-)}{12}h^2.$$

Suppose that f''' is continuous on $[x_0 - h, x_0 + h]$. By the Intermediate Value Theorem, $f'''(x)$ must assume every value between $f'''(\xi_-)$ and $f'''(\xi_+)$ on the interval (ξ_-, ξ_+), including the average of these two values. Therefore, we can simplify this equation to

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} - \frac{f'''(\xi)}{6}h^2, \tag{7.1}$$

where $\xi \in [x_0 - h, x_0 + h]$. We conclude that the centered-difference formula is *second-order accurate.* This is due to the cancellation of the terms involving $f''(x_0)$.

Example 7.1.1 *Consider the function*

$$f(x) = \frac{\sin^2\left(\frac{\sqrt{x^2+x}}{\cos x - x}\right)}{\sin\left(\frac{\sqrt{x}-1}{\sqrt{x^2+1}}\right)}.$$

Our goal is to compute $f'(0.25)$. Differentiating, using the Quotient Rule and the Chain Rule, we obtain

$$\begin{aligned} f'(x) \quad = \quad & \frac{2\sin\left(\frac{\sqrt{x^2+x}}{\cos x - x}\right)\cos\left(\frac{\sqrt{x^2+x}}{\cos x - x}\right)\left[\frac{2x+1}{2\sqrt{x^2+1}(\cos x - x)} + \frac{\sqrt{x^2+1}(\sin x + 1)}{(\cos x - x)^2}\right]}{\sin\left(\frac{\sqrt{x}-1}{\sqrt{x^2+1}}\right)} - \\ & \frac{\sin\left(\frac{\sqrt{x^2+x}}{\cos x - x}\right)\cos\left(\frac{\sqrt{x}-1}{\sqrt{x^2+1}}\right)\left[\frac{1}{2\sqrt{x}\sqrt{x^2+1}} - \frac{x(\sqrt{x}-1)}{(x^2+1)^{3/2}}\right]}{\sin^2\left(\frac{\sqrt{x}-1}{\sqrt{x^2+1}}\right)}. \end{aligned}$$

Evaluating this monstrous function at $x = 0.25$ *yields* $f'(0.25) = -9.066698770$.

An alternative approach is to use a centered difference *approximation,*

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

Using this formula with $x = 0.25$ *and* $h = 0.005$, *we obtain the approximation*

$$f'(0.25) \approx \frac{f(0.255) - f(0.245)}{0.01} = -9.067464295,$$

which has absolute error 7.7×10^{-4}. *While this complicated function must be evaluated twice to obtain this approximation, that is still much less work than using differentiation rules to compute* $f'(x)$, *and then evaluating* $f'(x)$, *which is much more complicated than* $f(x)$. □

A similar approach can be used to obtain finite difference approximations of $f'(x_0)$ involving any points of our choosing, and at an arbitrarily high order of accuracy, provided that sufficiently many points are used.

Exercise 7.1.1 *Use Taylor series expansions of* $f(x_0 \pm jh)$, *for* $j = 1, 2, 3$, *to derive a finite difference approximation of* $f'(x_0)$ *that is 6th-order accurate. What is the error formula?*

Exercise 7.1.2 *Generalizing the process carried out by hand in Exercise 7.1.1, write a* MATLAB *function* `c=makediffrule(p)` *that takes as input a row vector of indices* `p` *and returns in a vector* `c` *the coefficients of a finite-difference approximation of* $f'(x_0)$ *that has the form*

$$f'(x_0) \approx \frac{1}{h} \sum_{j=1}^{n} c_j f(x_0 + p_j h),$$

where n *is the length of* `p`.

7.1.2 Lagrange Interpolation

While Taylor's Theorem can be used to derive formulas with higher-order accuracy simply by evaluating $f(x)$ at more points, this process can be tedious. An alternative approach is to compute the derivative of the interpolating polynomial that fits $f(x)$ at these points. Specifically, suppose we want to compute the derivative at a point x_0 using the data

$$(x_{-j}, y_{-j}), \ldots, (x_{-1}, y_{-1}), (x_0, y_0), (x_1, y_1), \ldots, (x_k, y_k),$$

where j and k are known nonnegative integers, $x_{-j} < x_{-j+1} < \cdots < x_{k-1} < x_k$, and $y_i = f(x_i)$ for $i = -j, \ldots, k$. Then, a finite difference formula for $f'(x_0)$ can be obtained by analytically computing the derivatives of the Lagrange polynomials $\{\mathcal{L}_{n,i}(x)\}_{i=-j}^{k}$ for these points, where $n = j + k$, and the values of these derivatives at x_0 are the proper weights for the function values $y_{-j}, \ldots, y_k$. If $f(x)$ is $n+1$ times continuously differentiable on $[x_{-j}, x_k]$, then we obtain an approximation of the form

$$f'(x_0) = \sum_{i=-j}^{k} y_i \mathcal{L}'_{n,i}(x_0) + \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=-j, i\neq 0}^{k} (x_0 - x_i), \tag{7.2}$$

where $\xi \in [x_{-j}, x_k]$.

Exercise 7.1.3 *Prove (7.2) using the error formula for Lagrange interpolation.* Hint: *Use the fact that the unknown point ξ in the error formula is an (unknown) function of x.*

Exercise 7.1.4 *Modify your function* `makediffrule` *from Exercise 7.1.2 so that it uses Lagrange interpolation rather than Taylor series expansion. Make it return a second output* `err` *which is the constant C such that the error in (7.2) is of the form $Ch^n f^{(n+1)}(\xi)$, where $n = j + k$.*

Among the best-known finite difference formulas that can be derived using this approach is the second-order-accurate three-point formula

$$f'(x_0) = \frac{-3f(x_0) + 4f(x_0+h) - f(x_0+2h)}{2h} + \frac{f'''(\xi)}{3}h^2, \quad \xi \in [x_0, x_0+2h], \tag{7.3}$$

which is useful when there is no information available about $f(x)$ for $x < x_0$. If there is no information available about $f(x)$ for $x > x_0$, then we can replace h by $-h$ in the above formula to obtain a second-order-accurate three-point formula that uses the values of $f(x)$ at x_0, $x_0 - h$ and $x_0 - 2h$.

Another formula is the five-point formula

$$f'(x_0) = \frac{f(x_0-2h) - 8f(x_0-h) + 8f(x_0+h) - f(x_0+2h)}{12h} + \frac{f^{(5)}(\xi)}{30}h^4, \quad \xi \in [x_0-2h, x_0+2h],$$

which is fourth-order accurate. The reason it is called a five-point formula, even though it uses the value of $f(x)$ at four points, is that it is derived from the Lagrange polynomials for the five points $x_0 - 2h, x_0 - h, x_0, x_0 + h$, and $x_0 + 2h$. However, $f(x_0)$ is not used in the formula because $\mathcal{L}'_{4,0}(x_0) = 0$, where $\mathcal{L}_{4,0}(x)$ is the Lagrange polynomial that is equal to one at x_0 and zero at the other four points.

If we do not have any information about $f(x)$ for $x < x_0$, then we can use the following five-point formula that actually uses the values of $f(x)$ at five points,

$$f'(x_0) = \frac{-25f(x_0) + 48f(x_0+h) - 36f(x_0+2h) + 16f(x_0+3h) - 3f(x_0+4h)}{12h} + \frac{f^{(5)}(\xi)}{5}h^4,$$

where $\xi \in [x_0, x_0 + 4h]$. As before, we can replace h by $-h$ to obtain a similar formula that approximates $f'(x_0)$ using the values of $f(x)$ at $x_0, x_0 - h, x_0 - 2h, x_0 - 3h$, and $x_0 - 4h$.

Exercise 7.1.5 *Use (7.2) to derive a general error formula for the approximation of $f'(x_0)$ in the case where $x_i = x_0 + ih$, for $i = -j, \ldots, k$. Use the preceding examples to check the correctness of your error formula.*

Example 7.1.2 *We will construct a formula for approximating $f'(x)$ at a given point x_0 by interpolating $f(x)$ at the points x_0, $x_0 + h$, and $x_0 + 2h$ using a second-degree polynomial $p_2(x)$, and then approximating $f'(x_0)$ by $p_2'(x_0)$. Since $p_2(x)$ should be a good approximation of $f(x)$ near x_0, especially when h is small, its derivative should be a good approximation to $f'(x)$ near this point.*

Using Lagrange interpolation, we obtain

$$p_2(x) = f(x_0)\mathcal{L}_{2,0}(x) + f(x_0+h)\mathcal{L}_{2,1}(x) + f(x_0+2h)\mathcal{L}_{2,2}(x),$$

where $\{\mathcal{L}_{2,j}(x)\}_{j=0}^{2}$ *are the Lagrange polynomials for the points* x_0*,* $x_1 = x_0 + h$ *and* $x_2 = x_0 + 2h$*. Recall that these polynomials satisfy*

$$\mathcal{L}_{2,j}(x_k) = \delta_{jk} = \begin{cases} 1 & \textit{if } j = k \\ 0 & \textit{otherwise} \end{cases}.$$

Using the formula for the Lagrange polynomials,

$$\mathcal{L}_{2,j}(x) = \prod_{i=0, i\neq j}^{2} \frac{(x - x_i)}{(x_j - x_i)},$$

we obtain

$$\begin{aligned}
\mathcal{L}_{2,0}(x) &= \frac{(x-(x_0+h))(x-(x_0+2h))}{(x_0-(x_0+h))(x_0-(x_0+2h))} \\
&= \frac{x^2-(2x_0+3h)x+(x_0+h)(x_0+2h)}{2h^2}, \\
\mathcal{L}_{2,1}(x) &= \frac{(x-x_0)(x-(x_0+2h))}{(x_0+h-x_0)(x_0+h-(x_0+2h))} \\
&= \frac{x^2-(2x_0+2h)x+x_0(x_0+2h)}{-h^2}, \\
\mathcal{L}_{2,2}(x) &= \frac{(x-x_0)(x-(x_0+h))}{(x_0+2h-x_0)(x_0+2h-(x_0+h))} \\
&= \frac{x^2-(2x_0+h)x+x_0(x_0+h)}{2h^2}.
\end{aligned}$$

It follows that

$$\begin{aligned}
\mathcal{L}_{2,0}'(x) &= \frac{2x-(2x_0+3h)}{2h^2} \\
\mathcal{L}_{2,1}'(x) &= -\frac{2x-(2x_0+2h)}{h^2} \\
\mathcal{L}_{2,2}'(x) &= \frac{2x-(2x_0+h)}{2h^2}
\end{aligned}$$

We conclude that $f'(x_0) \approx p_2'(x_0)$*, where*

$$\begin{aligned}
p_2'(x_0) &= f(x_0)\mathcal{L}_{2,0}'(x_0) + f(x_0+h)\mathcal{L}_{2,0}'(x_0) + f(x_0+2h)\mathcal{L}_{2,0}'(x_0) \\
&\approx f(x_0)\frac{-3}{2h} + f(x_0+h)\frac{2}{h} + f(x_0+2h)\frac{-1}{2h} \\
&\approx \frac{3f(x_0)+4f(x_0+h)-f(x_0+2h)}{2h}.
\end{aligned}$$

From (7.2), it can be shown (see Exercise 7.1.5) that the error in this approximation is $O(h^2)$*, and that this formula is exact when* $f(x)$ *is a polynomial of degree 2 or less. The error formula is given in (7.3).* □

7.1.3 Higher-Order Derivatives

The approaches of combining Taylor series or differentiating Lagrange polynomials to approximate derivatives can be used to approximate higher-order derivatives. For example, the second derivative can be approximated using a centered difference formula,

$$f''(x_0) \approx \frac{f(x_0+h) - 2f(x_0) + f(x_0-h)}{h^2}, \tag{7.4}$$

which is second-order accurate.

Exercise 7.1.6 *Use both Taylor series and Lagrange polynomials to derive (7.4). Which approach best facilitates computing an error formula for this approximation? What is the error formula?*

Exercise 7.1.7 *Generalize your function* `makediffrule` *from Exercise 7.1.4 so that it can compute the coefficients of a finite difference approximation to a derivative of a given order, which is specified as an input argument.*

7.1.4 Sensitivity

Based on the error formula for each of these finite difference approximations, one would expect that it is possible to obtain an approximation that is accurate to within machine precision simply by choosing h sufficiently small. In the following exercise, we can put this expectation to the test.

Exercise 7.1.8 *Use the centered difference formula (7.1) to compute an approximation of* $f'(x_0)$ *for* $f(x) = \sin x$, $x_0 = 1.2$, *and* $h = 10^{-d}$ *for* $d = 1, 2, \ldots, 15$. *Compare the error in each approximation with an upper bound for the error formula given in (7.1). How does the actual error compare to theoretical expectations?*

The reason for the discrepancy observed in Exercise 7.1.8 is that the error formula in (7.1), or any other finite difference approximation, only accounts for *discretization error*, not roundoff error.

In a practical implementation of finite difference formulas, it is essential to note that roundoff error in evaluating $f(x)$ is bounded independently of the spacing h between points at which $f(x)$ is evaluated. It follows that the roundoff error in the approximation of $f'(x)$ actually *increases* as h decreases, because the errors incurred by evaluating $f(x)$ are divided by h. Therefore, one must choose h sufficiently small so that the finite difference formula can produce an accurate approximation, and sufficiently large so that this approximation is not too contaminated by roundoff error.

7.1.5 Differentiation Matrices

It is often necessary to compute derivatives of a function $f(x)$ at a set of points within a given domain. If both $f(x)$ and $f'(x)$ are represented by vectors $\mathbf{f}$ and $\mathbf{g}$, respectively, whose elements are the values of f and f' at N selected points. Then, in view of the linearity of differentiation, $\mathbf{f}$ and $\mathbf{g}$ should be related by a linear transformation. That is, $\mathbf{g} = D\mathbf{f}$, where D is an $N \times N$ matrix. In this context, D is called a **differentiation matrix**.

Example 7.1.3 *We construct a differentiation matrix for functions defined on* $[0,1]$*, and satisfying the* boundary conditions $f(0) = f(1) = 0$*. Let* $x_1, x_2, \ldots, x_n$ *be* n *equally spaced points in* $(0,1)$*, defined by* $x_i = ih$*, where* $h = 1/(n+1)$*. If we use the forward difference approximation, we then have*

$$\begin{aligned} f'(x_1) &\approx \frac{f(x_2) - f(x_1)}{h}, \\ f'(x_2) &\approx \frac{f(x_3) - f(x_2)}{h}, \\ &\vdots \\ f'(x_{n-1} &\approx \frac{f(x_n) - f(x_{n-1})}{h}, \\ f'(x_n) &\approx \frac{0 - f(x_n)}{h}. \end{aligned}$$

Writing these equations in matrix-vector form, we obtain a relation of the form $\mathbf{g} \approx D\mathbf{f}$*, where*

$$\mathbf{g} = \begin{bmatrix} f'(x_1) \\ f'(x_2) \\ \vdots \\ f'(x_n) \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}, \quad D = \frac{1}{h} \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ & & & & -1 \end{bmatrix}.$$

The entries of D *can be determined from the coefficients of each value* $f(x_j)$ *used to approximate* $f'(x_i)$*, for* $i = 1, 2, \ldots, n$*. From the structure of this upper bidiagonal matrix, it follows that we can approximate* $f'(x)$ *at these grid points by a matrix-vector multiplication which costs only* $O(n)$ *floating-point operations.*

Now, suppose that we instead impose periodic boundary conditions $f(0) = f(1)$*. In this case, we again use* n *equally spaced points, but including the left boundary:* $x_i = ih$*,* $i = 0, 1, \ldots, n-1$*, where* $h = 1/n$*. Using forward differencing again, we have the same approximations as before, except*

$$f'(x_{n-1}) \approx \frac{f(1) - f(x_{n-1})}{h} = \frac{f(0) - f(x_{n-1})}{h} = \frac{f(x_1) - f(x_{n-1})}{h}.$$

It follows that the differentiation matrix is

$$D = \frac{1}{h} \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ 1 & & & & -1 \end{bmatrix}.$$

Note the "wrap-around" effect in which the superdiagonal appears to continue past the last column into the first column. For this reason, D *is an example of what is called a* **circulant matrix**. □

Exercise 7.1.9 *What are the differentiation matrices corresponding to (7.4) for functions defined on* $[0,1]$*, for (a) boundary conditions* $f(0) = f(1) = 0$*, and (b) periodic boundary conditions* $f(0) = f(1)$*?*

7.2 Numerical Integration

Numerous applications call for the computation of the integral of some function $f : \mathbb{R} \to \mathbb{R}$ over an interval $[a, b]$,

$$I[f] = \int_a^b f(x)\,dx.$$

In some cases, $I[f]$ can be computed by applying the Fundamental Theorem of Calculus and computing

$$I[f] = F(b) - F(a),$$

where $F(x)$ is an *antiderivative* of f, meaning that $F'(x) = f(x)$. Unfortunately, this is not practical if an antiderivative of f is not available. In such cases, numerical techniques must be employed instead. The basics of integrals are reviewed in Section A.4.

7.2.1 Quadrature Rules

Clearly, if f is a Riemann integrable function and $\{R_n\}_{n=1}^{\infty}$ is any sequence of Riemann sums that converges to $I[f]$, then any particular Riemann sum R_n can be viewed as an approximation of $I[f]$. However, such an approximation is usually not practical since a large value of n may be necessary to achieve sufficient accuracy.

Exercise 7.2.1 *Write a* MATLAB *script that computes the Riemann sum R_n for*

$$\int_0^1 x^2\,dx = \frac{1}{3},$$

where the left *endpoint of each subinterval is used to obtain the height of the corresponding rectangle. How large must n, the number of subintervals, be to obtain an approximate answer that is accurate to within 10^{-5}?*

Instead, we use a **quadrature rule** to approximate $I[f]$. A quadrature rule is a sum of the form

$$Q_n[f] = \sum_{i=1}^{n} f(x_i)w_i, \tag{7.5}$$

where the points x_i, $i = 1, \ldots, n$, are called the **nodes** of the quadrature rule, and the numbers w_i, $i = 1, \ldots, n$, are the **weights**. We say that a quadrature rule is **open** if the nodes do not include the endpoints a and b, and **closed** if they do.

The objective in designing quadrature rules is to achieve sufficient accuracy in approximating $I[f]$, for any Riemann integrable function f, while using as few nodes as possible in order to maximize efficiency. In order to determine suitable nodes and weights, we consider the following questions:

- For what functions f is $I[f]$ easy to compute?
- Given a general Riemann integrable function f, can $I[f]$ be approximated by the integral of a function g for which $I[g]$ is easy to compute?

7.2.2 Interpolatory Quadrature

One class of functions for which integrals are easily evaluated is the class of polynomial functions. If we choose n nodes $x_1, \ldots, x_n$, then any polynomial $p_{n-1}(x)$ of degree $n-1$ can be written in the form

$$p_{n-1}(x) = \sum_{i=1}^{n} p_{n-1}(x_i)\mathcal{L}_{n-1,i}(x),$$

where $\mathcal{L}_{n-1,i}(x)$ is the ith Lagrange polynomial for the points $x_1, \ldots, x_n$. It follows that

$$\begin{aligned} I[p_{n-1}] &= \int_a^b p_{n-1}(x)\,dx \\ &= \int_a^b \sum_{i=1}^{n} p_{n-1}(x_i)\mathcal{L}_{n-1,i}(x)\,dx \\ &= \sum_{i=1}^{n} p_{n-1}(x_i)\left(\int_a^b \mathcal{L}_{n-1,i}(x)\,dx\right) \\ &= \sum_{i=1}^{n} p_{n-1}(x_i)w_i \\ &= Q_n[p_{n-1}] \end{aligned}$$

where

$$w_i = \int_a^b \mathcal{L}_{n-1,i}(x)\,dx, \quad i = 1, \ldots, n, \tag{7.6}$$

are the weights of a quadrature rule with nodes $x_1, \ldots, x_n$.

Therefore, *any* n-point quadrature rule with weights chosen as in (7.6) computes $I[f]$ *exactly* when f is a polynomial of degree less than n. For a more general function f, we can use this quadrature rule to approximate $I[f]$ by $I[p_{n-1}]$, where p_{n-1} is the polynomial that interpolates f at the points $x_1, \ldots, x_n$. Quadrature rules that use the weights defined above for given nodes $x_1, \ldots, x_n$ are called **interpolatory** quadrature rules. We say that an interpolatory quadrature rule has **degree of accuracy** n if it integrates polynomials of degree n exactly, but is not exact for polynomials of degree $n+1$.

Exercise 7.2.2 *Use* MATLAB*'s polynomial functions to write a function* `I=polydefint(p,a,b)` *that computes and returns the definite integral of a polynomial with coefficients stored in the vector* `p` *over the interval* $[a, b]$.

Exercise 7.2.3 *Use your function* `polydefint` *from Exercise 7.2.2 to write a function* `w=interpweights(x,a,b)` *that returns a vector of weights* `w` *for an interpolatory quadrature rule for the interval* $[a, b]$ *with nodes stored in the vector* `x`.

Exercise 7.2.4 *Use your function* `interpweights` *from Exercise 7.2.3 to write a function* `I=interpquad(f,a,b,x)` *that approximates* $I[f]$ *over* $[a, b]$ *using an interpolatory quadrature rule with nodes stored in the vector* `x`*. The input argument* `f` *must be a function handle. Test your function by using it to evaluate the integrals of polynomials of various degrees, comparing the results to the exact integrals returned by your function* `polydefint` *from Exercise 7.2.2.*

7.2.3 Sensitivity

To determine the sensitivity of $I[f]$, we define the ∞-norm of a function $f(x)$ by

$$\|f\|_\infty = \max_{x\in[a,b]} |f(x)|$$

and let $\hat{f}$ be a perturbation of f that is also Riemann integrable. Then the absolute condition number of the problem of computing $I[f]$ can be approximated by

$$\begin{aligned}
\frac{|I[\hat{f}] - I[f]|}{\|\hat{f} - f\|_\infty} &= \frac{|I[\hat{f} - f]|}{\|\hat{f} - f\|_\infty} \\
&\leq \frac{I[|\hat{f} - f|]}{\|\hat{f} - f\|_\infty} \\
&\leq \frac{(b-a)\|\hat{f} - f\|_\infty}{\|\hat{f} - f\|_\infty} \\
&\leq (b-a),
\end{aligned}$$

from which it follows that the problem is fairly well-conditioned in most cases. Similarly, perturbations of the endpoints a and b do not lead to large perturbations in $I[f]$, in most cases.

Exercise 7.2.5 *What is the relative condition number of the problem of computing $I[f]$?*

If the weights w_i, $i = 1, \ldots, n$, are nonnegative, then the quadrature rule is stable, as its absolute condition number can be bounded by $(b-a)$, which is the same absolute condition number as the underlying integration problem. However, if any of the weights are negative, then the condition number can be arbitrarily large.

Exercise 7.2.6 *Find the absolute condition number of the problem of computing $Q_n[f]$ for a general quadrature rule of the form (7.5).*

7.3 Newton-Cotes Rules

The family of **Newton-Cotes** quadrature rules consists of interpolatory quadrature rules in which the nodes are equally spaced points within the interval $[a,b]$. The most commonly used Newton-Cotes rules are:

- The **Trapezoidal Rule**, which is a closed rule with two nodes, is defined by

  $$\int_a^b f(x)\,dx \approx \frac{b-a}{2}[f(a) + f(b)].$$

  It is of degree one, and it is based on the principle that the area under $f(x)$ from $x = a$ to $x = b$ can be approximated by the area of a trapezoid with heights $f(a)$ and $f(b)$ and width $b - a$.

- The **Midpoint Rule**, which is an open rule with one node, is defined by

  $$\int_a^b f(x)\,dx \approx (b-a) f\left(\frac{a+b}{2}\right).$$

It is of degree one, and it is based on the principle that the area under $f(x)$ can be approximated by the area of a rectangle with width $b-a$ and height $f(m)$, where $m=(a+b)/2$ is the midpoint of the interval $[a,b]$.

- **Simpson's Rule**, which is a closed rule with three nodes, is defined by

  $$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right].$$

  It is of degree three, and it is derived by computing the integral of the quadratic polynomial that interpolates $f(x)$ at the points a, $(a+b)/2$, and b.

Example 7.3.1 *Let $f(x)=x^3$, $a=0$ and $b=1$. We have*

$$\int_a^b f(x)\,dx = \int_0^1 x^3\,dx = \left.\frac{x^4}{4}\right|_0^1 = \frac{1}{4}.$$

Approximating this integral with the Midpoint Rule yields

$$\int_0^1 x^3\,dx \approx (1-0)\left(\frac{0+1}{2}\right)^3 = \frac{1}{8}.$$

Using the Trapezoidal Rule, we obtain

$$\int_0^1 x^3\,dx \approx \frac{1-0}{2}[0^3+1^3] = \frac{1}{2}.$$

Finally, Simpson's Rule yields

$$\int_0^1 x^3\,dx \approx \frac{1-0}{6}\left[0^3+4\left(\frac{0+1}{2}\right)^3+1^3\right] = \frac{1}{6}\left[0+4\frac{1}{8}+1\right] = \frac{1}{4}.$$

That is, the approximation of the integral by Simpson's Rule is actually exact, which is expected because Simpson's Rule is of degree three. On the other hand, if we approximate the integral of $f(x)=x^4$ from 0 to 1, Simpson's Rule yields $5/24$, while the exact value is $1/5$. Still, this is a better approximation than those obtained using the Midpoint Rule (1/16) or the Trapezoidal Rule (1/2). □

Exercise 7.3.1 *Write* MATLAB *functions* `I=quadmidpoint(f,a,b)`, `I=quadtrapezoidal(f,a,b)` *and* `I=quadsimpsons(f,a,b)` *that implement the Midpoint Rule, Trapezoidal Rule and Simpson's Rule, respectively, to approximate the integral of $f(x)$, implemented by the function handle* `f`*, over the interval $[a,b]$.*

Exercise 7.3.2 *Use your code from Exercise 7.2.4 to write a function* `I=quadnewtoncotes(f,a,b,n)` *to integrate $f(x)$, implemented by the function handle* `f`*, over $[a,b]$ using an n-node Newton-Cotes rule.*

7.3.1 Error Analysis

The error in any interpolatory quadrature rule defined on an interval $[a, b]$, such as a Newton-Cotes rule or a Clenshaw-Curtis rule can be obtained by computing the integral from a to b of the error in the polynomial interpolant on which the rule is based.

For the Trapezoidal Rule, which is obtained by integrating a linear polynomial that interpolates the integrand $f(x)$ at $x = a$ and $x = b$, this approach to error analysis yields

$$\int_a^b f(x)\,dx - \frac{b-a}{2}[f(a) + f(b)] = \int_a^b \frac{f''(\xi(x))}{2}(x-a)(x-b)\,dx,$$

where $\xi(x)$ lies in $[a, b]$ for $a \leq x \leq b$. The function $(x-a)(x-b)$ does not change sign on $[a, b]$, which allows us to apply the Weighted Mean Value Theorem for Integrals and obtain a more useful expression for the error,

$$\int_a^b f(x)\,dx - \frac{b-a}{2}[f(a) + f(b)] = \frac{f''(\eta)}{2}\int_a^b (x-a)(x-b)\,dx = -\frac{f''(\eta)}{12}(b-a)^3, \tag{7.7}$$

where $a \leq \eta \leq b$. Because the error depends on the second derivative, it follows that the Trapezoidal Rule is exact for any linear function.

A similar approach can be used to obtain expressions for the error in the Midpoint Rule and Simpson's Rule, although the process is somewhat more complicated due to the fact that the functions $(x-m)$, for the Midpoint Rule, and $(x-a)(x-m)(x-b)$, for Simpson's Rule, where in both cases $m = (a+b)/2$, change sign on $[a, b]$, thus making the Weighted Mean Value Theorem for Integrals impossible to apply in the same straightforward manner as it was for the Trapezoidal Rule.

We instead use the following approach, illustrated for the Midpoint Rule and adapted from a similar proof for Simpson's Rule from [36]. We assume that f is twice continuously differentiable on $[a, b]$. First, we make a change of variable

$$x = \frac{a+b}{2} + \frac{b-a}{2}t, \quad t \in [-1, 1],$$

to map the interval $[-1, 1]$ to $[a, b]$, and then define $F(t) = f(x(t))$. The error in the Midpoint Rule is then given by

$$\int_a^b f(x)\,dx - (b-a)f\left(\frac{a+b}{2}\right) = \frac{b-a}{2}\left[\int_{-1}^1 F(\tau)\,d\tau - 2F(0)\right].$$

We now define

$$G(t) = \int_{-t}^t F(\tau)\,d\tau - 2tF(0).$$

It is easily seen that the error in the Midpoint Rule is $\frac{1}{2}(b-a)G(1)$. We then define

$$H(t) = G(t) - t^3 G(1).$$

Because $H(0) = H(1) = 0$, it follows from Rolle's Theorem that there exists a point $\xi_1 \in (0, 1)$ such that $H'(\xi_1) = 0$. However, from

$$H'(0) = G'(0) = [F(t) + F(-t)]|_{t=0} - 2F(0) = 2F(0) - 2F(0) = 0,$$

it follows from Rolle's Theorem that there exists a point $\xi_2 \in (0,1)$ such that $H''(\xi_2) = 0$.

From

$$H''(t) = G''(t) - 6tG(1) = F'(t) - F'(-t) - 6tG(1),$$

and the Mean Value Theorem, we obtain, for some $\xi_3 \in (-1,1)$,

$$0 = H''(\xi_2) = 2\xi_2 F''(\xi_3) - 6\xi_2 G(1),$$

or

$$G(1) = \frac{1}{3}F''(\xi_3) = \frac{1}{3}\left(\frac{b-a}{2}\right)^2 f''(x(\xi_3)).$$

Multiplying by $(b-a)/2$ yields the error in the Midpoint Rule.

The result of the analysis is that for the Midpoint Rule,

$$\int_a^b f(x)\,dx - (b-a)f\left(\frac{a+b}{2}\right) = \frac{f''(\eta)}{24}(b-a)^3, \tag{7.8}$$

and for Simpson's Rule,

$$\int_a^b f(x)\,dx - \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] = -\frac{f^{(4)}(\eta)}{90}\left(\frac{b-a}{2}\right)^5, \tag{7.9}$$

where, in both cases, η is some point in $[a,b]$.

It follows that the Midpoint Rule is exact for any linear function, just like the Trapezoidal Rule, even though it uses one less interpolation point, because of the cancellation that results from choosing the midpoint of $[a,b]$ as the interpolation point. Similar cancellation causes Simpson's Rule to be exact for polynomials of degree three or less, even though it is obtained by integrating a quadratic interpolant over $[a,b]$.

Exercise 7.3.3 *Adapt the approach in the preceding derivation of the error formula (7.8) for the Midpoint Rule to obtain the error formula (7.9) for Simpson's Rule.*

In general, the degree of accuracy of Newton-Cotes rules can easily be determined by expanding the integrand $f(x)$ in a Taylor series around the midpoint of $[a,b]$, $m = (a+b)/2$. This technique can be used to show that n-point Newton-Cotes rules with an odd number of nodes have degree n, which is surprising since, in general, interpolatory n-point quadrature rules have degree $n-1$. This extra degree of accuracy is due to the cancellation of the high-order error terms in the Taylor expansion used to determine the error. Such cancellation does not occur with Newton-Cotes rules that have an even number of nodes.

Exercise 7.3.4 *Prove the statement from the preceding paragraph: a n-node Newton-Cotes rule has degree of accuracy n if n is odd, and $n-1$ if n is even.*

7.3.2 Higher-Order Rules

Unfortunately, Newton-Cotes rules are not practical when the number of nodes is large, due to the inaccuracy of high-degree polynomial interpolation using equally spaced points. Furthermore, for $n \geq 11$, n-point Newton-Cotes rules have at least one negative weight, and therefore such rules can be ill-conditioned. This can be seen by revisiting *Runge's Example* from Section 5.4, and attempting to approximate

$$\int_{-5}^{5} \frac{1}{1+x^2}\,dx \tag{7.10}$$

using a Newton-Cotes rule. As n increases, the approximate integral does not converge to the exact result; in fact, it increases without bound.

Exercise 7.3.5 *What is the smallest value of n for which a n-node Newton-Cotes rule has a negative weight?*

Exercise 7.3.6 *Use your code from Exercise 7.3.2 to evaluate the integral from (7.10) for increasing values of n and describe the behavior of the error as n increases.*

7.4 Composite Rules

When using a quadrature rule to approximate $I[f]$ on some interval $[a,b]$, the error is proportional to h^r, where $h = b - a$ and r is some positive integer. Therefore, if the interval $[a,b]$ is large, it is advisable to divide $[a,b]$ into smaller intervals, use a quadrature rule to compute the integral of f on each subinterval, and add the results to approximate $I[f]$. Such a scheme is called a **composite quadrature rule**.

It can be shown that the approximate integral obtained using a composite rule that divides $[a,b]$ into n subintervals will converge to $I[f]$ as $n \to \infty$, provided that the maximum width of the n subintervals approaches zero, and the quadrature rule used on each subinterval has a degree of at least zero. It should be noted that using closed quadrature rules on each subinterval improves efficiency, because the nodes on the endpoints of each subinterval, except for a and b, are shared by two quadrature rules. As a result, fewer function evaluations are necessary, compared to a composite rule that uses open rules with the same number of nodes.

We will now state some of the most well-known composite quadrature rules. In the following discussion, we assume that the interval $[a,b]$ is divided into n subintervals of equal width $h = (b-a)/n$, and that these subintervals have endpoints $[x_{i-1}, x_i]$, where $x_i = a + ih$, for $i = 0, 1, 2, \ldots, n$. Given such a partition of $[a,b]$, we can compute $I[f]$ using

- the **Composite Midpoint Rule**

$$\int_a^b f(x)\,dx \approx 2h[f(x_1) + f(x_3) + \cdots + f(x_{n-1})], \quad n \text{ is even}, \tag{7.11}$$

- the **Composite Trapezoidal Rule**

$$\int_a^b f(x)\,dx \approx \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)], \tag{7.12}$$

  or

- the **Composite Simpson's Rule**

$$\int_a^b f(x)\,dx \approx \frac{h}{3}[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+\cdots+2f(x_{n-2})+4f(x_{n-1})+f(x_n)], \tag{7.13}$$

  for which n is required to be even, as in the Composite Midpoint Rule.

Exercise 7.4.1 *Write* Matlab *functions* `I=quadcompmidpt(f,a,b,n)`, `I=quadcomptrap(f,a,b,n)` *and* `I=quadcompsimp(f,a,b,n)` *that implement the Composite Midpoint rule (7.11), Composite Trapezoidal Rule (7.12), and Composite Simpson's Rule (7.13), respectively, to approximate the integral of $f(x)$, implemented by the function handle* `f`*, over $[a,b]$ with $n+1$ nodes $x_0, x_1, \ldots, x_n$.*

Exercise 7.4.2 *Apply your functions from Exercise 7.4.1 to approximate the integrals*

$$\int_0^1 \sqrt{dx}\,dx, \quad \int_1^2 \sqrt{x}\,dx.$$

Use different values of n, the number of subintervals. How does the accuracy increase as n increases? Explain any discrepancy between the observed behavior and theoretical expectations.

7.4.1 Error Analysis

To obtain the error in each of these composite rules, we can sum the errors in the corresponding basic rules over the n subintervals. For the Composite Trapezoidal Rule, this yields

$$\begin{aligned} E_{trap} &= \int_a^b f(x)\,dx - \frac{h}{2}\left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right] \\ &= -\sum_{i=1}^{n} \frac{f''(\eta_i)}{12}(x_i - x_{i-1})^3 \\ &= -\frac{h^3}{12}\sum_{i=1}^{n} f''(\eta_i) \\ &= -\frac{h^3}{12} n f''(\eta) \\ &= -\frac{f''(\eta)}{12} nh \cdot h^2 \\ &= -\frac{f''(\eta)}{12}(b-a)h^2, \end{aligned} \tag{7.14}$$

where, for $i = 1, \ldots, n$, η_i belongs to $[x_{i-1}, x_i]$, and $a \leq \eta \leq b$. The replacement of $\sum_{i=1}^{h} f''(\eta_i)$ by $nf''(\eta)$ is justified by the Intermediate Value Theorem, provided that $f''(x)$ is continuous on $[a,b]$. We see that the Composite Trapezoidal Rule is *second-order accurate*. Furthermore, its *degree of accuracy*, which is the highest degree of polynomial that is guaranteed to be integrated exactly, is the same as for the basic Trapezoidal Rule, which is one.

Similarly, for the Composite Midpoint Rule, we have

$$E_{mid} = \int_a^b f(x)\,dx - 2h\sum_{i=1}^{n/2} f(x_{2i-1}) = \sum_{i=1}^{n/2} \frac{f''(\eta_i)}{24}(2h)^3 = \frac{f''(\eta)}{6}(b-a)h^2.$$

Although it appears that the Composite Midpoint Rule is less accurate than the Composite Trapezoidal Rule, it should be noted that it uses about half as many function evaluations. In other words, the Basic Midpoint Rule is applied $n/2$ times, each on a subinterval of width $2h$. Rewriting the Composite Midpoint Rule in such a way that it uses n function evaluations, each on a subinterval of width h, we obtain

$$\int_a^b f(x)\,dx = h\sum_{i=1}^{n} f\left(x_{i-1} + \frac{h}{2}\right) + \frac{f''(\eta)}{24}(b-a)h^2, \tag{7.15}$$

which reveals that the Composite Midpoint Rule is generally more accurate.

Finally, for the Composite Simpson's Rule, we have

$$E_{simp} = -\sum_{i=1}^{n/2} \frac{f^{(4)}(\eta_i)}{90}h^5 = -\frac{f^{(4)}(\eta)}{180}(b-a)h^4, \tag{7.16}$$

because the Basic Simpson Rule is applied $n/2$ times, each on a subinterval of width $2h$. We conclude that the Simpson's Rule is *fourth-order accurate.*

Exercise 7.4.3 *Derive the error formula (7.16) for the Composite Simpson's Rule (7.13).*

Example 7.4.1 *We wish to approximate*

$$\int_0^1 e^x\,dx$$

using composite quadrature, to 3 decimal places. That is, the error must be less than 10^{-3}*. This requires choosing the number of subintervals,* n*, sufficiently large so that an upper bound on the error is less than* 10^{-3}*.*

For the Composite Trapezoidal Rule, the error is

$$E_{trap} = -\frac{f''(\eta)}{12}(b-a)h^2 = -\frac{e^\eta}{12n^2},$$

since $f(x) = e^x$*,* $a = 0$ *and* $b = 1$*, which yields* $h = (b-a)/n = 1/n$*. Since* $0 \leq \eta \leq 1$*, and* e^x *is increasing, the factor* e^η *is bounded above by* $e^1 = e$*. It follows that* $|E_{trap}| < 10^{-3}$ *if*

$$\frac{e}{12n^2} < 10^{-3} \quad \Rightarrow \quad \frac{1000e}{12} < n^2 \quad \Rightarrow \quad n > 15.0507.$$

Therefore, the error will be sufficiently small provided that we choose $n \geq 16$*.*

On the other hand, if we use the Composite Simpson's Rule, the error is

$$E_{simp} = -\frac{f^{(4)}(\eta)}{180}(b-a)h^4 = -\frac{e^\eta}{180n^4}$$

for some η *in* $[0,1]$*, which is less than* 10^{-3} *in absolute value if*

$$n > \left(\frac{1000e}{180}\right)^{1/4} \approx 1.9713,$$

so $n = 2$ *is sufficient. That is, we can approximate the integral to 3 decimal places by setting* $h = (b-a)/n = (1-0)/2 = 1/2$ *and computing*

$$\int_0^1 e^x\,dx \approx \frac{h}{3}[e^{x_0} + 4e^{x_1} + e^{x_2}] = \frac{1/2}{3}[e^0 + 4e^{1/2} + e^1] \approx 1.71886,$$

whereas the exact value is approximately 1.71828. □

7.5 Gaussian Quadrature

Previously, we learned that a Newton-Cotes quadrature rule with n nodes has degree at most n. Therefore, it is natural to ask whether it is possible to select the nodes and weights of an n-point quadrature rule so that the rule has degree greater than n. **Gaussian quadrature rules** [1] have the surprising property that they can be used to integrate polynomials of degree $2n-1$ exactly using only n nodes.

7.5.1 Direct Construction

Gaussian quadrature rules can be constructed using a technique known as **moment matching**, or **direct construction**. For any nonnegative integer k, the kth moment is defined to be

$$\mu_k = \int_a^b x^k\,dx.$$

For given n, our goal is to select weights and nodes so that the first $2n$ moments are computed exactly; i.e.,

$$\mu_k = \sum_{i=1}^n w_i x_i^k, \quad k = 0, 1, \ldots, 2n-1. \tag{7.17}$$

Since we have $2n$ free parameters, it is reasonable to think that appropriate nodes and weights can be found. Unfortunately, this system of equations is nonlinear, so it can be quite difficult to solve.

Exercise 7.5.1 *Use direct construction to solve the equations (7.17) for the case of* $n = 2$ *on the interval* $(a, b) = (-1, 1)$ *for the nodes* x_1, x_2 *and weights* w_1, w_2.

7.5.2 Orthogonal Polynomials

Suppose $g(x)$ is a polynomial of degree at most $2n-1$. For convenience, we will write $g \in \mathcal{P}_{2n-1}$, where, for any natural number k, $\mathcal{P}_k$ denotes the space of polynomials of degree at most k. We shall show that there exist nodes $\{x_i\}_{i=1}^n$ and nodes $\{w_i\}_{i=1}^n$ such that

$$I[g] = \int_a^b g(x)\,dx = \sum_{i=1}^n w_i g(x_i).$$

Furthermore, for more general functions, $G(x)$,

$$\int_a^b G(x)\,dx = \sum_{i=1}^n w_i G(x_i) + E[G]$$

where

1. x_i are real, distinct, and $a < x_i < b$ for $i = 1, 2, \ldots, n$.
2. The weights $\{w_i\}$ satisfy $w_i > 0$ for $i = 1, 2, \ldots, n$.
3. The error $E[G]$ satisfies $E[G] = \frac{G^{(2n)}(\xi)}{(2n)!} \int_a^b \prod_{i=1}^n (x - x_i)^2\,dx$.

Notice that this method is exact for polynomials of degree $2n-1$ since the error functional $E[G]$ depends on the $(2n)^{th}$ derivative of G.

To prove this, we shall construct an *orthonormal family* of polynomials $\{q_i(x)\}_{i=0}^n$, as in Section 6.2, so that

$$\langle q_r, q_s\rangle = \int_a^b q_r(x)q_s(x)\,dx = \begin{cases} 0 & r \neq s, \\ 1 & r = s. \end{cases}$$

Recall that this can be accomplished using the fact that such a family of polynomials satisfies a *three-term recurrence relation*

$$\beta_j q_j(x) = (x-\alpha_j)q_{j-1}(x) - \beta_{j-1}q_{j-2}(x), \quad q_0(x) = (b-a)^{-1/2}, \quad q_{-1}(x) = 0,$$

where

$$\alpha_j = \langle q_{j-1}, xq_{j-1}\rangle = \int_a^b xq_{j-1}(x)^2\,dx, \quad \beta_j^2 = \langle q_j, xq_{j-1}\rangle = \int_a^b xq_j(x)q_{j-1}(x)\,dx, \quad j \geq 1,$$

with $\beta_0^2 = b-a$.

We choose the nodes $\{x_i\}$ to be the roots of the nth-degree polynomial in this family, which are real, distinct and lie within (a,b), as proved in Section 6.2.6. Next, we construct the interpolant of degree $n-1$, denoted $p_{n-1}(x)$, of $g(x)$ through the nodes:

$$p_{n-1}(x) = \sum_{i=1}^n g(x_i)\mathcal{L}_{n-1,i}(x),$$

where, for $i = 1,\ldots,n$, $\mathcal{L}_{n-1,i}(x)$ is the ith Lagrange polynomial for the points $x_1,\ldots,x_n$. We shall now look at the interpolation error function

$$e(x) = g(x) - p_{n-1}(x).$$

Clearly, since $g \in \mathcal{P}_{2n-1}$, $e \in \mathcal{P}_{2n-1}$. Since $e(x)$ has roots at each of the roots of $q_n(x)$, we can factor e so that

$$e(x) = q_n(x)r(x),$$

where $r \in \mathcal{P}_{n-1}$. It follows from the fact that $q_n(x)$ is orthogonal to *any* polynomial in $\mathcal{P}_{n-1}$ that the integral of g can then be written as

$$\begin{aligned} I[g] &= \int_a^b p_{n-1}(x)\,dx + \int_a^b q_n(x)r(x)\,dx \\ &= \int_a^b p_{n-1}(x)\,dx \\ &= \int_a^b \sum_{i=1}^n g(x_i)\mathcal{L}_{n-1,i}(x)\,dx \\ &= \sum_{i=1}^n g(x_i)\int_a^b \mathcal{L}_{n-1,i}(x)\,dx \\ &= \sum_{i=1}^n g(x_i)w_i \end{aligned}$$

where

$$w_i = \int_a^b \mathcal{L}_{n-1,i}(x)\,dx, \quad i = 1, 2, \ldots, n.$$

For a more general function $G(x)$, the error functional $E[G]$ can be obtained from the expression for Hermite interpolation error presented in Section 5.5.1, as we will now investigate.

7.5.3 Error Analysis

It is easy to show that the weights w_i are positive. Since the interpolation basis functions $\mathcal{L}_{n-1,i}$ belong to $\mathcal{P}_{n-1}$, it follows that $\mathcal{L}^2_{n-1,i} \in \mathcal{P}_{2n-2}$, and therefore

$$0 < \int_a^b \mathcal{L}^2_{n-1,i}(x)\,dx = \sum_{j=0}^{n-1} w_j \mathcal{L}^2_{n-1,i}(x_j) = w_i.$$

Note that we have thus obtained an alternative formula for the weights.

This formula also arises from an alternative approach to constructing Gaussian quadrature rules, from which a representation of the error can easily be obtained. We construct the Hermite interpolating polynomial $G_{2n-1}(x)$ of $G(x)$, using the Gaussian quadrature nodes as interpolation points, that satisfies the $2n$ conditions

$$G_{2n-1}(x_i) = G(x_i), \quad G'_{2n-1}(x_i) = G'(x_i), \quad i = 1, 2, \ldots, n.$$

We recall from Section 5.5.1 that this interpolant has the form

$$G_{2n-1}(x) = \sum_{i=1}^{n} G(x_i)H_i(x) + \sum_{i=1}^{n} G'(x_i)K_i(x),$$

where, as in our previous discussion of Hermite interpolation,

$$H_i(x_j) = \delta_{ij}, \quad H'_i(x_j) = 0, \quad K_i(x_j) = 0, \quad K'_i(x_j) = \delta_{ij}, \quad i, j = 1, 2, \ldots, n.$$

Then, we have

$$\int_a^b G_{2n-1}(x)\,dx = \sum_{i=1}^{n} G(x_i) \int_a^b H_i(x)\,dx + \sum_{i=1}^{n} G'(x_i) \int_a^b K_i(x)\,dx.$$

We recall from Section 5.5.1 that

$$H_i(x) = \mathcal{L}_{n-1,i}(x)^2[1 - 2\mathcal{L}'_{n-1,i}(x_i)(x - x_i)], \quad K_i(x) = \mathcal{L}_{n-1,i}(x)^2(x - x_i), \quad i = 1, 2, \ldots, n,$$

and for convenience, we define

$$\pi_n(x) = (x - x_1)(x - x_2)\cdots(x - x_n),$$

and note that

$$\mathcal{L}_{n-1,i}(x) = \frac{\pi_n(x)}{(x - x_i)\pi'_n(x_i)}, \quad i = 1, 2, \ldots, n.$$

We then have

$$\begin{aligned}
\int_a^b H_i(x)\,dx &= \int_a^b \mathcal{L}_{n-1,i}(x)^2\,dx - 2\mathcal{L}'_{n-1,i}(x_i)\int_a^b \mathcal{L}_{n-1,i}(x)^2(x-x_i)\,dx \\
&= \int_a^b \mathcal{L}_{n-1,i}(x)^2\,dx - \frac{2\mathcal{L}'_{n-1,i}(x_i)}{\pi'_n(x_i)}\int_a^b \mathcal{L}_{n-1,i}(x)\pi_n(x)\,dx \\
&= \int_a^b \mathcal{L}_{n-1,i}(x)^2\,dx,
\end{aligned}$$

as the second term vanishes because $\mathcal{L}_{n-1,i}(x)$ is of degree $n-1$, and $\pi_n(x)$, a polynomial of degree n, is orthogonal to all polynomials of lesser degree.

Similarly,

$$\int_a^b K_i(x)\,dx = \int_a^b \mathcal{L}_{n-1,i}(x)^2(x-x_i)\,dx = \frac{1}{\pi'_n(x_i)}\int_a^b \mathcal{L}_{n-1,i}(x)\pi_n(x)\,dx = 0.$$

We conclude that

$$\int_a^b G_{2n-1}(x)\,dx = \sum_{i=1}^n G(x_i)w_i,$$

where, as before,

$$w_i = \int_a^b \mathcal{L}_{n-1,i}(x)^2\,dx = \int_a^b \mathcal{L}_{n-1,i}(x)\,dx, \quad i = 1, 2, \ldots, n.$$

The equivalence of these formulas for the weights can be seen from the fact that the difference $\mathcal{L}_{n-1,i}(x)^2 - \mathcal{L}_{n-1,i}(x)$ is a polynomial of degree $2n-2$ that is divisible by $\pi_n(x)$, because it vanishes at all of the nodes. The quotient, a polynomial of degree $n-2$, is orthogonal to $\pi_n(x)$. Therefore, the integrals of $\mathcal{L}_{n-1,i}(x)^2$ and $\mathcal{L}_{n-1,i}(x)$ must be equal.

We now use the error in the Hermite interpolating polynomial to obtain

$$\begin{aligned}
E[G] &= \int_a^b G(x)\,dx - \sum_{i=1}^n G(x_i)w_i \\
&= \int_a^b [G(x) - G_{2n-1}(x)]\,dx \\
&= \int_a^b \frac{G^{(2n)}(\xi(x))}{(2n)!}\pi_n(x)^2\,dx \\
&= \frac{G^{(2n)}(\xi)}{(2n)!}\int_a^b \prod_{i=1}^n (x-x_i)^2\,dx,
\end{aligned}$$

where $\xi \in (a,b)$. The last step is obtained using the Weighted Mean Value Theorem for Integrals, which applies because $\pi_n(x)^2$ does not change sign.

In addition to this error formula, we can easily obtain qualitative bounds on the error. For instance, if we know that the even derivatives of g are positive, then we know that the quadrature rule yields a lower bound for $I[g]$. Similarly, if the even derivatives of g are negative, then the quadrature rule gives an upper bound.

Finally, it can be shown that as $n \to \infty$, the n-node Gaussian quadrature approximation of $I[f]$ converges to $I[f]$. The key to the proof is the fact that the weights are guaranteed to be positive, and therefore the sum of the weights is always equal to $b-a$. Such a result does not hold for n-node Newton-Cotes quadrature rules, because the sum of the absolute values of the weights cannot be bounded, due to the presence of negative weights.

Example 7.5.1 *We will use* Gaussian quadrature *to approximate the integral*

$$\int_0^1 e^{-x^2}\,dx.$$

The particular Gaussian quadrature rule that we will use consists of 5 nodes x_1, x_2, x_3, x_4 *and* x_5*, and 5 weights* w_1, w_2, w_3, w_4 *and* w_5*. To determine the proper nodes and weights, we use the fact that the nodes and weights of a 5-point Gaussian rule for integrating over the interval* $[-1, 1]$ *are given by*

i	*Nodes* $r_{5,i}$	*Weights* $c_{5,i}$
1	0.9061798459	0.2369268850
2	0.5384693101	0.4786286705
3	0.0000000000	0.5688888889
4	−0.5384693101	0.4786286705
5	−0.9061798459	0.2369268850

To obtain the corresponding nodes and weights for integrating over $[0, 1]$*, we can use the fact that in general,*

$$\int_a^b f(x)\,dx = \int_{-1}^1 f\left(\frac{b-a}{2}t + \frac{a+b}{2}\right)\frac{b-a}{2}\,dt,$$

as can be shown using the change of variable $x = [(b-a)/2]t + (a+b)/2$ *that maps* $[a, b]$ *into* $[-1, 1]$*. We then have*

$$\begin{aligned} \int_a^b f(x)\,dx &= \int_{-1}^1 f\left(\frac{b-a}{2}t + \frac{a+b}{2}\right)\frac{b-a}{2}\,dt \\ &\approx \sum_{i=1}^5 f\left(\frac{b-a}{2}r_{5,i} + \frac{a+b}{2}\right)\frac{b-a}{2}c_{5,i} \\ &\approx \sum_{i=1}^5 f(x_i)w_i, \end{aligned}$$

where

$$x_i = \frac{b-a}{2}r_{5,i} + \frac{a+b}{2}, \quad w_i = \frac{b-a}{2}c_{5,i}, \quad i = 1, \ldots, 5.$$

In this example, $a = 0$ *and* $b = 1$*, so the nodes and weights for a 5-point Gaussian quadrature rule for integrating over* $[0, 1]$ *are given by*

$$x_i = \frac{1}{2}r_{5,i} + \frac{1}{2}, \quad w_i = \frac{1}{2}c_{5,i}, \quad i = 1, \ldots, 5,$$

which yields

i	*Nodes* x_i	*Weights* w_i
1	0.95308992295	0.11846344250
2	0.76923465505	0.23931433525
3	0.50000000000	0.28444444444
4	0.23076534495	0.23931433525
5	0.04691007705	0.11846344250

It follows that

$$\begin{aligned}\int_0^1 e^{-x^2}\,dx &\approx \sum_{i=1}^{5} e^{-x_i^2} w_i \\ &\approx 0.11846344250e^{-0.95308992295^2} + 0.23931433525e^{-0.76923465505^2} + \\ &\ 0.28444444444e^{-0.5^2} + 0.23931433525e^{-0.23076534495^2} + \\ &\ 0.11846344250e^{-0.04691007705^2} \\ &\approx 0.74682412673352.\end{aligned}$$

Since the exact value is 0.74682413281243, the absolute error is -6.08×10^{-9}*, which is remarkably accurate considering that only fives nodes are used.* □

The high degree of accuracy of Gaussian quadrature rules make them the most commonly used rules in practice. However, they are not without their drawbacks:

- They are not progressive, so the nodes must be recomputed whenever additional degrees of accuracy are desired. An alternative is to use **Gauss-Kronrod rules** [22]. A $(2n+1)$-point Gauss-Kronrod rule uses the nodes of the n-point Gaussian rule. For this reason, practical quadrature procedures use both the Gaussian rule and the corresponding Gauss-Kronrod rule to estimate accuracy.

- Because the nodes are the roots of a polynomial, they must be computed using traditional root-finding methods, which are not always accurate. Errors in the computed nodes lead to lost degrees of accuracy in the approximate integral. In practice, however, this does not normally cause significant difficulty.

Exercise 7.5.2 *Write a* MATLAB *function* `I=gaussquadrule(f,a,b,n)` *that approximate the integral of* $f(x)$*, implemented by the function handle* `f`*, over* $[a,b]$ *with a* n*-node Gaussian quadrature rule. Use your function* `interpquad` *from Exercise 7.2.4 as well as your function* `makelegendre` *from Section 6.2. Test your function by comparing its output to that of the built-in function* `integral`*. How does its accuracy compare to that of your function* `quadnewtoncotes` *from Exercise 7.3.2?*

Exercise 7.5.3 *A* 5*-node Gaussian quadrature rule is exact for the integrand* $f(x) = x^8$*, while a 5-node Newton-Cotes rule is not. How important is it that the Gaussian quadrature nodes be computed with high accuracy? Investigate this by approximating* $\int_{-1}^{1} x^8\,dx$ *using a 5-node interpolatory quadrature rule with nodes*

$$x_i = \theta\tilde{x}_i + (1-\theta)\hat{x}_i,$$

where $\{\tilde{x}_i\}_{i=1}^5$ *and* $\{\hat{x}_i\}_{i=1}^5$ *are the nodes for a 5-node Gaussian and Newton-Cotes rule, respectively, and* $\theta \in [0,1]$*. Use your function* `interpquad` *from Exercise 7.2.4 and let* θ *vary from 0 to 1. How does the error behave as* θ *increases?*

7.5.4 Other Weight Functions

In Section 6.2 we learned how to construct sequences of orthogonal polynomials for the inner product

$$\langle f, g\rangle = \int_a^b f(x)g(x)w(x)\,dx,$$

where f and g are real-valued functions on (a,b) and $w(x)$ is a *weight function* satisfying $w(x) > 0$ on (a,b). These orthogonal polynomials can be used to construct Gauss quadrature rules for integrals with weight functions, in a similar manner to how they were constructed earlier in this section for the case $w(x) \equiv 1$.

Exercise 7.5.4 *Let* $w(x)$ *be a weight function satisfying* $w(x) > 0$ *in* (a,b)*. Derive a Gauss quadrature rule of the form*

$$\int_a^b f(x)w(x)\,dx = \sum_{i=1}^n f(x_i)w_i + E[f]$$

that is exact for $f \in \mathcal{P}_{2n-1}$*. What is the error functional* $E[f]$*?*

A case of particular interest is the interval $(-1,1)$ with the weight function $w(x) = (1-x^2)^{-1/2}$, as the orthogonal polynomials for the corresponding inner product are the *Chebyshev polynomials*, introduced in Section 5.4.2. Unlike other Gauss quadrature rules, there are simple formulas for the nodes and weights in this case.

Exercise 7.5.5 *Use trigonometric identities to prove that*

$$\cos\theta + \cos 3\theta + \cos 5\theta + \cdots + \cos(2n-1)\theta = \frac{\sin 2n\theta}{\sin\theta},$$

when θ *is not an integer multiple of* π*.*

Exercise 7.5.6 *Use the result of Exercise 7.5.5 and direct construction to derive the nodes and weights for an n-node Gaussian quadrature rule of the form*

$$\int_{-1}^{1} (1-x^2)^{-1/2} f(x)\,dx = \sum_{i=1}^{n} f(x_i)w_i,$$

that is exact when $f \in \mathcal{P}_{2n-1}$.

7.5.5 Prescribing Nodes

We have seen that for an integrand $f(x)$ that has even derivatives that do not change sign on (a,b), it can be determined that the Gauss quadrature approximation of $I[f]$ is either an upper bound or a lower bound. By prescribing either or both of the endpoints $x=a$ or $x=b$ as quadrature nodes, we can obtain additional bounds and bracket the exact value of $I[f]$. However, it is important to prescribe such nodes in a manner that, as much as possible, maintains the high degree of accuracy of the quadrature rule.

A Gauss quadrature rule with $n+1$ nodes is exact for any integrand in $\mathcal{P}_{2n+1}$. Our goal is to construct a quadrature rule with $n+1$ nodes, one of which is at $x=a$, that computes

$$I[f] = \int_a^b f(x)w(x)\,dx,$$

for a given weight function $w(x)$, exactly when $f \in \mathcal{P}_{2n}$. That is, prescribing a node reduces the degree of accuracy by only one. Such a quadrature rule is called a **Gauss-Radau quadrature rule** [1].

We begin by dividing $f(x)$ by $(x-a)$, which yields

$$f(x) = (x-a)q_{2n-1}(x) + f(a).$$

We then construct a n-node Gauss quadrature rule

$$\int_a^b g(x)w^*(x)\,dx = \sum_{i=1}^{n} g(x_i^*)w_i^* + E[g]$$

for the weight function $w^*(x) = (x-a)w(x)$. It is clear that $w^*(x) > 0$ on (a,b). Because this rule is exact for $g \in \mathcal{P}_{2n-1}$, we have

$$\begin{aligned} I[f] &= \int_a^b q_{2n-1}(x)w^*(x)\,dx + f(a)\int_a^b w(x)\,dx \\ &= \sum_{i=1}^{n} q_{2n-1}(x_i^*)w_i^* + f(a)\int_a^b w(x)\,dx \\ &= \sum_{i=1}^{n} \frac{f(x_i^*)-f(a)}{x_i^*-a}w_i^* + f(a)\int_a^b w(x)\,dx \\ &= \sum_{i=1}^{n} f(x_i^*)w_i + f(a)\left[\int_a^b w(x)\,dx - \sum_{i=1}^{n} w_i\right], \end{aligned}$$

where $w_i = w_i^*/(x_i^* - a)$ for $i = 1, 2, \ldots, n$. By defining

$$w_0 = \int_a^b w(x)\,dx - \sum_{i=1}^n w_i,$$

and defining the nodes

$$x_0 = a, \quad x_i = x_i^*, \quad i = 1, 2, \ldots, n,$$

we obtain a quadrature rule

$$I[f] = \sum_{i=0}^n f(x_i)w_i + E[f]$$

that is exact for $f \in \mathcal{P}_{2n}$. Clearly the weights $w_1, w_2, \ldots, w_n$ are positive; it can be shown that $w_0 > 0$ by noting that it is the error in the Gauss quadrature approximation of

$$I^*\left[\frac{1}{x-a}\right] = \int_a^b \frac{1}{x-a} w^*(x)\,dx.$$

It can also be shown that if the integrand $f(x)$ is sufficiently differentiable and satisfies $f^{(2n)} > 0$ on (a, b), then this Gauss-Radau rule yields a lower bound for $I[f]$.

Exercise 7.5.7 *Following the discussion above, derive a Gauss-Radau quadrature rule in which a node is prescribed at $x = b$. Prove that the weights $w_1, w_2, \ldots, w_{n+1}$ are positive. Does this rule yield an upper bound or lower bound for the integrand $f(x) = 1/(x - a)$?*

Exercise 7.5.8 *Derive formulas for the nodes and weights for a* **Gauss-Lobatto quadrature rule** *[1], in which nodes are prescribed at $x = a$* and *$x = b$. Specifically, the rule must have $n + 1$ nodes $x_0 = a < x_1 < x_2 < \cdots < x_{n-1} < x_n = b$. Prove that the weights $w_0, w_1, \ldots, w_n$ are positive. What is the degree of accuracy of this rule?*

Exercise 7.5.9 *Explain why developing a Gauss-Radau rule by prescribing a node at $x = c$, where $c \in (a, b)$, is problematic.*

7.6 Extrapolation to the Limit

We have seen that the accuracy of methods for computing integrals or derivatives of a function $f(x)$ depends on the spacing between points at which f is evaluated, and that the approximation tends to the exact value as this spacing tends to 0.

Suppose that a uniform spacing h is used. We denote by $F(h)$ the approximation computed using the spacing h, from which it follows that the exact value is given by $F(0)$. Let p be the order of accuracy in our approximation; that is,

$$F(h) = a_0 + a_1h^p + O(h^r), \quad r > p, \tag{7.18}$$

where a_0 is the exact value $F(0)$. Then, if we choose a value for h and compute $F(h)$ and $F(h/q)$ for some positive integer q, then we can neglect the $O(h^r)$ terms and solve a system of two equations for the unknowns a_0 and a_1, thus obtaining an approximation that is rth order accurate. If we can describe the error in this approximation in the same way that we can describe the error in our original approximation $F(h)$, we can repeat this process to obtain an approximation that is even more accurate.

7.6.1 Richardson Extrapolation

This process of extrapolating from $F(h)$ and $F(h/q)$ to approximate $F(0)$ with a higher order of accuracy is called **Richardson extrapolation** [31]. In a sense, Richardson extrapolation is similar in spirit to Aitken's Δ^2 method (see Section 8.5), as both methods use assumptions about the convergence of a sequence of approximations to "solve" for the exact solution, resulting in a more accurate method of computing approximations.

Example 7.6.1 *Consider the function*

$$f(x) = \frac{\sin^2\left(\frac{\sqrt{x^2+x}}{\cos x - x}\right)}{\sin\left(\frac{\sqrt{x}-1}{\sqrt{x^2+1}}\right)}.$$

Our goal is to compute $f'(0.25)$ *as accurately as possible. Using a centered difference approximation,*

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2),$$

with $x = 0.25$ *and* $h = 0.01$*, we obtain the approximation*

$$f'(0.25) \approx \frac{f(0.26) - f(0.24)}{0.02} = -9.06975297890147,$$

which has absolute error 3.0×10^{-3}*, and if we use* $h = 0.005$*, we obtain the approximation*

$$f'(0.25) \approx \frac{f(0.255) - f(0.245)}{0.01} = -9.06746429492149,$$

which has absolute error 7.7×10^{-4}*. As expected, the error decreases by a factor of approximately 4 when we halve the step size* h*, because the error in the centered difference formula is of* $O(h^2)$*.*

We can obtain a more accurate approximation by applying Richardson Extrapolation *to these approximations. We define the function* $N_1(h)$ *to be the centered difference approximation to* $f'(0.25)$ *obtained using the step size* h*. Then, with* $h = 0.01$*, we have*

$$N_1(h) = -9.06975297890147, \quad N_1(h/2) = -9.06746429492149,$$

and the exact value is given by $N_1(0) = -9.06669877124279$*. Because the error in the centered difference approximation satisfies*

$$N_1(h) = N_1(0) + K_1h^2 + K_2h^4 + K_3h^6 + O(h^8), \tag{7.19}$$

where the constants K_1*,* K_2 *and* K_3 *depend on the derivatives of* $f(x)$ *at* $x = 0.25$*, it follows that the new approximation*

$$N_2(h) = N_1(h/2) + \frac{N_1(h/2) - N_1(h)}{2^2 - 1} = -9.06670140026149,$$

has fourth-order accuracy. Specifically, if we denote the exact value by $N_2(0)$*, we have*

$$N_2(h) = N_2(0) + \tilde{K}_2h^4 + \tilde{K}_3h^6 + O(h^8),$$

where the constants $\tilde{K}_2$ *and* $\tilde{K}_3$ *are independent of* h.

Now, suppose that we compute

$$N_1(h/4) = \frac{f(x+h/4) - f(x-h/4)}{2(h/4)} = \frac{f(0.2525) - f(0.2475)}{0.005} = -9.06689027527046,$$

which has an absolute error of 1.9×10^{-4}, *we can use extrapolation again to obtain a second fourth-order accurate approximation,*

$$N_2(h/2) = N_1(h/4) + \frac{N_1(h/4) - N_1(h/2)}{3} = -9.06669893538678,$$

which has absolute error of 1.7×10^{-7}. *It follows from the form of the error in* $N_2(h)$ *that we can use extrapolation on* $N_2(h)$ *and* $N_2(h/2)$ *to obtain a* sixth-*order accurate approximation,*

$$N_3(h) = N_2(h/2) + \frac{N_2(h/2) - N_2(h)}{2^4 - 1} = -9.06669877106180,$$

which has an absolute error of 1.8×10^{-10}. □

Exercise 7.6.1 *Use Taylor series expansion to prove (7.19); that is, the error in the centered difference approximation can be expressed as a sum of terms involving only even powers of* h.

Exercise 7.6.2 *Based on the preceding example, give a general formula for Richardson extrapolation, applied to the approximation* $F(h)$ *from (7.18), that uses* $F(h)$ *and* $F(h/d)$, *for some integer* $d > 1$, *to obtain an approximation of* $F(0)$ *that is of order* r.

7.6.2 The Euler-Maclaurin Expansion

In the previous example, it was stated, without proof, that the error in the centered difference approximation could be expressed as a sum of terms involving even powers of the spacing h. We would like to use Richardson Extrapolation to enhance the accuracy of approximate integrals computed using the Composite Trapezoidal Rule, but first we must determine the form of the error in these approximations. We have established that the Composite Trapezoidal Rule is second-order accurate, but if Richardson Extrapolation is used once to eliminate the $O(h^2)$ portion of the error, we do not know the order of what remains.

Suppose that $g(t)$ is differentiable on $(-1,1)$. From integration by parts, we have

$$\int_{-1}^{1} g(t)\,dt = tg(t)|_{-1}^{1} - \int_{-1}^{1} tg'(t)\,dt = [g(-1)+g(1)] - \int_{-1}^{1} tg'(t)\,dt.$$

The first term on the right side of the equals sign is the basic Trapezoidal Rule approximation of the integral on the left side of the equals sign. The second term on the right side is the error in this approximation. If g is $2k$-times differentiabe on $(-1,1)$, and we repeatedly apply integration by parts, $2k-1$ times, we obtain

$$\begin{aligned}\int_{-1}^{1} g(t)\,dt - [g(-1)+g(1)] &= \left[q_2(t)g'(t) - q_3(t)g''(t) + \cdots + q_{2k}(t)g^{(2k-1)}(t)\right]\Big|_{-1}^{1} - \\ &\quad \int_{-1}^{1} q_{2k}(t)g^{(2k)}(t)\,dt,\end{aligned}$$

where the sequence of polynomials $q_1(t), \dots, q_{2k}(t)$ satisfy

$$q_1(t) = -t, \quad q'_{r+1}(t) = q_r(t), \quad r = 1, 2, \dots, 2k-1.$$

If we choose the constants of integration correctly, then, because $q_1(t)$ is an odd function, we can ensure that $q_r(t)$ is an odd function if r is odd, and an even function if r is even. This ensures that $q_{2r}(-1) = q_{2r}(1)$. Furthermore, we can ensure that $q_r(-1) = q_r(1) = 0$ if r is odd. This causes the boundary terms involving $q_r(t)$ to vanish when r is odd, which yields

$$\int_{-1}^{1} g(t)\,dt - [g(-1) + g(1)] = \sum_{r=1}^{k} q_{2r}(1)[g^{(2r-1)}(1) - g^{(2r-1)}(-1)] - \int_{-1}^{1} q_{2k}(t) g^{(2k)}(t)\,dt.$$

Using this expression for the error in the context of the Composite Trapezoidal Rule, applied to the integral of a $2k$-times differentiable function $f(x)$ on a general interval $[a, b]$, yields the **Euler-Maclaurin Expansion**

$$\begin{aligned}\int_a^b f(x)\,dx &= \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right] + \\ &\quad \sum_{r=1}^{k} c_r h^{2r}[f^{(2r-1)}(b) - f^{(2r-1)}(a)] - \left(\frac{h}{2}\right)^{2k} \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} q_{2k}(t) f^{(2k)}(x)\,dx,\end{aligned}$$

where, for each $i = 1, 2, \dots, n$, $t = -1 + \frac{2}{h}(x - x_{i-1})$, and the constants

$$c_r = \frac{q_{2r}(1)}{2^{2r}} = -\frac{B_{2r}}{(2r)!}, \quad r = 1, 2, \dots, k$$

are closely related to the *Bernoulli numbers* B_r.

It can be seen from this expansion that the error $E_{trap}(h)$ in the Composite Trapezoidal Rule, like the error in the centered difference approximation of the derivative, has the form

$$E_{trap}(h) = K_1 h^2 + K_2 h^4 + K_3 h^6 + \cdots + O(h^{2k}),$$

where the constants K_i are independent of h, provided that the integrand is at least $2k$ times continuously differentiable. This knowledge of the error provides guidance on how Richardson Extrapolation can be repeatedly applied to approximations obtained using the Composite Trapezoidal Rule at different spacings in order to obtain higher-order accurate approximations.

It can also be seen from the Euler-Maclaurin Expansion that the Composite Trapezoidal Rule is particularly accurate when the integrand is a periodic function, of period $b - a$, as this causes the terms involving the derivatives of the integrand at a and b to vanish. Specifically, if $f(x)$ is periodic with period $b - a$, and is at least $2k$ times continuously differentiable, then the error in the Composite Trapezoidal Rule approximation to $\int_a^b f(x)\,dx$, with spacing h, is $O(h^{2k})$, rather than $O(h^2)$. It follows that if $f(x)$ is infinitely differentiable, such as a finite linear combination of sines or cosines, then the Composite Trapezoidal Rule has an *exponential* order of accuracy, meaning that as $h \to 0$, the error converges to zero more rapidly than *any* power of h.

Exercise 7.6.3 *Use integration by parts to obtain explicit formulas for the polynomials* $q_r(t)$ *for* $r = 2, 3, 4$.

Exercise 7.6.4 *Use the Composite Trapezoidal Rule to integrate* $f(x) = \sin k\pi x$, *where* k *is an integer, over* $[0, 1]$. *How does the error behave?*

7.6.3 Romberg Integration

Richardson extrapolation is not only used to compute more accurate approximations of derivatives, but is also used as the foundation of a numerical integration scheme called **Romberg integration** [32]. In this scheme, the integral

$$I[f] = \int_a^b f(x)\,dx$$

is approximated using the Composite Trapezoidal Rule with step sizes $h_k = (b-a)2^{-k}$, where k is a nonnegative integer. Then, for each k, Richardson extrapolation is used $k-1$ times to previously computed approximations in order to improve the order of accuracy as much as possible.

More precisely, suppose that we compute approximations $T_{1,1}$ and $T_{2,1}$ to the integral, using the Composite Trapezoidal Rule with one and two subintervals, respectively. That is,

$$\begin{aligned} T_{1,1} &= \frac{b-a}{2}[f(a)+f(b)] \\ T_{2,1} &= \frac{b-a}{4}\left[f(a)+2f\left(\frac{a+b}{2}\right)+f(b)\right]. \end{aligned}$$

Suppose that f has continuous derivatives of all orders on $[a,b]$. Then, the Composite Trapezoidal Rule, for a general number of subintervals n, satisfies

$$\int_a^b f(x)\,dx = \frac{h}{2}\left[f(a)+2\sum_{j=1}^{n-1} f(x_j)+f(b)\right]+\sum_{i=1}^{\infty} K_i h^{2i},$$

where $h=(b-a)/n$, $x_j = a+jh$, and the constants $\{K_i\}_{i=1}^{\infty}$ depend only on the derivatives of f. It follows that we can use Richardson Extrapolation to compute an approximation with a higher order of accuracy. If we denote the exact value of the integral by $I[f]$ then we have

$$\begin{aligned} T_{1,1} &= I[f]+K_1h^2+O(h^4) \\ T_{2,1} &= I[f]+K_1(h/2)^2+O(h^4) \end{aligned}$$

Neglecting the $O(h^4)$ terms, we have a system of equations that we can solve for K_1 and $I[f]$. The value of $I[f]$, which we denote by $T_{2,2}$, is an improved approximation given by

$$T_{2,2} = T_{2,1} + \frac{T_{2,1}-T_{1,1}}{3}.$$

It follows from the representation of the error in the Composite Trapezoidal Rule that $I[f] = T_{2,2}+O(h^4)$.

Suppose that we compute another approximation $T_{3,1}$ using the Composite Trapezoidal Rule with 4 subintervals. Then, as before, we can use Richardson Extrapolation with $T_{2,1}$ and $T_{3,1}$ to obtain a new approximation $T_{3,2}$ that is fourth-order accurate. Now, however, we have two approximations, $T_{2,2}$ and $T_{3,2}$, that satisfy

$$\begin{aligned} T_{2,2} &= I[f]+\tilde{K}_2h^4+O(h^6) \\ T_{3,2} &= I[f]+\tilde{K}_2(h/2)^4+O(h^6) \end{aligned}$$

for some constant $\tilde{K}_2$. It follows that we can apply Richardson Extrapolation to these approximations to obtain a new approximation $T_{3,3}$ that is *sixth-order* accurate. We can continue this process to obtain as high an order of accuracy as we wish. We now describe the entire algorithm.

Algorithm 7.6.2 *(Romberg Integration) Given a positive integer* J*, an interval* $[a,b]$ *and a function* $f(x)$*, the following algorithm computes an approximation to* $I[f] = \int_a^b f(x)\,dx$ *that is accurate to order* $2J$*.*

```
$h = b - a$
for $j = 1, 2, \ldots, J$ do
    $T_{j,1} = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{2^{j-1}-1} f(a+ih) + f(b)\right]$ (Composite Trapezoidal Rule)
    for $k = 2, 3, \ldots, j$ do
        $T_{j,k} = T_{j,k-1} + \frac{T_{j,k-1} - T_{j-1,k-1}}{4^{k-1}-1}$ (Richardson Extrapolation)
    end
    $h = h/2$
end
```

It should be noted that in a practical implementation, $T_{j,1}$ can be computed more efficiently by using $T_{j-1,1}$, because $T_{j-1,1}$ already includes more than half of the function values used to compute $T_{j,1}$, and they are weighted correctly relative to one another. It follows that for $j > 1$, if we split the summation in the algorithm into two summations containing odd- and even-numbered terms, respectively, we obtain

$$\begin{aligned} T_{j,1} &= \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{2^{j-2}} f(a+(2i-1)h) + 2\sum_{i=1}^{2^{j-2}-1} f(a+2ih) + f(b)\right] \\ &= \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{2^{j-2}-1} f(a+2ih) + f(b)\right] + \frac{h}{2}\left[2\sum_{i=1}^{2^{j-2}} f(a+(2i-1)h)\right] \\ &= \frac{1}{2}T_{j-1,1} + h\sum_{i=1}^{2^{j-2}} f(a+(2i-1)h). \end{aligned}$$

Example 7.6.3 *We will use* Romberg integration *to obtain a sixth-order accurate approximation to*

$$\int_0^1 e^{-x^2}\,dx,$$

an integral that cannot *be computed using the Fundamental Theorem of Calculus. We begin by using the Trapezoidal Rule, or, equivalently, the Composite Trapezoidal Rule*

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(a) + \sum_{j=1}^{n-1} f(x_j) + f(b)\right], \quad h = \frac{b-a}{n}, \quad x_j = a + jh,$$

with $n = 1$ *subintervals. Since* $h = (b-a)/n = (1-0)/1 = 1$*, we have*

$$T_{1,1} = \frac{1}{2}[f(0) + f(1)] = 0.68393972058572,$$

which has an absolute error of 6.3×10^{-2}.

If we bisect the interval $[0, 1]$ *into two subintervals of equal width, and approximate the area under* e^{-x^2} *using two trapezoids, then we are applying the Composite Trapezoidal Rule with* $n = 2$ *and* $h = (1 - 0)/2 = 1/2$*, which yields*

$$T_{2,1} = \frac{0.5}{2}[f(0) + 2f(0.5) + f(1)] = 0.73137025182856,$$

which has an absolute error of 1.5×10^{-2}. *As expected, the error is reduced by a factor of 4 when the step size is halved, since the error in the Composite Trapezoidal Rule is of* $O(h^2)$.

Now, we can use Richardson Extrapolation to obtain a more accurate approximation,

$$T_{2,2} = T_{2,1} + \frac{T_{2,1} - T_{1,1}}{3} = 0.74718042890951,$$

which has an absolute error of 3.6×10^{-4}. *Because the error in the Composite Trapezoidal Rule satisfies*

$$\int_a^b f(x)\,dx = \frac{h}{2}\left[f(a) + \sum_{j=1}^{n-1} f(x_j) + f(b)\right] + K_1h^2 + K_2h^4 + K_3h^6 + O(h^8),$$

where the constants K_1, K_2 *and* K_3 *depend on the derivatives of* $f(x)$ *on* $[a, b]$ *and are independent of* h*, we can conclude that* $T_{2,1}$ *has fourth-order accuracy.*

We can obtain a second approximation of fourth-order accuracy by using the Composite Trapezoidal Rule with $n = 4$ *to obtain a third approximation of second-order accuracy. We set* $h = (1 - 0)/4 = 1/4$*, and then compute*

$$T_{3,1} = \frac{0.25}{2}\left[f(0) + 2[f(0.25) + f(0.5) + f(0.75)] + f(1)\right] = 0.74298409780038,$$

which has an absolute error of 3.8×10^{-3}. *Now, we can apply Richardson Extrapolation to* $T_{2,1}$ *and* $T_{3,1}$ *to obtain*

$$T_{3,2} = T_{3,1} + \frac{T_{3,1} - T_{2,1}}{3} = 0.74685537979099,$$

which has an absolute error of 3.1×10^{-5}. *This significant decrease in error from* $T_{2,2}$ *is to be expected, since both* $T_{2,2}$ *and* $T_{3,2}$ *have fourth-order accuracy, and* $T_{3,2}$ *is computed using half the step size of* $T_{2,2}$.

It follows from the error term in the Composite Trapezoidal Rule, and the formula for Richardson Extrapolation, that

$$T_{2,2} = \int_0^1 e^{-x^2}\,dx + \tilde{K}_2h^4 + O(h^6), \quad T_{2,2} = \int_0^1 e^{-x^2}\,dx + \tilde{K}_2\left(\frac{h}{2}\right)^4 + O(h^6).$$

Therefore, we can use Richardson Extrapolation with these two approximations to obtain a new approximation

$$T_{3,3} = T_{3,2} + \frac{T_{3,2} - T_{2,2}}{2^4 - 1} = 0.74683370984975,$$

which has an absolute error of 9.6×10^{-6}. *Because* $T_{3,3}$ *is a linear combination of* $T_{3,2}$ *and* $T_{2,2}$ *in which the terms of order* h^4 *cancel, we can conclude that* $T_{3,3}$ *is of sixth-order accuracy.* □

Exercise 7.6.5 *Write a* MATLAB *function* `I=quadromberg(f,a,b,J)` *that implements Algorithm 7.6.2 for Romberg integration described in this section. Apply it to approximate the integrals*

$$\int_0^1 e^x\,dx, \quad \int_0^1 \frac{1}{1+x^2}\,dx, \quad \int_0^1 \sqrt{x}\,dx.$$

How does the accuracy of the approximations improve as the number of extrapolations, J, increases? Explain the difference in the observed behavior.

7.7 Adaptive Quadrature

Composite rules can be used to implement an **automatic** quadrature procedure, in which the all of the subintervals of $[a,b]$ are continually subdivided until sufficient accuracy is achieved. However, this approach is impractical since small subintervals are not necessary in regions where the integrand is smooth.

An alternative is **adaptive quadrature** [30]. Adaptive quadrature is a technique in which the interval $[a,b]$ is divided into n subintervals $[a_j,b_j]$, for $j=0,1,\ldots,n-1$, and a quadrature rule, such as the Trapezoidal Rule, is used on each subinterval to compute

$$I_j[f] = \int_{a_j}^{b_j} f(x)\,dx,$$

as in any composite quadrature rule. However, in adaptive quadrature, a subinterval $[a_j,b_j]$ is subdivided if it is determined that the quadrature rule has not computed $I_j[f]$ with sufficient accuracy.

To make this determination, we use the quadrature rule on $[a_j,b_j]$ to obtain an approximation I_1, and then use the corresponding composite rule on $[a_j,b_j]$, with two subintervals, to compute a second approximation I_2. If I_1 and I_2 are sufficiently close, then it is reasonable to conclude that these two approximations are accurate, so there is no need to subdivide $[a_j,b_j]$. Otherwise, we divide $[a_j,b_j]$ into two subintervals, and repeat this process on these subintervals. We apply this technique to all subintervals, until we can determine that the integral of f over each one has been computed with sufficient accuracy. By subdividing only when it is necessary, we avoid unnecessary computation and obtain the same accuracy as with composite rules or automatic quadrature procedures, but with much less computational effort.

How do we determine whether I_1 and I_2 are sufficiently close? Suppose that the composite rule has order of accuracy p. Then, I_1 and I_2 should satisfy

$$I - I_2 \approx \frac{1}{2^p}(I - I_1),$$

where I is the exact value of the integral over $[a_j,b_j]$. We then have

$$I - I_2 \approx \frac{1}{2^p}(I - I_2 + I_2 - I_1)$$

which can be rearranged to obtain

$$I - I_2 \approx \frac{1}{2^p - 1}(I_2 - I_1).$$

Thus we have obtained an *error estimate* in terms of our two approximations.

We now describe an algorithm for adaptive quadrature. This algorithm uses the Trapezoidal Rule to integrate over intervals, and intervals are subdivided as necessary into two subintervals of equal width. The algorithm uses a data structure called a *stack* in order to keep track of the subintervals over which f still needs to be integrated. A stack is essentially a list of elements, where the elements, in this case, are subintervals. An element is added to the stack using a *push* operation, and is removed using a *pop* operation. A stack is often described using the phrase "last-in-first-out," because the most recent element to be pushed onto the stack is the first element to be popped. This corresponds to our intuitive notion of a stack of objects, in which objects are placed on top of the stack and are removed from the top as well.

Algorithm 7.7.1 *(Adaptive Quadrature) Given a function $f(x)$ that is integrable on an interval $[a, b]$, the following algorithm computes an approximation I to $I[f] = \int_a^b f(x)\,dx$ that is accurate to within $(b-a)TOL$, where TOL is a given error tolerance.*

```
S is an empty stack
push(S, [a, b])
I = 0
while S is not empty do
    [a, b] = pop(S) (the interval [a, b] on top of S is removed from S)
    $I_1 = [(b-a)/2][f(a) + f(b)]$ (Trapezoidal Rule)
    $m = (a+b)/2$
    $I_2 = [(b-a)/4][f(a) + 2f(m) + f(b)]$ (Composite Trapezoidal Rule with 2 subintervals)
    if $|I_1 - I_2| < 3(b-a)TOL$ then
        $I = I_1 + I_2$ (from error term in Trapezoidal Rule, $|I[f] - I_2| \approx \frac{1}{3}|I_1 - I_2|$)
    else
        push(S, [a, m])
        push(S, [m, b])
    end
end
```

Throughout the execution of the loop in the above algorithm, the stack S contains all intervals over which f still needs to be integrated to within the desired accuracy. Initially, the only such interval is the original interval $[a, b]$. As long as intervals remain in the stack, the interval on top of the stack is removed, and we attempt to integrate over it. If we obtain a sufficiently accurate result, then we are finished with the interval. Otherwise, the interval is bisected into two subintervals, both of which are pushed on the stack so that they can be processed later. Once the stack is empty, we know that we have accurately integrated f over a collection of intervals whose union is the original interval $[a, b]$, so the algorithm can terminate.

Example 7.7.2 *We will use* adaptive quadrature *to compute the integral*

$$\int_0^{\pi/4} e^{3x} \sin 2x\,dx$$

to within $(\pi/4)10^{-4}$.

Let $f(x) = e^{3x}\sin 2x$ denote the integrand. First, we use Simpson's Rule, or, equivalently, the Composite Simpson's Rule with $n = 2$ subintervals, to obtain an approximation I_1 to this integral.

We have

$$I_1 = \frac{\pi/4}{6}[f(0) + 4f(\pi/8) + f(\pi/4)] = 2.58369640324748.$$

Then, we divide the interval $[0, \pi/4]$ *into two subintervals of equal width,* $[0, \pi/8]$ *and* $[\pi/8, \pi/4]$, *and integrate over each one using Simpson's Rule to obtain a second approximation* I_2. *This is equivalent to using the Composite Simpson's Rule on* $[0, \pi/4]$ *with* $n = 4$ *subintervals. We obtain*

$$\begin{aligned} I_2 &= \frac{\pi/8}{6}[f(0) + 4f(\pi/16) + f(\pi/8)] + \frac{\pi/8}{6}[f(\pi/8) + 4f(3\pi/16) + f(\pi/4)] \\ &= \frac{\pi/16}{3}[f(0) + 4f(\pi/16) + 2f(\pi/8) + 4f(3\pi/16) + f(\pi/4)] \\ &= 2.58770145345862. \end{aligned}$$

Now, we need to determine whether the approximation I_2 *is sufficiently accurate. Because the error in the Composite Simpson's Rule is* $O(h^4)$, *where* h *is the width of each subinterval used in the rule, it follows that the actual error in* I_2 *satisfies*

$$|I_2 - I[f]| \approx \frac{1}{15}|I_2 - I_1|,$$

where $I[f]$ *is the exact value of the integral of* f.

We find that the relation

$$|I_2 - I[f]| \approx \frac{1}{15}|I_2 - I_1| < \frac{\pi}{4}10^{-4}$$

is not *satisfied, so we must divide the interval* $[0, \pi/4]$ *into two subintervals of equal width,* $[0, \pi/8]$ *and* $[\pi/8, \pi/4]$, *and use the Composite Simpson's Rule with these smaller intervals in order to achieve the desired accuracy.*

First, we work with the interval $[0, \pi/8]$. *Proceeding as before, we use the Composite Simpson's Rule with* $n = 2$ *and* $n = 4$ *subintervals to obtain approximations* I_1 *and* I_2 *to the integral of* $f(x)$ *over this interval. We have*

$$I_1 = \frac{\pi/8}{6}[f(0) + 4f(\pi/16) + f(\pi/8)] = 0.33088926959519.$$

and

$$\begin{aligned} I_2 &= \frac{\pi/16}{6}[f(0) + 4f(\pi/32) + f(\pi/16)] + \frac{\pi/16}{6}[f(\pi/16) + 4f(3\pi/32) + f(\pi/8)] \\ &= \frac{\pi/32}{3}[f(0) + 4f(\pi/32) + 2f(\pi/16) + 4f(3\pi/32) + f(\pi/8)] \\ &= 0.33054510467064. \end{aligned}$$

Since these approximations satisfy the relation

$$|I_2 - I[f]| \approx \frac{1}{15}|I_2 - I_1| < \frac{\pi}{8}10^{-4},$$

where $I[f]$ *denotes the exact value of the integral of* f *over* $[0, \pi/8]$, *we have achieved sufficient accuracy on this interval and we do not need to subdivide it further. The more accurate approximation* I_2 *can be included in our approximation to the integral over the original interval* $[0, \pi/4]$.

Now, we need to achieve sufficient accuracy on the remaining subinterval, $[\pi/4, \pi/8]$*. As before, we compute the approximations* I_1 *and* I_2 *of the integral of* f *over this interval and obtain*

$$I_1 = \frac{\pi/8}{6}[f(\pi/8) + 4f(3\pi/16) + f(\pi/4)] = 2.25681218386343.$$

and

$$\begin{aligned} I_2 &= \frac{\pi/16}{6}[f(\pi/8) + 4f(5\pi/32) + f(3\pi/16)] + \frac{\pi/16}{6}[f(3\pi/16) + 4f(7\pi/32) + f(\pi/4)] \\ &= \frac{\pi/32}{3}[f(\pi/8) + 4f(5\pi/32) + 2f(3\pi/16) + 4f(7\pi/32) + f(\pi/4)] \\ &= 2.25801455892266. \end{aligned}$$

Since these approximations do not satisfy the relation

$$|I_2 - I[f]| \approx \frac{1}{15}|I_2 - I_1| < \frac{\pi}{8}10^{-4},$$

where $I[f]$ *denotes the exact value of the integral of* f *over* $[\pi/8, \pi/4]$*, we have not achieved sufficient accuracy on this interval and we need to subdivide it into two subintervals of equal width,* $[\pi/8, 3\pi/16]$ *and* $[3\pi/16, \pi/4]$*, and use the Composite Simpson's Rule with these smaller intervals in order to achieve the desired accuracy.*

The discrepancy in these two approximations to the integral of f *over* $[\pi/4, \pi/8]$ *is larger than the discrepancy in the two approximations of the integral over* $[0, \pi/8]$ *because even though these intervals have the same width, the derivatives of* f *are larger on* $[\pi/8, \pi/4]$*, and therefore the error in the Composite Simpson's Rule is larger.*

We continue the process of adaptive quadrature on the interval $[\pi/8, 3\pi/16]$*. As before, we compute the approximations* I_1 *and* I_2 *of the integral of* f *over this interval and obtain*

$$I_1 = \frac{\pi/16}{6}[f(\pi/8) + 4f(5\pi/32) + f(3\pi/16)] = 0.72676545197054.$$

and

$$\begin{aligned} I_2 &= \frac{\pi/32}{6}[f(\pi/8) + 4f(9\pi/64) + f(5\pi/32)] + \frac{\pi/32}{6}[f(5\pi/32) + 4f(11\pi/64) + f(3\pi/16)] \\ &= \frac{\pi/64}{3}[f(\pi/8) + 4f(9\pi/64) + 2f(5\pi/32) + 4f(11\pi/64) + f(3\pi/16)] \\ &= 0.72677918153379. \end{aligned}$$

Since these approximations satisfy the relation

$$|I_2 - I[f]| \approx \frac{1}{15}|I_2 - I_1| < \frac{\pi}{16}10^{-4},$$

where $I[f]$ *denotes the exact value of the integral of* f *over* $[\pi/8, 3\pi/16]$*, we have achieved sufficient accuracy on this interval and we do not need to subdivide it further. The more accurate approximation* I_2 *can be included in our approximation to the integral over the original interval* $[0, \pi/4]$*.*

Now, we work with the interval $[3\pi/16, \pi/4]$*. Proceeding as before, we use the Composite Simpson's Rule with* $n = 2$ *and* $n = 4$ *subintervals to obtain approximations* I_1 *and* I_2 *to the integral of* $f(x)$ *over this interval. We have*

$$I_1 = \frac{\pi/16}{6}[f(3\pi/16) + 4f(7\pi/32) + f(\pi/4)] = 1.53124910695212.$$

and

$$\begin{aligned} I_2 &= \frac{\pi/32}{6}[f(3\pi/16) + 4f(13\pi/64) + f(7\pi/32)] + \frac{\pi/32}{6}[f(7\pi/32) + 4f(15\pi/64) + f(\pi/4)] \\ &= \frac{\pi/64}{3}[f(3\pi/16) + 4f(13\pi/64) + 2f(7\pi/32) + 4f(15\pi/64) + f(\pi/4)] \\ &= 1.53131941583939. \end{aligned}$$

Since these approximations satisfy the relation

$$|I_2 - I[f]| \approx \frac{1}{15}|I_2 - I_1| < \frac{\pi}{16}10^{-4},$$

where $I[f]$ *denotes the exact value of the integral of* f *over* $[3\pi/16, \pi/4]$*, we have achieved sufficient accuracy on this interval and we do not need to subdivide it further. The more accurate approximation* I_2 *can be included in our approximation to the integral over the original interval* $[0, \pi/4]$*.*

We conclude that the integral of $f(x)$ *over* $[0, \pi/4]$ *can be approximated by the sum of our approximate integrals over* $[0, \pi/8]$*,* $[\pi/8, 3\pi/16]$*, and* $[3\pi/16, \pi/4]$*, which yields*

$$\begin{aligned} \int_0^{\pi/4} e^{3x}\sin 2x\,dx &\approx 0.33054510467064 + 0.72677918153379 + 1.53131941583939 \\ &\approx 2.58864370204382. \end{aligned}$$

Since the exact value is 2.58862863250716, the absolute error is -1.507×10^{-5}*, which is less in magnitude than our desired error bound of* $(\pi/4)10^{-4} \approx 7.854 \times 10^{-5}$*. This is because on each subinterval, we ensured that our approximation was accurate to within* 10^{-4} *times the width of the subinterval, so that when we added these approximations, the total error in the sum would be bounded by* 10^{-4} *times the width of the union of these subintervals, which is the original interval* $[0, \pi/4]$*.*

The graph of the integrand over the interval of integration is shown in Figure 7.1. □

Adaptive quadrature can be very effective, but it should be used cautiously, for the following reasons:

- The integrand is only evaluated at a few points within each subinterval. Such sampling can miss a portion of the integrand whose contribution to the integral can be misjudged.
- Regions in which the function is not smooth will still only make a small contribution to the integral if the region itself is small, so this should be taken into account to avoid unnecessary function evaluations.

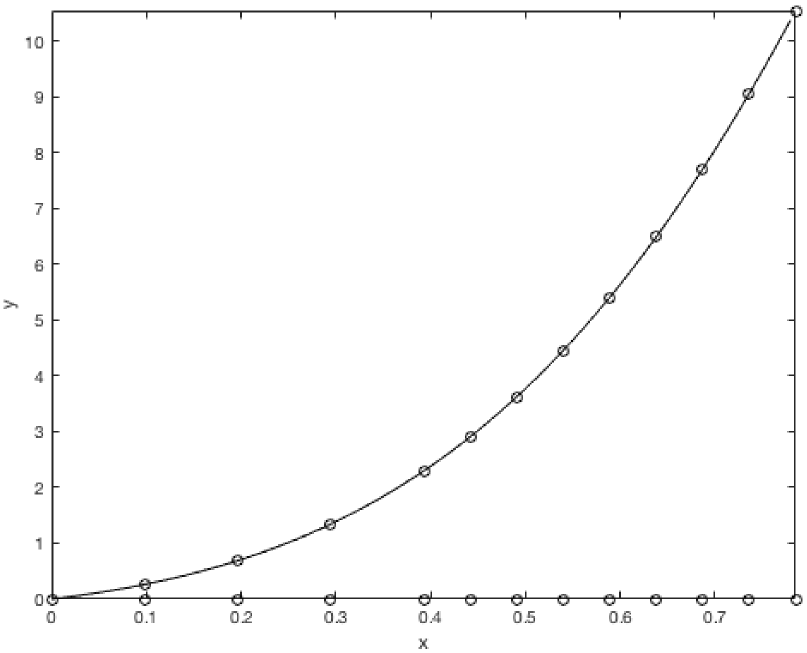


Figure 7.1: Graph of $f(x) = e^{3x}\sin 2x$ on $[0,\pi/4]$, with quadrature nodes from Example 7.7.2 shown on the graph and on the x-axis.

- Adaptive quadrature can be very inefficient if the integrand has a discontinuity within a subinterval, since repeated subdivision will occur. This is unnecessary if the integrand is smooth on either side of the discontinuity, so subintervals should be chosen so that discontinuities occur between subintervals whenever possible.

Exercise 7.7.1 *Write* MATLAB *functions* `S=stackpush(S,v)` *and* `[S,v]=stackpop(S)` *that implement the* push *and* pop *operations described in this section, in a manner that is useful for adaptive quadrature. Assume that* `v` *is a* row *vector of a fixed length, and* `S` *is a matrix in which each row represents an element of the stack.*

Exercise 7.7.2 *Write a* MATLAB *function* `I=adapquad(f,a,b,tol)` *that implements Algorithm 7.7.1 and uses the functions* `stackpush` *and* `stackpop` *from Exercise 7.7.1. Then change your function so that Simpson's Rule is used in place of the Trapezoidal Rule.*

Exercise 7.7.3 *Write a* MATLAB *function* `I=adapquadrecur(f,a,b,tol)` *that implements adaptive quadrature as described in this section, but uses recursion instead of a stack to keep track of subintervals.*

Exercise 7.7.4 *Let* $f(x) = e^{-1000(x-c)^2}$*, where* c *is a parameter. Use your function* `adapquadrecur` *from Exercise 7.7.3 to approximate* $\int_0^1 f(x)\,dx$ *for the cases* $c = 1/8$ *and* $c = 1/4$*. Explain the difference in performance between these two cases.*

Exercise 7.7.5 *Explain why a straightforward implementation of adaptive quadrature using recursion, as in your function* `adapquadrecur` *from Exercise 7.7.3, is not as efficient as it could be. What can be done to make it more efficient? Modify your implementation accordingly.*

7.8 Multiple Integrals

As many problems in scientific computing involve two- or three-dimensional domains, it is essential to be able to compute integrals over such domains. In this section, we explore the generalization of techniques for integrals of functions of one variable to such multivariable cases.

7.8.1 Double Integrals

Double integrals can be evaluated using the following strategies:

- If a two-dimensional domain Ω can be decomposed into rectangles, then the integral of a function $f(x, y)$ over Ω can be computed by evaluating integrals of the form

$$I[f] = \int_a^b \int_c^d f(x, y)\,dy\,dx. \tag{7.20}$$

  Then, to evaluate $I[f]$, one can use a **Cartesian product rule**, whose nodes and weights are obtained by combining one-dimensional quadrature rules that are applied to each dimension. For example, if functions of x are integrated along the line between $x = a$ and $x = b$ using nodes x_i and weights w_i, for $i = 1, \ldots, n$, and if functions of y are integrated along the line between $y = c$ and $y = d$ using nodes y_i and weights z_i, for $i = 1, \ldots, m$, then the resulting Cartesian product rule

$$Q_{n,m}[f] = \sum_{i=1}^{n} \sum_{j=1}^{m} f(x_i, y_j) w_i z_j$$

  has nodes (x_i, y_j) and corresponding weights $w_i z_j$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

- If the domain Ω can be described as the region between two curves $y_1(x)$ and $y_2(x)$ for $x \in [a, b]$, then we can write

$$I[f] = \int\int_{\Omega} f(x, y)\,dA$$

  as an *iterated integral*

$$I[f] = \int_a^b \int_{y_1(x)}^{y_2(x)} f(x, y)\,dy\,dx$$

  which can be evaluated by applying a one-dimensional quadrature rule to compute the *outer integral*

$$I[f] = \int_a^b g(x)\,dx$$

where $g(x)$ is evaluated by using a one-dimensional quadrature rule to compute the *inner integral*

$$g(x) = \int_{y_1(x)}^{y_2(x)} f(x,y)\,dy.$$

- For various simple regions such as triangles, there exist **cubature rules** that are not combinations of one-dimensional quadrature rules. Cubature rules are more direct generalizations of quadrature rules, in that they evaluate the integrand at selected nodes and use weights determined by the geometry of the domain and the placement of the nodes.

It should be noted that all of these strategies apply to certain special cases. The first algorithm capable of integrating over a general two-dimensional domain was developed by Lambers and Rice [21]. This algorithm combines the second and third strategies described above, decomposing the domain into subdomains that are either triangles or regions between two curves.

Example 7.8.1 *We will use the Composite Trapezoidal Rule with* $m = n = 2$ *to evaluate the double integral*

$$\int_0^{1/2}\int_0^{1/2} e^{y-x}\,dy\,dx.$$

The Composite Trapezoidal Rule with $n = 2$ *subintervals is*

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(a) + 2f\left(\frac{a+b}{2}\right) + f(b)\right], \quad h = \frac{b-a}{n}.$$

If $a = 0$ *and* $b = 1/2$*, then* $h = (1/2-0)/2 = 1/4$ *and this simplifies to*

$$\int_0^{1/2} f(x)\,dx \approx \frac{1}{8}[f(0) + 2f(1/4) + f(1/2)].$$

We first use this rule to evaluate the "single" integral

$$\int_0^{1/2} g(x)\,dx$$

where

$$g(x) = \int_0^1 e^{y-x}\,dy.$$

This yields

$$\begin{aligned}\int_0^{1/2}\int_0^{1/2} e^{y-x}\,dy\,dx &= \int_0^{1/2} g(x)\,dx \\ &\approx \frac{1}{8}[g(0) + 2g(1/4) + g(1/2)] \\ &\approx \frac{1}{8}\left[\int_0^{1/2} e^{y-0}\,dy + 2\int_0^{1/2} e^{y-1/4}\,dy + \int_0^{1/2} e^{y-1/2}\,dy\right].\end{aligned}$$

Now, to evaluate each of these integrals, we use the Composite Trapezoidal Rule in the y-direction with $m = 2$*. If we let* k *denote the step size in the y-direction, we have* $k = (1/2 - 0)/2 = 1/4$*, and therefore we have*

$$\begin{aligned}
\int_0^{1/2}\int_0^{1/2} e^{y-x}\,dy\,dx &\approx \frac{1}{8}\left[\int_0^{1/2} e^{y-0}\,dy + 2\int_0^{1/2} e^{y-1/4}\,dy + \int_0^{1/2} e^{y-1/2}\,dy\right] \\
&\approx \frac{1}{8}\left[\frac{1}{8}\left[e^{0-0} + 2e^{1/4-0} + e^{1/2-0}\right] + \right. \\
&\qquad 2\frac{1}{8}\left[e^{0-1/4} + 2e^{1/4-1/4} + e^{1/2-1/4}\right] + \\
&\qquad \left.\frac{1}{8}\left[e^{0-1/2} + 2e^{1/4-1/2} + e^{1/2-1/2}\right]\right] \\
&\approx \frac{1}{64}\left[e^{0} + 2e^{1/4} + e^{1/2}\right] + \\
&\qquad \frac{1}{32}\left[e^{-1/4} + 2e^{0} + e^{1/4}\right] + \\
&\qquad \frac{1}{64}\left[e^{-1/2} + 2e^{-1/4} + e^{0}\right] \\
&\approx \frac{3}{32}e^{0} + \frac{1}{16}e^{-1/4} + \frac{1}{64}e^{-1/2} + \frac{1}{16}e^{1/4} + \frac{1}{64}e^{1/2} \\
&\approx 0.25791494889765.
\end{aligned}$$

The exact value, to 15 digits, is 0.255251930412762. The error is 2.66×10^{-3}*, which is to be expected due to the use of few subintervals, and the fact that the Composite Trapezoidal Rule is only second-order-accurate.* □

Example 7.8.2 *We will use the Composite Simpson's Rule with* $n = 2$ *and* $m = 4$ *to evaluate the double integral*

$$\int_0^1\int_x^{2x} x^2 + y^3\,dy\,dx.$$

In this case, the domain of integration described by the limits is not a rectangle, but a triangle defined by the lines $y = x$*,* $y = 2x$*, and* $x = 1$*. The Composite Simpson's Rule with* $n = 2$ *subintervals is*

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right], \quad h = \frac{b-a}{n}.$$

If $a = 0$ *and* $b = 1$*, then* $h = (1-0)/2 = 1/2$*, and this simplifies to*

$$\int_0^{1/2} f(x)\,dx \approx \frac{1}{6}[f(0) + 4f(1/2) + f(1)].$$

We first use this rule to evaluate the "single" integral

$$\int_0^1 g(x)\,dx$$

where

$$g(x) = \int_x^{2x} x^2 + y^3\,dy.$$

This yields

$$\begin{aligned}\int_0^1\int_x^{2x} x^2+y^3\,dy\,dx &= \int_0^1 g(x)\,dx\\ &\approx \frac{1}{6}[g(0)+4g(1/2)+g(1)]\\ &\approx \frac{1}{6}\left[\int_0^0 0^2+y^3\,dy+4\int_{1/2}^1\left(\frac{1}{2}\right)^2+y^3\,dy+\int_1^2 1^2+y^3\,dy\right].\end{aligned}$$

The first integral will be zero, since the limits of integration are equal. To evaluate the second and third integrals, we use the Composite Simpson's Rule in the y-direction with $m = 4$. If we let k denote the step size in the y-direction, we have $k = (2x-x)/4 = x/4$, and therefore we have $k = 1/8$ for the second integral and $k = 1/4$ for the third. This yields

$$\begin{aligned}\int_0^1\int_x^{2x} x^2+y^3\,dy\,dx &\approx \frac{1}{6}\left[4\int_{1/2}^1\left(\frac{1}{2}\right)^2+y^3\,dy+\int_1^2 1^2+y^3\,dy\right]\\ &\approx \frac{1}{6}\left\{4\frac{1}{24}\left[\left(\frac{1}{4}+\left(\frac{1}{2}\right)^3\right)+4\left(\frac{1}{4}+\left(\frac{5}{8}\right)^3\right)+2\left(\frac{1}{4}+\left(\frac{3}{4}\right)^3\right)+\right.\right.\\ &\qquad 4\left(\frac{1}{4}+\left(\frac{7}{8}\right)^3\right)+\left.\left(\frac{1}{4}+1^3\right)\right]+\frac{1}{12}\left[(1+1^3)+4\left(1+\left(\frac{5}{4}\right)^3\right)+\right.\\ &\qquad \left.\left.2\left(1+\left(\frac{3}{2}\right)^3\right)+4\left(1+\left(\frac{7}{4}\right)^3\right)+(1+2^3)\right]\right\}\\ &\approx 1.03125.\end{aligned}$$

The exact value is 1. The error 3.125×10^{-2} is rather large, which is to be expected due to the poor distribution of nodes through the triangular domain of integration. A better distribution is achieved if we use $n = 4$ and $m = 2$, which yields the much more accurate approximation of 1.001953125. □

Exercise 7.8.1 *Write a* MATLAB *function* `I=quadcomptrap2d(f,a,b,c,d,m,n)` *that approximates (7.20) using the Composite Trapezoidal Rule with m subintervals in the x-direction and n subintervals in the y-direction.*

Exercise 7.8.2 *Generalize your function* `quadcomptrap2d` *from Exercise 7.8.1 so that the arguments* `c` *and* `d` *can be either scalars or function handles. If they are function handles, then your function approximates the integral*

$$I[f] = \int_a^b\int_{c(x)}^{d(x)} f(x,y)\,dy\,dx.$$

Hint: *use the* MATLAB *function* `isnumeric` *to determine whether* `c` *and* `d` *are numbers.*

Exercise 7.8.3 *Generalize your function* `quadcomptrap2d` *from Exercise 7.8.2 so that the arguments* `a`*,* `b`*,* `c` *and* `d` *can be either scalars or function handles. If* `a` *and* `b` *are function handles, then your function approximates the integral*

$$I[f] = \int_c^d \int_{a(y)}^{b(y)} f(x,y)\,dx\,dy.$$

Exercise 7.8.4 *Use the error formula for the Composite Trapezoidal Rule to obtain an error formula for a Cartesian product rule such as the one implemented in Exercise 7.8.1. As in that exercise, assume that m subintervals are used in the x-direction and n subintervals in the y-direction.* Hint: *first, apply the single-variable error formula to the integral*

$$\int_a^b g(x)\,dx, \quad g(x) = \int_c^d f(x,y)\,dy.$$

7.8.2 Higher Dimensions

The techniques presented in this section can readily be adapted for the purpose of approximating triple integrals. This adaptation is carried out in the following exercises.

Exercise 7.8.5 *Write a* MATLAB *function* `I=quadcomptrap3d(f,a,b,c,d,s,t,m,n,p)` *that approximates*

$$\int_a^b \int_c^d \int_s^t f(x,y,z)\,dz\,dy\,dx$$

using the Composite Trapezoidal Rule with m subintervals in the x-direction, n subintervals in the y-direction, and p subintervals in the z-direction. Hint: *use your function* `quadcomptrap2d` *from Exercise 7.8.1 to simplify the implementation.*

Exercise 7.8.6 *Modify your function* `quadcomptrap3d` *from Exercise 7.8.5 to obtain a function* `quadcompsimp3d` *that uses the Composite Simpson's Rule in each direction. Then, use an approach similar to that used in Exercise 7.8.4 to obtain an error formula for the Cartesian product rule used in* `quadcompsimp3d`*.*

Exercise 7.8.7 *Generalize your function* `quadcomptrap3d` *from Exercise 7.8.5 so that any of the arguments* `a`*,* `b`*,* `c`*,* `d`*,* `s` *and* `t` *can be either scalars or function handles. For example, if* `a` *and* `b` *are scalars,* `c` *and* `d` *are function handles that have two input arguments, and* `s` *and* `t` *are function handles that have one input argument, then your function approximates the integral*

$$I[f] = \int_a^b \int_{s(x)}^{t(x)} \int_{c(x,z)}^{d(x,z)} f(x,y,z)\,dy\,dz\,dx.$$

Hint: *use the* MATLAB *function* `isnumeric` *to determine whether any arguments are numbers, and the function* `nargin` *to determine how many arguments a function handle requires.*

In more than three dimensions, generalizations of quadrature rules are not practical, since the number of function evaluations needed to attain sufficient accuracy grows very rapidly as the number of dimensions increases. An alternative is the **Monte Carlo method**, which samples the integrand at n randomly selected points and attempts to compute the mean value of the integrand on the entire domain. The method converges rather slowly but its convergence rate depends only on n, not the number of dimensions.

Part IV

Nonlinear and Differential Equations

Chapter 8

Zeros of Nonlinear Functions

8.1 Nonlinear Equations in One Variable

To this point, we have only considered the solution of linear equations. We now explore the much more difficult problem of solving nonlinear equations of the form

$$f(\mathbf{x}) = \mathbf{0},$$

where $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$ can be any known function. A solution $\mathbf{x}$ of such a nonlinear equation is called a *root* of the equation, as well as a *zero* of the function f.

In general, nonlinear equations cannot be solved in a finite sequence of steps. As linear equations can be solved using *direct methods* such as Gaussian elimination, nonlinear equations usually require *iterative methods*. In iterative methods, an approximate solution is refined with each iteration until it is determined to be sufficiently accurate, at which time the iteration terminates. Since it is desirable for iterative methods to converge to the solution as rapidly as possible, it is necessary to be able to measure the speed with which an iterative method converges.

To that end, we assume that an iterative method generates a sequence of iterates $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ that converges to the exact solution $\mathbf{x}^*$. Ideally, we would like the error in a given iterate $\mathbf{x}_{k+1}$ to be much smaller than the error in the previous iterate $\mathbf{x}_k$. For example, if the error is raised to a power greater than 1 from iteration to iteration, then, because the error is typically less than 1, it will approach zero very rapidly. This leads to the following definition.

Definition 8.1.1 (Order and Rate of Convergence) *Let* $\{\mathbf{x}_k\}_{k=0}^{\infty}$ *be a sequence in* $\mathbb{R}^n$ *that converges to* $\mathbf{x}^* \in \mathbb{R}^n$ *and assume that* $\mathbf{x}_k \neq \mathbf{x}^*$ *for each* k. *We say that the* **order of convergence** *of* $\{\mathbf{x}_k\}$ *to* $\mathbf{x}^*$ *is* **order** r, *with* **asymptotic error constant** C, *if*

$$\lim_{k\to\infty} \frac{\|\mathbf{x}_{k+1} - \mathbf{x}^*\|}{\|\mathbf{x}_k - \mathbf{x}^*\|^r} = C,$$

where $r \geq 1$. *If* $r = 1$, *then the number* $\rho = -\log_{10} C$ *is called the* **asymptotic rate of convergence**.

If $r = 1$, and $0 < C < 1$, we say that convergence is *linear*. If $r = 1$ and $C = 0$, or if $1 < r < 2$ for any positive C, then we say that convergence is *superlinear*. If $r = 2$, then the method converges *quadratically*, and if $r = 3$, we say it converges *cubically*, and so on. Note that the value of C need only be bounded above in the case of linear convergence.

When convergence is linear, the asymptotic rate of convergence ρ indicates the number of correct decimal digits obtained in a single iteration. In other words, $\lfloor 1/\rho \rfloor + 1$ iterations are required to obtain an additional correct decimal digit, where $\lfloor x \rfloor$ is the "floor" of x, which is the largest integer that is less than or equal to x.

8.1.1 Existence and Uniqueness

For simplicity, we assume that the function $f : \mathbb{R}^n \to \mathbb{R}^m$ is continuous on the domain under consideration. Then, each equation $f_i(\mathbf{x}) = 0$, $i = 1, \ldots, m$, defines a hypersurface in $\mathbb{R}^m$. The solution of $f(\mathbf{x}) = 0$ is the intersection of these hypersurfaces, if the intersection is not empty. It is not hard to see that there can be a unique solution, infinitely many solutions, or no solution at all.

For a general equation $f(\mathbf{x}) = \mathbf{0}$, it is not possible to characterize the conditions under which a solution exists or is unique. However, in some situations, it is possible to determine existence analytically. For example, in one dimension, the Intermediate Value Theorem implies that if a continuous function $f(x)$ satisfies $f(a) \leq 0$ and $f(b) \geq 0$ where $a < b$, then $f(x) = 0$ for some $x \in (a, b)$.

Similarly, it can be concluded that $f(x) = 0$ for some $x \in (a, b)$ if the function $(x - z)f(x) \geq 0$ for $x = a$ and $x = b$, where $z \in (a, b)$. This condition can be generalized to higher dimensions. If $S \subset \mathbb{R}^n$ is an open, bounded set, and $(\mathbf{x} - \mathbf{z})^T f(\mathbf{x}) \geq 0$ for all $\mathbf{x}$ on the boundary of S and for some $\mathbf{z} \in S$, then $f(\mathbf{x}) = \mathbf{0}$ for some $\mathbf{x} \in S$. Unfortunately, checking this condition can be difficult in practice.

One useful result from calculus that can be used to establish existence and, in some sense, uniqueness of a solution is the *Inverse Function Theorem*, which states that if the Jacobian of f is nonsingular at a point $\mathbf{x}_0$, then f is invertible near $\mathbf{x}_0$ and the equation $f(\mathbf{x}) = \mathbf{y}$ has a unique solution for all $\mathbf{y}$ near $f(\mathbf{x}_0)$.

If the Jacobian of f at a point $\mathbf{x}_0$ is singular, then f is said to be *degenerate* at $\mathbf{x}_0$. Suppose that $\mathbf{x}_0$ is a solution of $f(\mathbf{x}) = \mathbf{0}$. Then, in one dimension, degeneracy means $f'(x_0) = 0$, and we say that x_0 is a *double root* of $f(x)$. Similarly, if $f^{(j)}(x_0) = 0$ for $j = 0, \ldots, m-1$, then x_0 is a root of multiplicity m. We will see that degeneracy can cause difficulties when trying to solve nonlinear equations.

8.1.2 Sensitivity of Solutions

The *absolute condition number* of a function $f(x)$ is a measure of how a perturbation in x, denoted by $x + \epsilon$ for some small ϵ, is amplified by $f(x)$. Using the Mean Value Theorem, we have

$$|f(x + \epsilon) - f(x)| = |f'(c)(x + \epsilon - x)| = |f'(c)||\epsilon|$$

where c is between x and $x + \epsilon$. With ϵ being small, the absolute condition number can be approximated by $|f'(x)|$, the factor by which the perturbation in x (ϵ) is amplified to obtain the perturbation in $f(x)$.

In solving a nonlinear equation in one dimension, we are trying to solve an *inverse problem*; that is, instead of computing $y = f(x)$ (the *forward problem*), we are computing $x = f^{-1}(0)$, assuming that f is invertible near the root. It follows from the differentiation rule

$$\frac{d}{dx}[f^{-1}(x)] = \frac{1}{f'(f^{-1}(x))}$$

that the condition number for solving $f(x) = 0$ is approximately $1/|f'(x^*)|$, where x^* is the solution. This discussion can be generalized to higher dimensions, where the condition number is measured using the norm of the Jacobian.

Using *backward error analysis*, we assume that the approximate solution $\hat{x} = \hat{f}^{-1}(0)$, obtained by evaluating an approximation of f^{-1} at the exact input $y = 0$, can also be viewed as evaluating the exact function f^{-1} at a nearby input $\hat{y} = \epsilon$. That is, the approximate solution $\hat{x} = f^{-1}(\epsilon)$ is the exact solution of a nearby problem.

From this viewpoint, it can be seen from a graph that if $|f'|$ is large near x^*, which means that the condition number of the problem $f(x) = 0$ is small (that is, the problem is *well-conditioned*), then even if ϵ is relatively large, $\hat{x} = f^{-1}(\epsilon)$ is close to x^*. On the other hand, if $|f'|$ is small near x^*, so that the problem is *ill-conditioned*, then even if ϵ is small, $\hat{x}$ can be far away from x^*. These contrasting situations are illustrated in Figure 8.1.

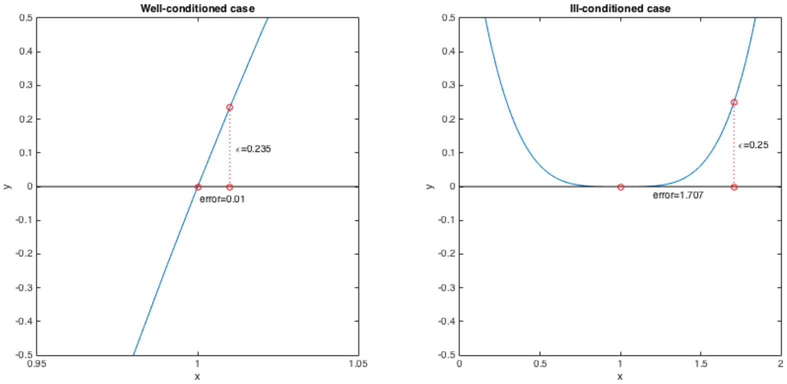


Figure 8.1: Left plot: Well-conditioned problem of solving $f(x) = 0$. $f'(x^*) = 24$, and an approximate solution $\hat{y} = f^{-1}(\epsilon)$ has small error relative to ϵ. Right plot: Ill-conditioned problem of solving $f(x) = 0$. $f'(x^*) = 0$, and $\hat{y}$ has large error relative to ϵ.

8.2 The Bisection Method

Suppose that $f(x)$ is a continuous function that changes sign on the interval $[a, b]$. Then, by the Intermediate Value Theorem, $f(x) = 0$ for some $x \in [a, b]$. How can we find the solution, knowing that it lies in this interval?

To determine how to proceed, we consider some examples in which such a sign change occurs. We work with the functions

$$f(x) = x - \cos x, \quad g(x) = e^x \cos(x^2),$$

on the intervals $[0, \pi/2]$ and $[0, \pi]$, respectively. The graphs of these functions are shown in Figure 8.2. It can be seen that $f(a)$ and $f(b)$ have different signs, and since both functions are continuous

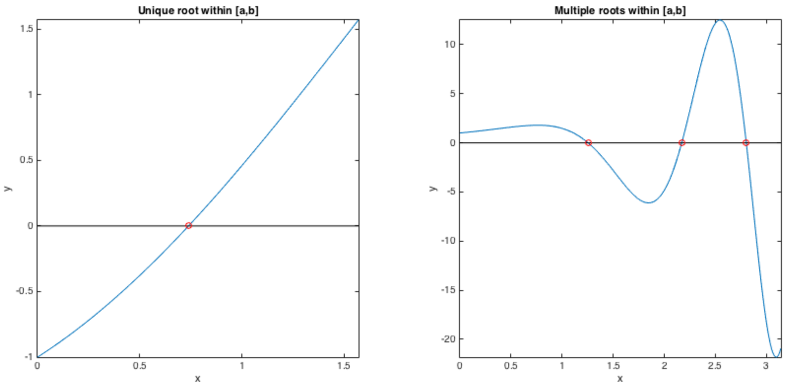


Figure 8.2: Illustrations of the Intermediate Value Theorem. Left plot: $f(x) = x - \cos x$ has a unique root on $[0, \pi/2]$. Right plot: $g(x) = e^x \cos(x^2)$ has multiple roots on $[0, \pi]$.

on $[a, b]$, the Intermediate Value Theorem guarantees the existence of a root in $[a, b]$. However, both of these intervals are rather large, so we cannot obtain a useful approximation of a root from this information alone.

At each root in these examples, $f(x)$ changes sign, so $f(x) > 0$ for x on one side of the root x^*, and $f(x) < 0$ on the other side. Therefore, if we can find two values a' and b' such that $f(a')$ and $f(b')$ have different signs, but a' and b' are very close to one another, then we can accurately approximate x^*.

Consider the first example in Figure 8.2, that has a unique root. We have $f(0) < 0$ and $f(\pi/2) > 0$. From the graph, we see that if we evaluate f at *any* other point x_0 in $(0, \pi/2)$, and we do not "get lucky" and happen to choose the root, then either $f(x_0) > 0$ or $f(x_0) < 0$. If $f(x_0) > 0$, then f has a root on $(0, x_0)$, because f changes sign on this interval. On the other hand, if $f(x_0) < 0$, then f has a root on $(x_0, \pi/2)$, because of a sign change. This is illustrated in Figure 8.3. The bottom line is, by evaluating $f(x)$ at an intermediate point within (a, b), the size of the interval in which we need to search for a root can be reduced.

The *Method of Bisection* attempts to reduce the size of the interval in which a solution is known to exist. Suppose that we evaluate $f(m)$, where $m = (a + b)/2$. If $f(m) = 0$, then we are done. Otherwise, f must change sign on the interval $[a, m]$ or $[m, b]$, since $f(a)$ and $f(b)$ have different signs and therefore $f(m)$ must have a different sign from one of these values.

Let us try this approach on the function $f(x) = x - \cos x$, on $[a, b] = [0, \pi/2]$. This example can be set up in MATLAB as follows:

```
>> a=0;
>> b=pi/2;
>> f=inline('x-cos(x)');
```

To help visualize the results of the computational process that we will carry out to find an approximate solution, we will also graph $f(x)$ on $[a, b]$:

```
>> dx=(b-a)/100;
>> x=a:dx:b;
>> plot(x,f(x))
```

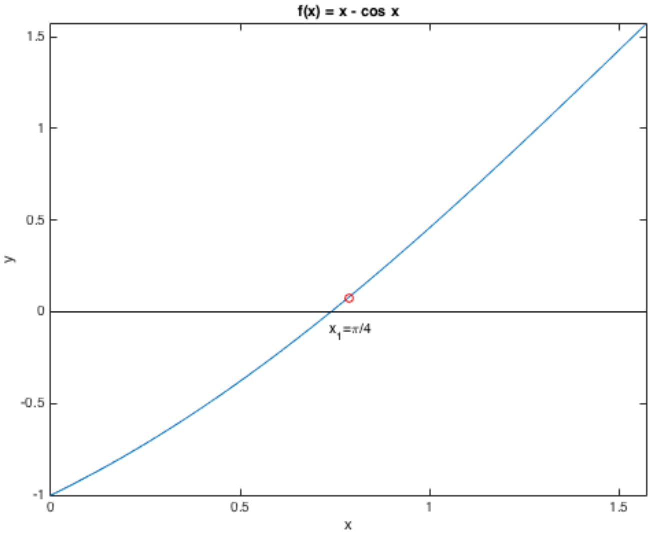


Figure 8.3: Because $f(\pi/4) > 0$, $f(x)$ has a root in $(0, \pi/4)$.

```
>> axis tight
>> xlabel('x')
>> ylabel('y')
>> title('f(x) = x - cos x')
>> hold on
>> plot([ 0 pi/2 ],[ 0 0 ],'k')
```

The last statement is used to plot the relevant portion of the x-axis, so that we can more easily visualize our progress toward computing a root.

Now, we can begin searching for an approximate root of $f(x)$. We can reduce the size of our search space by evaluating $f(x)$ at *any* point in (a, b), but for convenience, we choose the midpoint:

```
>> m=(a+b)/2;
>> plot(m,f(m),'ro')
```

The `plot` statement is used to plot the point $(m, f(m))$ on the graph of $f(x)$, using a red circle. You have now reproduced Figure 8.3.

Now, we examine the values of f at a, b and the midpoint m:

```
>> f(a)
ans =
    -1
>> f(m)
ans =
   0.078291382210901
>> f(b)
ans =
   1.570796326794897
```

We can see that $f(a)$ and $f(m)$ have different signs, so a root exists within (a, m). We can therefore update our search space $[a, b]$ accordingly:

```
>> b=m;
```

We then repeat the process, working with the midpoint of our new interval:

```
>> m=(a+b)/2
>> plot(m,f(m),'ro')
```

Now, it does not matter at which endpoint of our interval $f(x)$ has a positive value; we only need the signs of $f(a)$ and $f(b)$ to be different. Therefore, we can simply check whether the product of the values of f at the endpoints is negative:

```
>> f(a)*f(m)
ans =
   0.531180450812563
>> f(m)*f(b)
ans =
  -0.041586851697525
```

We see that the sign of f changes on (m, b), so we update a to reflect that this is our new interval to search:

```
>> a=m;
>> m=(a+b)/2
>> plot(m,f(m),'ro')
```

The progress toward a root can be seen in Figure 8.4.

Exercise 8.2.1 *Repeat this process a few more times: check whether f changes sign on (a, m) or (m, b), update $[a, b]$ accordingly, and then compute a new midpoint m. After computing some more midpoints, and plotting each one as we have been, what behavior can you observe? Are the midpoints converging, and if so, are they converging to a root of f? Check by evaluating f at each midpoint.*

We can continue this process until the interval of interest $[a, b]$ is sufficiently small, in which case we must be close to a solution. By including these steps in a loop, we obtain the following algorithm that implements the approach that we have been carrying out.

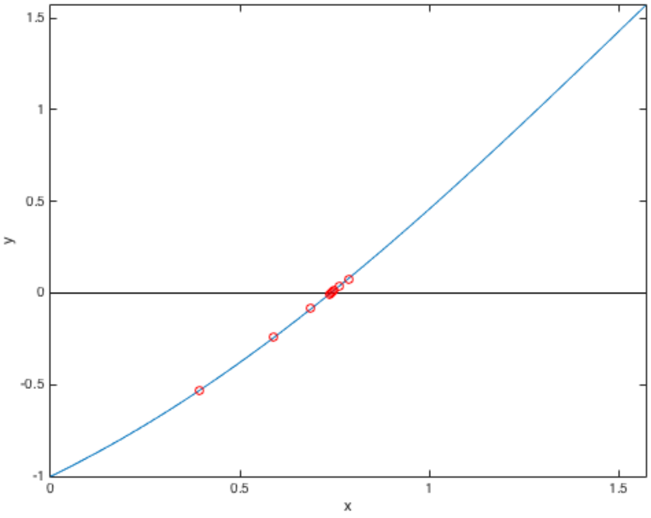


Figure 8.4: Progress of the Bisection method toward finding a root of $f(x) = x - \cos x$ on $(0, \pi/2)$

Algorithm 8.2.1 (Bisection) *Let f be a continuous function on the interval $[a, b]$ that changes sign on (a, b). The following algorithm computes an approximation x^* to a number x in (a, b) such that $f(x) = 0$.*

```
for $j = 1, 2, \ldots$ do
    $m = (a + b)/2$
    if $f(m) = 0$ or $b - a$ is sufficiently small then
        $x^* = m$
        return $x^*$
    end
    if $f(a)f(m) < 0$ then
        $b = m$
    else
        $a = m$
    end
end
```

At the beginning, it is known that (a, b) contains a solution. During each iteration, this algorithm updates the interval (a, b) by checking whether f changes sign in the first half (a, m), or in the second half (m, b). Once the correct half is found, the interval (a, b) is set equal to that half. Therefore, at the beginning of *each* iteration, it is known that the current interval (a, b) contains a solution.

Exercise 8.2.2 *Implement Algorithm 8.2.1 in a* MATLAB *function* `[x,niter]=bisection(f,a,b,tol)` *that accepts as input a function handle* `f` *for* $f(x)$*, the endpoints* `a` *and* `b` *of an interval* $[a,b]$*, and an absolute error tolerance* `tol` *so that the function will exit and return the latest midpoint* m *of* $[a,b]$*, in* `x` *when the length of the interval* $[a,b]$ *is less than* 2`tol`*, meaning that* m *is a distance less than* `tol` *from a root* x^* *of* f*. The output* `niter` *is the number of iterations; that is, the number of times that the midpoint* m *of* $[a,b]$ *is examined. Use the* `error` *function to ensure that if* f *does not change sign on* $[a,b]$*, the function immediately exits with an appropriate error message.*

In comparison to other methods, including some that we will discuss, bisection tends to converge rather slowly, but it is also guaranteed to converge. These qualities can be seen in the following result concerning the accuracy of bisection.

Theorem 8.2.2 *Let* f *be continuous on* $[a,b]$*, and assume that* $f(a)f(b) < 0$*. For each positive integer* n*, let* p_n *be the* n*th iterate that is produced by the bisection algorithm. Then the sequence* $\{p_n\}_{n=1}^{\infty}$ *converges to a number* p *in* (a,b) *such that* $f(p) = 0$*, and each iterate* p_n *satisfies*

$$|p_n - p| \leq \frac{b-a}{2^n}.$$

Exercise 8.2.3 *Use induction to prove Theorem 8.2.2.*

Exercise 8.2.4 *On a computer using the IEEE double-precision floating-point system, what is the largest number of iterations of bisection that is practical to perform? Justify your answer.*

It should be noted that because the nth iterate can lie anywhere within the interval (a,b) that is used during the nth iteration, it is possible that the error bound given by this theorem may be quite conservative.

Example 8.2.3 *We seek a solution of the equation* $f(x) = 0$*, where*

$$f(x) = x^2 - x - 1.$$

Because $f(1) = -1$ *and* $f(2) = 1$*, and* f *is continuous, we can use the Intermediate Value Theorem to conclude that* $f(x) = 0$ *has a solution in the interval* $(1,2)$*, since* $f(x)$ *must assume every value between* -1 *and* 1 *in this interval.*

We use the method of bisection *to find a solution. First, we compute the midpoint of the interval, which is* $(1+2)/2 = 1.5$*. Since* $f(1.5) = -0.25$*, we see that* $f(x)$ *changes sign between* $x = 1.5$ *and* $x = 2$*, so we can apply the Intermediate Value Theorem again to conclude that* $f(x) = 0$ *has a solution in the interval* $(1.5, 2)$*.*

Continuing this process, we compute the midpoint of the interval $(1.5, 2)$*, which is* $(1.5+2)/2 = 1.75$*. Since* $f(1.75) = 0.3125$*, we see that* $f(x)$ *changes sign between* $x = 1.5$ *and* $x = 1.75$*, so we conclude that there is a solution in the interval* $(1.5, 1.75)$*. The following table shows the outcome of several more iterations of this procedure. Each row shows the current interval* (a,b) *in which we know that a solution exists, as well as the midpoint of the interval, given by* $(a+b)/2$*, and the value of* f *at the midpoint. Note that from iteration to iteration, only one of* a *or* b *changes, and the endpoint that changes is always set equal to the midpoint.*

a	b	$m = (a+b)/2$	$f(m)$
1	2	1.5	-0.25
1.5	2	1.75	0.3125
1.5	1.75	1.625	0.015625
1.5	1.625	1.5625	-0.12109
1.5625	1.625	1.59375	-0.053711
1.59375	1.625	1.609375	-0.019287
1.609375	1.625	1.6171875	-0.0018921
1.6171875	1.625	1.62109325	0.0068512
1.6171875	1.62109325	1.619140625	0.0024757
1.6171875	1.619140625	1.6181640625	0.00029087

The correct solution, to ten decimal places, is 1.6180339887*, which is the number known as the* golden ratio. □

Exercise 8.2.5 *The function* $f(x) = x^2 - 2x - 2$ *has two real roots, one positive and one negative. Find two disjoint intervals* $[a_1, b_1]$ *and* $[a_2, b_2]$ *that can be used with bisection to find the negative and positive roots, respectively. Why is it not practical to use a much larger interval that contains both roots, so that bisection can supposedly find one of them?*

For this method, it is easier to determine the order of convergence if we use a different measure of the error in each iterate x_k. Since each iterate is contained within an interval $[a_k, b_k]$ where $b_k - a_k = 2^{-k}(b-a)$, with $[a, b]$ being the original interval, it follows that we can bound the error $x_k - x^*$ by $e_k = b_k - a_k$. Using this measure, we can easily conclude that bisection converges linearly, with asymptotic error constant 1/2.

8.3 Fixed-Point Iteration

A nonlinear equation of the form $f(x) = 0$ can be rewritten to obtain an equation of the form

$$x = g(x),$$

in which case the solution is a *fixed point* of the function g.

8.3.1 Successive Substitution

The formulation of the original problem $f(x) = 0$ into one of the form $x = g(x)$ leads to a simple solution method known as *fixed-point iteration*, or *simple iteration*, which we now describe.

Algorithm 8.3.1 (Fixed-Point Iteration) *Let g be a continuous function defined on the interval $[a,b]$. The following algorithm computes a number $x^* \in (a,b)$ that is a solution to the equation $g(x) = x$.*

```
Choose an initial guess $x_0$ in $[a,b]$.
for $k = 0, 1, 2, \ldots$ do
    $x_{k+1} = g(x_k)$
    if $|x_{k+1} - x_k|$ is sufficiently small then
        $x^* = x_{k+1}$
        return $x^*$
    end
end
```

When rewriting this equation in the form $x = g(x)$, it is essential to choose the function g wisely. One guideline is to choose $g(x) = x - \phi(x)f(x)$, where the function $\phi(x)$ is, ideally, nonzero except possibly at a solution of $f(x) = 0$. This can be satisfied by choosing $\phi(x)$ to be constant, but this can fail, as the following example illustrates.

Example 8.3.2 *Consider the equation*

$$x + \ln x = 0.$$

By the Intermediate Value Theorem, this equation has a solution in the interval $[0.5, 0.6]$. Furthermore, this solution is unique. To see this, let $f(x) = x + \ln x$. Then $f'(x) = 1 + 1/x > 0$ on the domain of f, which means that f is increasing on its entire domain. Therefore, it is not possible for $f(x) = 0$ to have more than one solution.

We consider using Fixed-point Iteration to solve the equivalent equation

$$x = g(x) = x - \phi(x)f(x) = x - (1)(x + \ln x) = -\ln x.$$

That is, we choose $\phi(x) \equiv 1$. Let us try applying Fixed-point Iteration in Matlab*:*

```
>> g=inline('-log(x)')
g =
     Inline function:
     g(x) = -log(x)
>> x=0.55;
>> x=g(x)
x =
   0.597837000755620
>> x=g(x)
x =
   0.514437136173803
>> x=g(x)
x =
   0.664681915480620
```

Exercise 8.3.1 *Try this for a few more iterations. What happens?*

Clearly, we need to use a different approach for converting our original equation $f(x) = 0$ to an equivalent equation of the form $x = g(x)$.

What went wrong? To help us answer this question, we examine the error $e_k = x_k - x^*$. Suppose that $x = g(x)$ has a solution x^* in $[a, b]$, as it does in the preceding example, and that g is also *continuously* differentiable on $[a, b]$, as was the case in that example. We can use the Mean Value Theorem to obtain

$$e_{k+1} = x_{k+1} - x^* = g(x_k) - g(x^*) = g'(\xi_k)(x_k - x^*) = g'(\xi_k)e_k,$$

where ξ_k lies between x_k and x^*.

We do not yet know the conditions under which Fixed-Point Iteration will converge, but if it *does* converge, then it follows from the continuity of g' at x^* that it does so linearly with asymptotic error constant $|g'(x^*)|$, since, by the definition of ξ_k and the continuity of g',

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|} = \lim_{k\to\infty} |g'(\xi_k)| = |g'(x^*)|.$$

Recall, though, that for linear convergence, the asymptotic error constant $C = |g'(x^*)|$ must satisfy $C < 1$. Unfortunately, with $g(x) = -\ln x$, we have $|g'(x)| = |-1/x| > 1$ on the interval $[0.5, 0.6]$, so it is not surprising that the iteration diverged.

What if we could convert the original equation $f(x) = 0$ into an equation of the form $x = g(x)$ so that g' satisfied $|g'(x)| < 1$ on an interval $[a, b]$ where a fixed point was known to exist? What we can do is take advantage of the differentiation rule

$$\frac{d}{dx}[f^{-1}(x)] = \frac{1}{f'(f^{-1}(x)}$$

and apply $g^{-1}(x) = e^{-x}$ to both sides of the equation $x = g(x)$ to obtain

$$g^{-1}(x) = g^{-1}(g(x)) = x,$$

which simplifies to

$$x = e^{-x}.$$

The function $g(x) = e^{-x}$ satisfies $|g'(x)| < 1$ on $[0.5, 0.6]$, as $g'(x) = -e^{-x}$, and $e^{-x} < 1$ when the argument x is positive. What happens if you try Fixed-point Iteration with this choice of g?

```
>> g=inline('exp(-x)')
g =
     Inline function:
     g(x) = exp(-x)
>> x=0.55;
>> x=g(x)
x =
   0.576949810380487
>> x=g(x)
x =
   0.561608769952327
>> x=g(x)
x =
   0.570290858658895
```

This is more promising.

Exercise 8.3.2 *Continue this process to confirm that the iteration is in fact converging.*

□

Having seen what can go wrong if we are not careful in applying Fixed-Point Iteration, we should now address the questions of existence and uniqueness of a solution to the modified problem $g(x) = x$. The following result, first proved in [9], answers the first of these questions.

Theorem 8.3.3 (Brouwer's Fixed Point Theorem) Let g be continuous on $[a, b]$. If $g(x) \in [a, b]$ for each $x \in [a, b]$, then g has a fixed point in $[a, b]$.

Exercise 8.3.3 *Use the Intermediate Value Theorem (see Theorem A.1.10) to prove Theorem 8.3.3.*

Given a continuous function g that is known to have a fixed point in an interval $[a, b]$, we can try to find this fixed point by repeatedly evaluating g at points in $[a, b]$ until we find a point x for which $g(x) = x$. This is the essence of the method of Fixed-point Iteration. However, just because g has a fixed point does not mean that this iteration will necessarily converge. We will now investigate this further.

8.3.2 Convergence Analysis

Under what circumstances will fixed-point iteration converge to a fixed point x^*? We say that a function g that is continuous on $[a, b]$ satisfies a *Lipschitz condition* on $[a, b]$ if there exists a positive constant L such that

$$|g(x) - g(y)| \leq L|x - y|, \quad x, y \in [a, b].$$

The constant L is called a *Lipschitz constant*. If, in addition, $L < 1$, we say that g is a *contraction* on $[a, b]$.

If we denote the error in x_k by $e_k = x_k - x^*$, we can see from the fact that $g(x^*) = x^*$ that if $x_k \in [a, b]$, then

$$|e_{k+1}| = |x_{k+1} - x^*| = |g(x_k) - g(x^*)| \leq L|x_k - x^*| \leq L|e_k| < |e_k|.$$

Therefore, if g satisfies the conditions of the Brouwer Fixed-Point Theorem, and g is a contraction on $[a, b]$, and $x_0 \in [a, b]$, then fixed-point iteration is *convergent*; that is, x_k converges to x^*.

Furthermore, the fixed point x^* must be *unique*, for if there exist two distinct fixed points x^* and y^* in $[a, b]$, then, by the Lipschitz condition, we have

$$0 < |x^* - y^*| = |g(x^*) - g(y^*)| \leq L|x^* - y^*| < |x^* - y^*|,$$

which is a contradiction. Therefore, we must have $x^* = y^*$. We summarize our findings with the statement of the following result, first established in [3].

Theorem 8.3.4 (Contraction Mapping Theorem) *Let g be a continuous function on the interval $[a,b]$. If $g(x) \in [a,b]$ for each $x \in [a,b]$, and if there exists a constant $0 < L < 1$ such that*

$$|g(x) - g(y)| \leq L|x - y|, \quad x, y \in [a,b],$$

then g has a unique fixed point x^* in $[a,b]$, and the sequence of iterates $\{x_k\}_{k=0}^{\infty}$ converges to x^*, for any initial guess $x_0 \in [a,b]$.

In general, when fixed-point iteration converges, it does so at a rate that varies inversely with the Lipschitz constant L.

If g satisfies the conditions of the Contraction Mapping Theorem with Lipschitz constant L, then Fixed-point Iteration achieves at least linear convergence, with an asymptotic error constant that is bounded above by L. This value can be used to estimate the number of iterations needed to obtain an additional correct decimal digit, but it can also be used to estimate the *total* number of iterations needed for a specified degree of accuracy.

From the Lipschitz condition, we have, for $k \geq 1$,

$$|x_k - x^*| \leq L|x_{k-1} - x^*| \leq L^k|x_0 - x^*|.$$

From

$$|x_0 - x^*| \leq |x_0 - x_1 + x_1 - x^*| \leq |x_0 - x_1| + |x_1 - x^*| \leq |x_0 - x_1| + L|x_0 - x^*|$$

we obtain

$$|x_k - x^*| \leq \frac{L^k}{1-L}|x_1 - x_0|. \tag{8.1}$$

We can bound the number of iterations after performing a single iteration, as long as the Lipschitz constant L is known.

Exercise 8.3.4 *Use (8.1) to determine a lower bound on the number of iterations required to ensure that $|x_k - x^*| \leq \epsilon$ for some error tolerance ϵ.*

We can now develop a practical implementation of Fixed-Point Iteration.

Exercise 8.3.5 *Write a* MATLAB *function* `[x,niter]=fixedpt(g,x0,tol)` *that implements Algorithm 8.3.1 to solve the equation $x = g(x)$ with initial guess x_0, except that instead of simply using the absolute difference between iterates to test for convergence, the error estimate (8.1) is compared to the specified tolerance* `tol`*, and a maximum number of iterations is determined based on the result of Exercise 8.3.4. The output arguments* `x` *and* `niter` *are the computed solution x^* and number of iterations, respectively. Test your function on the equation from Example 8.3.2.*

We know that Fixed-point Iteration will converge to the unique fixed point in $[a,b]$ if g satisfies the conditions of the Contraction Mapping Theorem. However, if g is differentiable on $[a,b]$, its derivative can be used to obtain an alternative criterion for convergence that can be more practical than computing the Lipschitz constant L. If we denote the error in x_k by $e_k = x_k - x^*$, we can see from the Mean Value Theorem and the fact that $g(x^*) = x^*$ that

$$e_{k+1} = x_{k+1} - x^* = g(x_k) - g(x^*) = g'(\xi_k)(x_k - x^*)$$

where ξ_k lies between x_k and x^*. However, from

$$|g'(\xi_k)| = \left|\frac{g(x_k) - g(x^*)}{x_k - x^*}\right|$$

it follows that if $|g'(x)| \leq L$ on (a, b), where $L < 1$, then the Contraction Mapping Theorem applies. This leads to the following result.

Theorem 8.3.5 (Fixed-point Theorem) *Let g be a continuous function on the interval $[a, b]$, and let g be differentiable on (a, b). If $g(x) \in [a, b]$ for each $x \in [a, b]$, and if there exists a constant $L < 1$ such that*

$$|g'(x)| \leq L, \quad x \in (a, b),$$

then the sequence of iterates $\{x_k\}_{k=0}^{\infty}$ converges to the unique fixed point x^* of g in $[a, b]$, for any initial guess $x_0 \in [a, b]$.

Using Taylor expansion of the error around x^*, it can also be shown that if g' is continuous at x^* and $|g'(x^*)| < 1$, then Fixed-point Iteration is *locally convergent*; that is, it converges if x_0 is chosen sufficiently close to x^*.

It can be seen from the preceding discussion why $g'(x)$ must be bounded away from 1 on (a, b), as opposed to the weaker condition $|g'(x)| < 1$ on (a, b). If $g'(x)$ is allowed to approach 1 as x approaches a point $c \in (a, b)$, then it is possible that the error e_k might not approach zero as k increases, in which case Fixed-point Iteration would not converge.

Exercise 8.3.6 *Find a function g and interval $[a, b]$ such that g continuous on $[a, b]$ and differentiable on (a, b), but does* not *satisfy a Lipschitz condition on $[a, b]$ for any Lipschitz constant L.*

The derivative can also be used to indicate why Fixed-point Iteration might *not* converge.

Example 8.3.6 *The function $g(x) = x^2 + \frac{3}{16}$ has two fixed points, $x_1^* = 1/4$ and $x_2^* = 3/4$, as can be determined by solving the quadratic equation $x^2 + \frac{3}{16} = x$. If we consider the interval $[0, 3/8]$, then g satisfies the conditions of the Fixed-point Theorem, as $g'(x) = 2x < 1$ on this interval, and therefore Fixed-point Iteration will converge to x_1^* for any $x_0 \in [0, 3/8]$.*

On the other hand, $g'(3/4) = 2(3/4) = 3/2 > 1$. Therefore, it is not possible for g to satisfy the conditions of the Fixed-point Theorem. Furthemore, if x_0 is chosen so that $1/4 < x_0 < 3/4$, then Fixed-point Iteration will converge to $x_1^* = 1/4$, whereas if $x_0 > 3/4$, then Fixed-point Iteration diverges. □

The fixed point $x_2^* = 3/4$ in the preceding example is an *unstable fixed point* of g, meaning that *no* choice of x_0 yields a sequence of iterates that converges to x_2^*. The fixed point $x_1^* = 1/4$ is a *stable fixed point* of g, meaning that *any* choice of x_0 that is sufficiently close to x_1^* yields a sequence of iterates that converges to x_1^*.

The preceding example shows that Fixed-point Iteration applied to an equation of the form $x = g(x)$ can fail to converge to a fixed point x^* if $|g'(x^*)| > 1$. We wish to determine whether this condition indicates non-convergence in general. If $|g'(x^*)| > 1$, and g' is continuous in a neighborhood of x^*, then there exists an interval $|x - x^*| \leq \delta$ such that $|g'(x)| > 1$ on the interval. If x_k lies within this interval, it follows from the Mean Value Theorem that

$$|x_{k+1} - x^*| = |g(x_k) - g(x^*)| = |g'(\eta)||x_k - x^*|,$$

where η lies between x_k and x^*. Because η is also in this interval, we have

$$|x_{k+1} - x^*| > |x_k - x^*|.$$

In other words, the error in the iterates *increases* whenever they fall within a sufficiently small interval containing the fixed point. Because of this increase, the iterates must eventually fall outside of the interval. Therefore, it is not possible to find a k_0, for any given δ, such that $|x_k - x^*| \leq \delta$ for all $k \geq k_0$. We have thus proven the following result.

Theorem 8.3.7 *Let g have a fixed point at x^*, and let g' be continuous in a neighborhood of x^*. If $|g'(x^*)| > 1$, then Fixed-point Iteration does not converge to x^* for any initial guess x_0 except in a finite number of iterations.*

Now, suppose that in addition to the conditions of the Fixed-point Theorem, we assume that $g'(x^*) = 0$, and that g is twice continuously differentiable on $[a, b]$. Then, using Taylor's Theorem, we obtain

$$e_{k+1} = g(x_k) - g(x^*) = g'(x^*)(x_k - x^*) + \frac{1}{2}g''(\xi_k)(x_k - x^*)^2 = \frac{1}{2}g''(\xi_k)e_k^2,$$

where ξ_k lies between x_k and x^*. It follows that for any initial iterate $x_0 \in [a, b]$, Fixed-point Iteration converges at least quadratically, with asymptotic error constant $|g''(x^*)/2|$. Later, this will be exploited to obtain a quadratically convergent method for solving nonlinear equations of the form $f(x) = 0$.

8.3.3 Relaxation

Now that we understand the convergence behavior of Fixed-point Iteration, we consider the application of Fixed-point Iteration to the solution of an equation of the form $f(x) = 0$.

Example 8.3.8 *We use Fixed-point Iteration to solve the equation $f(x) = 0$, where $f(x) = x - \cos x - 2$. It makes sense to work with the equation $x = g(x)$, where $g(x) = \cos x + 2$.*

Where should we look for a solution to this equation? For example, consider the interval $[0, \pi/4]$. On this interval, $g'(x) = -\sin x$, which certainly satisfies the condition $|g'(x)| \leq \rho < 1$ where $\rho = \sqrt{2}/2$, but g does not *map this interval into itself, as required by the Brouwer Fixed-point Theorem.*

On the other hand, if we consider the interval $[1, 3]$, it can readily be confirmed that $g(x)$ maps this interval to itself, as $1 \leq 2 + \cos x \leq 3$ for all real x, so a fixed point exists in this interval.

First, let's set up a figure with a graph of $g(x)$:

```
>> x=1:0.01:3;
>> figure(1)
>> plot(x,g(x))
>> hold on
>> plot([ 1 3 ],[ 1 3 ],'k--')
>> xlabel('x')
>> ylabel('y')
>> title('g(x) = cos x + 2')
```

Exercise 8.3.7 *Go ahead and try Fixed-point Iteration on $g(x)$, with initial guess $x_0 = 2$. What happens?*

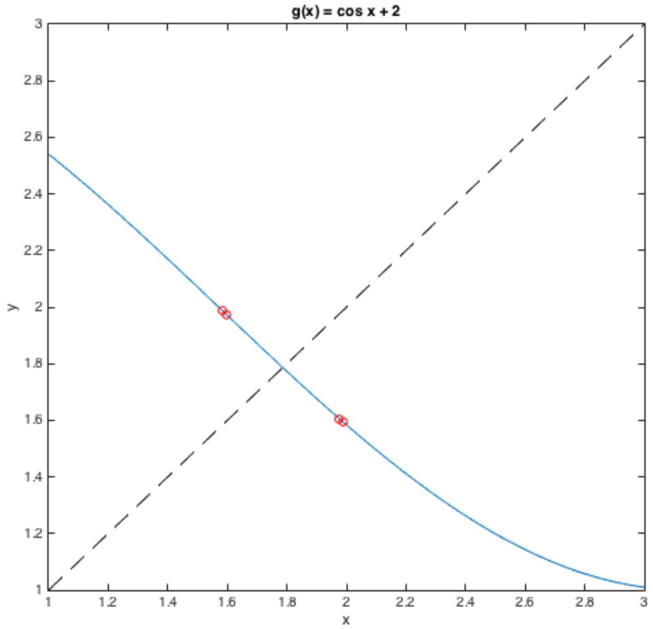


Figure 8.5: Fixed-point Iteration applied to $g(x) = \cos x + 2$.

The behavior is quite interesting, as the iterates seem to bounce back and forth. This is illustrated in Figure 8.5. Continuing, we see that convergence is achieved, but it is quite slow. An examination of the derivative explains why: $g'(x) = -\sin x$, and we have $|g'(\pi/2)| = |-\sin \pi/2| = 1$, so the conditions of the Fixed-point Theorem are *not* satisfied–in fact, we could not be assured of convergence at all, though it does occur in this case.

An examination of the iterates shown in Figure 8.5, along with an indication of the solution, suggests how convergence can be accelerated. What if we used the *average* of x and $g(x)$ at each iteration? That is, we solve $x = h(x)$, where

$$h(x) = \frac{1}{2}[x + g(x)] = \frac{1}{2}[x + \cos x + 2].$$

You can confirm that if $x = h(x)$, then $f(x) = 0$. However, we have

$$h'(x) = \frac{1}{2}[1 - \sin x],$$

and how large can this be on the interval $[1, 3]$? In this case, the Fixed-point Theorem *does* apply.

Exercise 8.3.8 *Try Fixed-point Iteration with $h(x)$, and with initial guess $x_0 = 2$. What behavior can you observe?*

The lesson to be learned here is that the most straightforward choice of $g(x)$ is not always the wisest–the key is to minimize the size of the derivative near the solution. □

As previously discussed, a common choice for a function $g(x)$ to use with Fixed-point Iteration to solve the equation $f(x) = 0$ is a function of the form $g(x) = x - \phi(x)f(x)$, where $\phi(x)$ is nonzero.

Clearly, the simplest choice of $\phi(x)$ is a constant function $\phi(x) \equiv \lambda$, but it is important to choose λ so that Fixed-point Iteration with $g(x)$ will converge.

Suppose that x^* is a solution of the equation $f(x) = 0$, and that f is continuously differentiable in a neighborhood of x^*, with $f'(x^*) = \alpha > 0$. Then, by continuity, there exists an interval $[x^* - \delta, x^* + \delta]$ containing x^* on which $m \leq f'(x) \leq M$, for positive constants m and M. We can then prove the following results.

Exercise 8.3.9 *Let* $f'(x^*) = \alpha > 0$. *Prove that there exist* $\delta, \lambda > 0$ *such that on the interval* $|x - x^*| \leq \delta$, *there exists* $L < 1$ *such that* $|g'(x)| \leq L$, *where* $g(x) = x - \lambda f(x)$. *What is the value of* L? Hint: *choose* λ *so that upper and lower bounds on* $g'(x)$ *are equal to* $\pm L$.

Exercise 8.3.10 *Under the assumptions of Exercise 8.3.9, prove that if* $I_\delta = [x^* - \delta, x^* + \delta]$, *and* $\lambda > 0$ *is chosen so that* $|g'(x)| \leq L < 1$ *on* I_δ, *then* g *maps* I_δ *into itself.*

We conclude from the preceding two exercises that the Fixed-point Theorem applies, and Fixed-point Iteration converges linearly to x^* for any choice of x_0 in $[x^* - \delta, x^* + \delta]$, with asymptotic error constant $|1 - \lambda\alpha| \leq L$.

In summary, if f is continuously differentiable in a neighborhood of a root x^* of $f(x) = 0$, and $f(x^*)$ is nonzero, then there exists a constant λ such that Fixed-point Iteration with $g(x) = x - \lambda f(x)$ converges to x^* for x_0 chosen sufficiently close to x^*. This approach to Fixed-point Iteration, with a constant ϕ, is known as *relaxation*.

Convergence can be accelerated by allowing λ to vary from iteration to iteration. Intuitively, an effective choice is to try to minimize $|g'(x)|$ near x^* by setting $\lambda = 1/f'(x_k)$, for each k, so that $g'(x_k) = 1 - \lambda f'(x_k) = 0$. This results in linear convergence with an asymptotic error constant of 0, which indicates faster than linear convergence. We will see that convergence is actually quadratic.

8.4 Newton's Method and the Secant Method

To develop a more effective method for solving this problem of computing a solution to $f(x) = 0$, we can address the following questions:

- Are there cases in which the problem easy to solve, and if so, how do we solve it in such cases?
- Is it possible to apply our method of solving the problem in these "easy" cases to more general cases?

A recurring theme in this book is that these questions are useful for solving a variety of problems.

8.4.1 Newton's Method

For the problem at hand, we ask whether the equation $f(x) = 0$ is easy to solve for any particular choice of the function f. Certainly, if f is a linear function, then it has a formula of the form $f(x) = m(x - a) + b$, where m and b are constants and $m \neq 0$. Setting $f(x) = 0$ yields the equation

$$m(x - a) + b = 0,$$

which can easily be solved for x to obtain the unique solution

$$x = a - \frac{b}{m}.$$

We now consider the case where f is not a linear function. For example, recall when we solved the equation $f(x) = x - \cos x$ using Bisection. In Figure 8.2, note that near the root, f is well-approximated by a linear function. How shall we exploit this?

Using Taylor's theorem, it is simple to construct a linear function that approximates $f(x)$ near a given point x_0. This function is simply the first Taylor polynomial of $f(x)$ with center x_0,

$$P_1(x) = f(x_0) + f'(x_0)(x - x_0).$$

This function has a useful geometric interpretation, as its graph is the tangent line of $f(x)$ at the point $(x_0, f(x_0))$.

We will illustrate this in MATLAB, for the example $f(x) = x - \cos x$, and initial guess $x_0 = 1$. The following code plots $f(x)$ and the tangent line at $(x_0, f(x_0))$.

```
>> f=inline('x - cos(x)');
>> a=0.5;
>> b=1.5;
>> h=0.01;
>> x=a:h:b;
>> % plot f(x) on [a,b]
>> plot(x,f(x))
>> hold on
>> % plot x-axis
>> plot([ a b ],[ 0 0 ],'k')
>> x0=1;
>> % plot initial guess on graph of f(x)
>> plot(x0,f(x0),'ro')
>> % f'(x) = 1 + sin(x)
>> % slope of tangent line: m = f'(x0)
>> m=1+sin(x0);
>> % plot tangent line using points x=a,b
>> plot([ a b ],[ f(x0) + m*([ a b ] - x0) ],'r')
>> xlabel('x')
>> ylabel('y')
```

Exercise 8.4.1 *Rearrange the formula for the tangent line approximation $P_1(x)$ to obtain a formula for its x-intercept x_1. Compute this value in* MATLAB *and plot the point $(x_1, f(x_1))$ as a red '+'.*

The plot that should be obtained from the preceding code and Exercise 8.4.1 is shown in Figure 8.6.

As can be seen in Figure 8.6, we can obtain an approximate solution to the equation $f(x) = 0$ by determining where the linear function $P_1(x)$ is equal to zero. If the resulting value, x_1, is not a

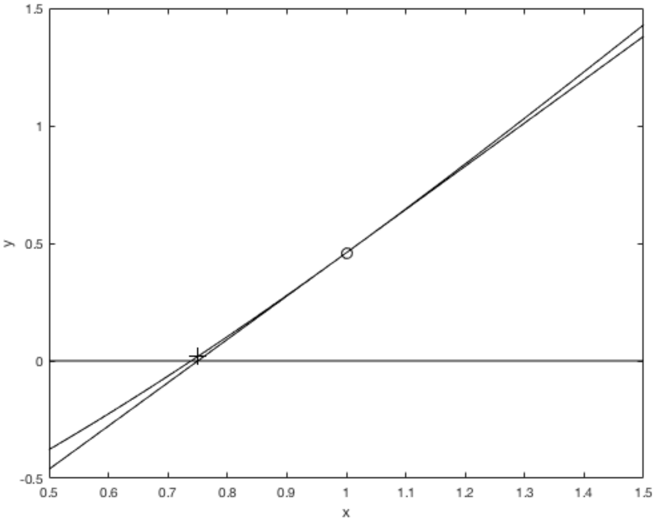


Figure 8.6: Approximating a root of $f(x) = x - \cos x$ using the tangent line of $f(x)$ at $x_0 = 1$.

solution, then we can repeat this process, approximating f by a linear function near x_1 and once again determining where this approximation is equal to zero.

Exercise 8.4.2 *Modify the above* MATLAB *statements to effectively "zoom in" on the graph of $f(x)$ near $x = x_1$, the zero of the tangent line approximation $P_1(x)$ above. Use the tangent line at $(x_1, f(x_1))$ to compute a second approximation x_2 of the root of $f(x)$. What do you observe?*

The algorithm that results from repeating this process of approximating a root of $f(x)$ using tangent line approximations is known as *Newton's method*, which we now describe in detail.

Algorithm 8.4.1 (Newton's Method) *Let $f : \mathbb{R} \to \mathbb{R}$ be a differentiable function. The following algorithm computes an approximate solution x^* to the equation $f(x) = 0$.*

```
Choose an initial guess $x_0$
for $k = 0, 1, 2, \dots$ do
    if $f(x_k)$ is sufficiently small then
        $x^* = x_k$
        return $x^*$
    end
    $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$
    if $|x_{k+1} - x_k|$ is sufficiently small then
        $x^* = x_{k+1}$
        return $x^*$
    end
end
```

Exercise 8.4.3 *Write a* MATLAB *function* `[x,niter]=newton(f,fp,x0,tol)` *that implements Algorithm 8.4.1 to solve the equation $f(x) = 0$ using Newton's Method with initial guess x_0 and absolute error tolerance* `tol`*. The second input argument* `fp` *must be a function handle for $f'(x)$. The output arguments* `x` *and* `niter` *are the computed solution x^* and number of iterations, respectively.*

It is worth noting that Newton's Method is equivalent to performing Fixed-point Iteration with $g(x) = x - \lambda_k f(x)$, where $\lambda_k = 1/f'(x_k)$ for the kth iteration that computes x_{k+1}. Recall that this choice of constant was chosen in order make relaxation as rapidly convergent as possible.

In fact, when Newton's method converges, it does so very rapidly. However, it can be difficult to ensure convergence, particularly if $f(x)$ has horizontal tangents near the solution x^*. Typically, it is necessary to choose a starting iterate x_0 that is close to x^*. As the following result indicates, such a choice, if close enough, is indeed sufficient for convergence [36, Theorem 1.7].

Theorem 8.4.2 (Convergence of Newton's Method) *Let f be twice continuously differentiable on the interval $[a, b]$, and suppose that $f(c) = 0$ and $f'(c) \neq 0$ for some $c \in [a, b]$. Then there exists a $\delta > 0$ such that Newton's Method applied to $f(x)$ converges to c for any initial guess x_0 in the interval $[c - \delta, c + \delta]$.*

Example 8.4.3 *We will use Newton's Method to compute a root of $f(x) = x - \cos x$. Since $f'(x) = 1 + \sin x$, it follows that in Newton's Method, we can obtain the next iterate x_{n+1} from the previous iterate x_n by*

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n - \cos x_n}{1 + \sin x_n} = \frac{x_n \sin x_n + \cos x_n}{1 + \sin x_n}.$$

We choose our starting iterate $x_0 = 1$, and compute the next several iterates as follows:

$$\begin{aligned}
x_1 &= \frac{(1)\sin 1 + \cos 1}{1 + \sin 1} = 0.750363867840244 \\
x_2 &= \frac{x_1 \sin x_1 + \cos x_1}{1 + \sin x_1} = 0.739112890911362 \\
x_3 &= 0.739085133385284 \\
x_4 &= 0.739085133215161 \\
x_5 &= 0.739085133215161.
\end{aligned}$$

Since the fourth and fifth iterates agree to 15 decimal places, we assume that 0.739085133215161 is a correct solution to $f(x) = 0$, to at least 15 decimal places. □

Exercise 8.4.4 *How many correct decimal places are obtained in each x_k in the preceding example? What does this suggest about the order of convergence of Newton's method?*

We can see from this example that Newton's method converged to a root far more rapidly than the Bisection method did when applied to the same function. However, unlike Bisection, Newton's method is not guaranteed to converge. Whereas Bisection is guaranteed to converge to a root in $[a, b]$ if $f(a)f(b) < 0$, we must be careful about our choice of the initial guess x_0 for Newton's method, as the following example illustrates.

Example 8.4.4 *Newton's Method can be used to compute the reciprocal of a number* a without performing any divisions. *The solution,* $1/a$, *satisfies the equation* $f(x) = 0$, *where*

$$f(x) = a - \frac{1}{x}.$$

Since

$$f'(x) = \frac{1}{x^2},$$

it follows that in Newton's Method, we can obtain the next iterate x_{n+1} *from the previous iterate* x_n *by*

$$x_{n+1} = x_n - \frac{a - 1/x_n}{1/x_n^2} = x_n - \frac{a}{1/x_n^2} + \frac{1/x_n}{1/x_n^2} = 2x_n - ax_n^2.$$

Note that no divisions are necessary to obtain x_{n+1} *from* x_n. *This iteration was actually used on older IBM computers to implement division in hardware.*

We use this iteration to compute the reciprocal of $a = 12$. *Choosing our starting iterate to be* 0.1, *we compute the next several iterates as follows:*

$$\begin{aligned} x_1 &= 2(0.1) - 12(0.1)^2 = 0.08 \\ x_2 &= 2(0.12) - 12(0.12)^2 = 0.0832 \\ x_3 &= 0.0833312 \\ x_4 &= 0.08333333333279 \\ x_5 &= 0.08333333333333. \end{aligned}$$

We conclude that 0.08333333333333 *is an accurate approximation to the correct solution.*

Now, suppose we repeat this process, but with an initial iterate of $x_0 = 1$. *Then, we have*

$$\begin{aligned} x_1 &= 2(1) - 12(1)^2 = -10 \\ x_2 &= 2(-10) - 12(-10)^2 = -1220 \\ x_3 &= 2(-1220) - 12(-1220)^2 = -17863240 \end{aligned}$$

It is clear that this sequence of iterates is not going to converge to the correct solution. In general, for this iteration to converge to the reciprocal of a, *the initial iterate* x_0 *must be chosen so that* $0 < x_0 < 2/a$. *This condition guarantees that the next iterate* x_1 *will at least be positive. The contrast between the two choices of* x_0 *are illustrated in Figure 8.7 for* $a = 8$. □

8.4.2 Convergence Analysis

We now analyze the convergence of Newton's Method applied to the equation $f(x) = 0$, where we assume that f is twice continuously differentiable near the exact solution x^*. As before, we define

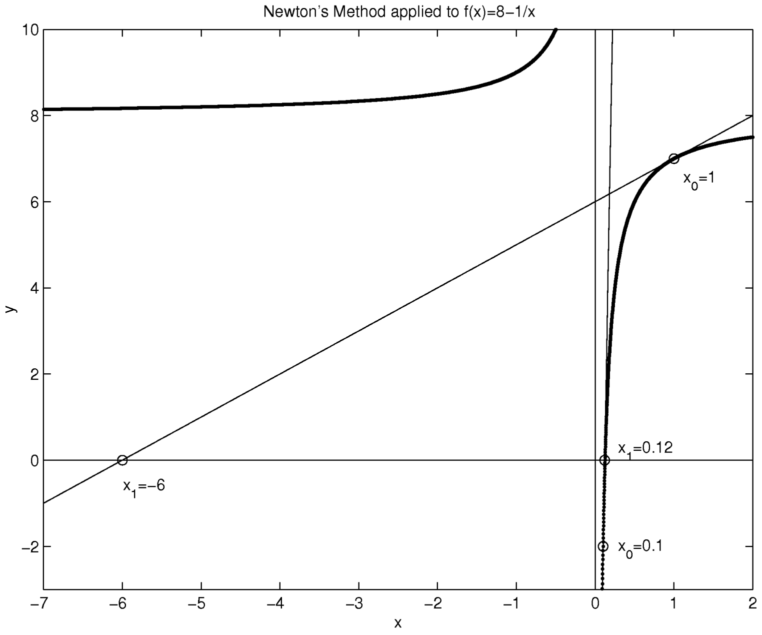


Figure 8.7: Newton's Method used to compute the reciprocal of 8 by solving the equation $f(x) = 8 - 1/x = 0$. When $x_0 = 0.1$, the tangent line of $f(x)$ at $(x_0, f(x_0))$ crosses the x-axis at $x_1 = 0.12$, which is close to the exact solution. When $x_0 = 1$, the tangent line crosses the x-axis at $x_1 = -6$, which causes searching to continue on the wrong portion of the graph, so the sequence of iterates does not converge to the correct solution.

$e_k = x_k - x^*$ to be the error after k iterations. Using a Taylor expansion around x_k, we obtain

$$\begin{aligned} e_{k+1} &= x_{k+1} - x^* \\ &= x_k - \frac{f(x_k)}{f'(x_k)} - x^* \\ &= e_k - \frac{f(x_k)}{f'(x_k)} \\ &= e_k - \frac{1}{f'(x_k)}\left[f(x^*) - f'(x_k)(x^* - x_k) - \frac{1}{2}f''(\xi_k)(x_k - x^*)^2\right] \\ &= \frac{f''(\xi_k)}{2f'(x_k)}e_k^2 \end{aligned}$$

where ξ_k is between x_k and x^*.

Because, for each k, ξ_k lies between x_k and x^*, ξ_k converges to x^* as well. By the continuity of f'', we conclude that Newton's method converges quadratically to x^*, with asymptotic error constant

$$C = \frac{|f''(x^*)|}{2|f'(x^*)|}.$$

Example 8.4.5 *Suppose that Newton's Method is used to find the solution of* $f(x) = 0$*, where*

$f(x) = x^2 - 2$. *We examine the error* $e_k = x_k - x^*$, *where* $x^* = \sqrt{2}$ *is the exact solution. The first two iterations are illustrated in Figure 8.8. Continuing, we obtain*

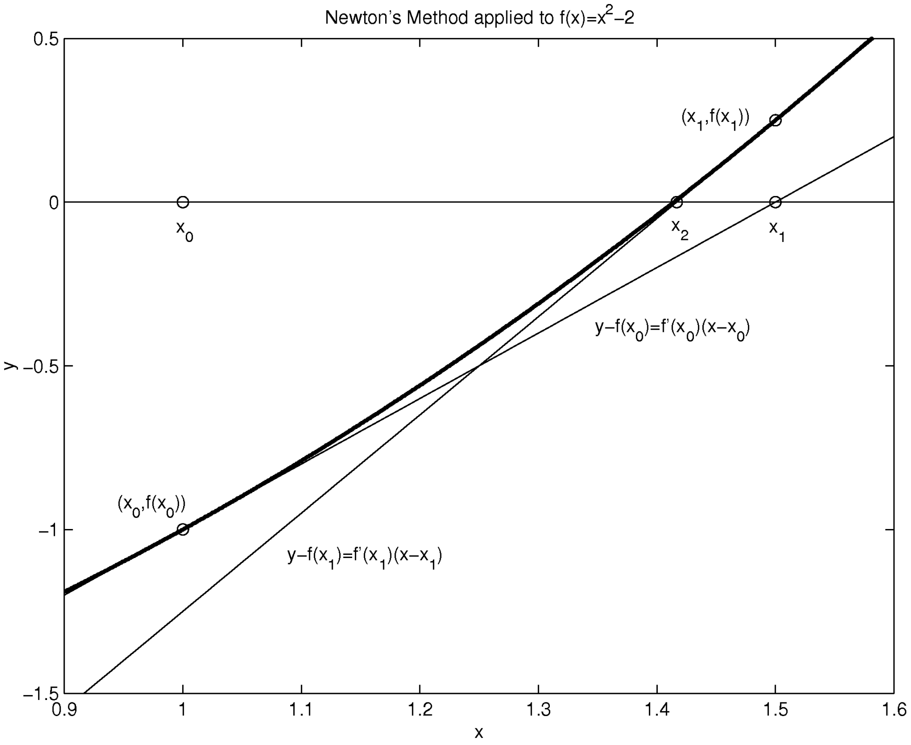


Figure 8.8: Newton's Method applied to $f(x) = x^2 - 2$. The bold curve is the graph of f. The initial iterate x_0 is chosen to be 1. The tangent line of $f(x)$ at the point $(x_0, f(x_0))$ is used to approximate $f(x)$, and it crosses the x-axis at $x_1 = 1.5$, which is much closer to the exact solution than x_0. Then, the tangent line at $(x_1, f(x_1))$ is used to approximate $f(x)$, and it crosses the x-axis at $x_2 = 1.41\bar{6}$, which is already very close to the exact solution.

k	x_k	$\lvert e_k \rvert$
0	*1*	*0.41421356237310*
1	*1.5*	*0.08578643762690*
2	*1.41666666666667*	*0.00245310429357*
3	*1.41421568627457*	*0.00000212390141*
4	*1.41421356237469*	*0.00000000000159*

We can determine analytically that Newton's Method converges quadratically, and in this example, the asymptotic error constant is $|f''(\sqrt{2})/2f'(\sqrt{2})| \approx 0.35355$. *Examining the numbers in the table above, we can see that the number of correct decimal places approximately doubles with each iteration, which is typical of quadratic convergence. Furthermore, we have*

$$\frac{|e_4|}{|e_3|^2} \approx 0.35352,$$

so the actual behavior of the error is consistent with the behavior that is predicted by theory. □

It is easy to see from the above analysis, however, that if $f'(x^*)$ is very small, or zero, then convergence can be very slow, or may not even occur.

Example 8.4.6 *The function*

$$f(x) = (x-1)^2 e^x$$

has a double root at $x^* = 1$*, and therefore* $f'(x^*) = 0$*. Therefore, the previous convergence analysis does not apply. Instead, we obtain*

$$\begin{aligned} e_{k+1} &= x_{k+1} - 1 \\ &= x_k - \frac{f(x_k)}{f'(x_k)} - 1 \\ &= x_k - \frac{(x_k-1)^2 e^{x_k}}{[2(x_k-1) + (x_k-1)^2]e^{x_k}} - 1 \\ &= e_k - \frac{e_k^2}{2e_k + e_k^2} \\ &= \frac{x_k}{x_k+1} e_k. \end{aligned}$$

It follows that if we choose $x_0 > 0$*, then Newton's method converges to* $x^* = 1$ linearly*, with asymptotic error constant* $C = \frac{1}{2}$*.* □

Exercise 8.4.5 *Prove that if* $f(x^*) = 0$*,* $f'(x^*) = 0$*, and* $f''(x^*) \neq 0$*, then Newton's method converges linearly with asymptotic error constant* $C = \frac{1}{2}$*. That is, the result obtained in Example 8.4.6 applies in general.*

Normally, convergence of Newton's method is only assured if x_0 is chosen sufficiently close to x^*. However, in some cases, it is possible to prove that Newton's method converges quadratically on an interval, under certain conditions on the sign of the derivatives of f on that interval. For example, suppose that on the interval $I_\delta = [x^*, x^* + \delta]$, $f'(x) > 0$ and $f''(x) > 0$, so that f is increasing and concave up on this interval.

Let $x_k \in I_\delta$. Then, from

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},$$

we have $x_{k+1} < x_k$, because f, being equal to zero at x^* and increasing on I_δ, must be positive at x_k. However, because

$$x_{k+1} - x^* = \frac{f''(\xi_k)}{2f'(x_k)}(x_k - x^*)^2,$$

and f' and f'' are both positive at x_k, we must also have $x_{k+1} > x^*$.

It follows that the sequence $\{x_k\}$ is monotonic and bounded, and therefore must be convergent to a limit $x_* \in I_\delta$. From the convergence of the sequence and the determination of x_{k+1} from x_k, it follows that $f(x_*) = 0$. However, f is positive on $(x^*, x^* + \delta]$, which means that we must have

$x_* = x^*$, so Newton's method converges to x^*. Using the previous analysis, it can be shown that this convergence is quadratic.

Exercise 8.4.6 *Prove that Newton's method converges quadratically if* $f'(x) > 0$ *and* $f''(x) < 0$ *on an interval* $[a, b]$ *that contains* x^* *and* x_0*, where* $x_0 < x^*$*. What goes wrong if* $x_0 > x^*$*?*

8.4.3 The Secant Method

One drawback of Newton's method is that it is necessary to evaluate $f'(x)$ at various points, which may not be practical for some choices of f. The *secant method* avoids this issue by using a finite difference to approximate the derivative. As a result, $f(x)$ is approximated by a *secant line* through two points on the graph of f, rather than a tangent line through one point on the graph.

Since a secant line is defined using two points on the graph of $f(x)$, as opposed to a tangent line that requires information at only one point on the graph, it is necessary to choose two initial iterates x_0 and x_1. Then, as in Newton's method, the next iterate x_2 is then obtained by computing the x-value at which the secant line passing through the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$ has a y-coordinate of zero. This yields the equation

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_1) + f(x_1) = 0$$

which has the solution

$$x_2 = x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)}.$$

This leads to the following algorithm.

Algorithm 8.4.7 (Secant Method) *Let* $f : \mathbb{R} \to \mathbb{R}$ *be a continuous function. The following algorithm computes an approximate solution* x^* *to the equation* $f(x) = 0$.

```
Choose two initial guesses $x_0$ and $x_1$
for $k = 1, 2, 3, \ldots$ do
    if $f(x_k)$ is sufficiently small then
        $x^* = x_k$
        return $x^*$
    end
    $x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$
    if $|x_{k+1} - x_k|$ is sufficiently small then
        $x^* = x_{k+1}$
        return $x^*$
    end
end
```

Exercise 8.4.7 *Write a* MATLAB *function* `[x,niter]=secant(f,x0,x1,tol)` *that implements Algorithm 8.4.7 to solve the equation* $f(x) = 0$ *using the Secant Method with initial guesses* x_0 *and* x_1*, and absolute error tolerance* `tol`*. The output arguments* `x` *and* `niter` *are the computed solution* x^* *and number of iterations, respectively.*

Like Newton's method, it is necessary to choose the starting iterate x_0 to be reasonably close to the solution x^*. Convergence is not as rapid as that of Newton's Method, since the secant-line approximation of f is not as accurate as the tangent-line approximation employed by Newton's method.

Example 8.4.8 *We will use the Secant Method to solve the equation $f(x) = 0$, where $f(x) = x^2 - 2$. This method requires that we choose two initial iterates x_0 and x_1, and then compute subsequent iterates using the formula*

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}, \quad n = 1, 2, 3, \ldots .$$

We choose $x_0 = 1$ and $x_1 = 1.5$. Applying the above formula, we obtain

$$\begin{aligned} x_2 &= 1.4 \\ x_3 &= 1.413793103448276 \\ x_4 &= 1.414215686274510 \\ x_5 &= 1.414213562057320. \end{aligned}$$

As we can see, the iterates produced by the Secant Method are converging to the exact solution $x^* = \sqrt{2}$, but not as rapidly as those produced by Newton's Method. □

Exercise 8.4.8 *How many correct decimal places are obtained in each x_k in the preceding example? What does this suggest about the order of convergence of the Secant method?*

We now prove that the Secant Method converges if x_0 is chosen sufficiently close to a solution x^* of $f(x) = 0$, if f is continuously differentiable near x^* and $f'(x^*) = \alpha \neq 0$. Without loss of generality, we assume $\alpha > 0$. Then, by the continuity of f', there exists an interval $I_\delta = [x^* - \delta, x^* + \delta]$ such that

$$\frac{3\alpha}{4} \leq f'(x) \leq \frac{5\alpha}{4}, \quad x \in I_\delta.$$

It follows from the Mean Value Theorem that

$$\begin{aligned} x_{k+1} - x^* &= x_k - x^* - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \\ &= x_k - x^* - \frac{f'(\theta_k)(x_k - x^*)}{f'(\varphi_k)} \\ &= \left[1 - \frac{f'(\theta_k)}{f'(\varphi_k)}\right](x_k - x^*), \end{aligned}$$

where θ_k lies between x_k and x^*, and φ_k lies between x_k and x_{k-1}. Therefore, if x_{k-1} and x_k are in I_δ, then so are φ_k and θ_k, and x_{k+1} satisfies

$$|x_{k+1} - x^*| \leq \max\left\{\left|1 - \frac{5\alpha/4}{3\alpha/4}\right|, \left|1 - \frac{3\alpha/4}{5\alpha/4}\right|\right\}|x_k - x^*| \leq \frac{2}{3}|x_k - x^*|.$$

We conclude that if $x_0, x_1 \in I_\delta$, then all subsequent iterates lie in I_δ, and the Secant Method converges at least linearly, with asymptotic rate constant $2/3$.

The order of convergence of the Secant Method can be determined using a result, which we will not prove here, stating that if $\{x_k\}_{k=0}^{\infty}$ is the sequence of iterates produced by the Secant Method for solving $f(x) = 0$, and if this sequence converges to a solution x^*, then for k sufficiently large,

$$|x_{k+1} - x^*| \approx S|x_k - x^*||x_{k-1} - x^*|$$

for some constant S.

We assume that $\{x_k\}$ converges to x^* of order α. Then, dividing both sides of the above relation by $|x_k - x^*|^\alpha$, we obtain

$$\frac{|x_{k+1} - x^*|}{|x_k - x^*|^\alpha} \approx S|x_k - x^*|^{1-\alpha}|x_{k-1} - x^*|.$$

Because α is the order of convergence, the left side must converge to a positive constant C as $k \to \infty$. It follows that the right side must converge to a positive constant as well, as must its reciprocal. In other words, there must exist positive constants C_1 and C_2

$$\frac{|x_k - x^*|}{|x_{k-1} - x^*|^\alpha} \to C_1, \quad \frac{|x_k - x^*|^{\alpha-1}}{|x_{k-1} - x^*|} \to C_2.$$

This can only be the case if there exists a nonzero constant β such that

$$\frac{|x_k - x^*|}{|x_{k-1} - x^*|^\alpha} = \left(\frac{|x_k - x^*|^{\alpha-1}}{|x_{k-1} - x^*|}\right)^\beta,$$

which implies that

$$1 = (\alpha - 1)\beta \quad \text{and} \quad \alpha = \beta.$$

Eliminating β, we obtain the equation

$$\alpha^2 - \alpha - 1 = 0,$$

which has the solutions

$$\alpha_1 = \frac{1 + \sqrt{5}}{2} \approx 1.618, \quad \alpha_2 = \frac{1 - \sqrt{5}}{2} \approx -0.618.$$

Since we must have $\alpha \geq 1$, the order of convergence is 1.618.

Exercise 8.4.9 *What is the value of S in the preceding discussion? That is, compute*

$$S = \lim_{x_{k-1}, x_k \to x^*} \frac{x_{k+1} - x^*}{(x_k - x^*)(x_{k-1} - x^*)}.$$

Hint: *Take one limit at a time, and use Taylor expansion. Assume that x^* is not a double root.*

Exercise 8.4.10 *Use the value of S from the preceding exercise to obtain the asymptotic error constant for the Secant method.*

Exercise 8.4.11 *Use both Newton's method and the Secant method to compute a root of the same polynomial. For both methods, count the number of floating-point operations required in each iteration, and the number of iterations required to achieve convergence with the same error tolerance. Which method requires fewer floating-point operations?*

8.5 Convergence Acceleration

Suppose that a sequence $\{x_k\}_{k=0}^\infty$ converges linearly to a limit x^*, in such a way that if k is sufficiently large, then $x_k - x^*$ has the same sign; that is, $\{x_k\}$ converges *monotonically* to x^*. It follows from the linear convergence of $\{x_k\}$ that for sufficiently large k,

$$\frac{x_{k+2} - x^*}{x_{k+1} - x^*} \approx \frac{x_{k+1} - x^*}{x_k - x^*}. \tag{8.2}$$

Solving for x^* yields

$$x^* \approx x_k - \frac{(x_{k+1} - x_k)^2}{x_{k+2} - 2x_{k+1} + x_k}. \tag{8.3}$$

Exercise 8.5.1 *Solve (8.2) for x^* to obtain (8.3).*

Therefore, we can construct an alternative sequence $\{\hat{x}_k\}_{k=0}^\infty$, where

$$\hat{x}_k = x_k - \frac{(x_{k+1} - x_k)^2}{x_{k+2} - 2x_{k+1} + x_k},$$

that also converges to x^*. This sequence has the following desirable property.

Theorem 8.5.1 *Suppose that the sequence $\{x_k\}_{k=0}^\infty$ converges linearly to a limit x^* and that for k sufficiently large, $(x_{k+1} - x^*)(x_k - x^*) > 0$. Then, if the sequence $\{\hat{x}_k\}_{k=0}^\infty$ is defined by*

$$\hat{x}_k = x_k - \frac{(x_{k+1} - x_k)^2}{x_{k+2} - 2x_{k+1} + x_k}, \quad k = 0, 1, 2, \ldots,$$

then

$$\lim_{k\to\infty} \frac{\hat{x}_k - x^*}{x_k - x^*} = 0.$$

In other words, the sequence $\{\hat{x}_k\}$ converges to x^* more rapidly than $\{x_k\}$ does.

Exercise 8.5.2 *Prove Theorem 8.5.1. Assume that the sequence $\{x_k\}_{k=0}^\infty$ converges linearly with asymptotic error constant C, where $0 < C < 1$.*

If we define the *forward difference operator* Δ by

$$\Delta x_k = x_{k+1} - x_k,$$

then

$$\Delta^2 x_k = \Delta(x_{k+1} - x_k) = (x_{k+2} - x_{k+1}) - (x_{k+1} - x_k) = x_{k+2} - 2x_{k+1} + x_k,$$

and therefore $\hat{x}_k$ can be rewritten as

$$\hat{x}_k = x_k - \frac{(\Delta x_k)^2}{\Delta^2 x_k}, \quad k = 0, 1, 2, \ldots$$

For this reason, the method of accelerating the convergence of $\{x_k\}$ by constructing $\{\hat{x}_k\}$ is called *Aitken's Δ^2 Method* [2].

A slight variation of this method, called *Steffensen's Method*, can be used to accelerate the convergence of Fixed-point Iteration, which, as previously discussed, is linearly convergent. The basic idea is as follows:

1. Choose an initial iterate x_0
2. Compute x_1 and x_2 using Fixed-point Iteration
3. Use Aitken's Δ^2 Method to compute $\hat{x}_0$ from x_0, x_1, and x_2
4. Repeat steps 2 and 3 with $x_0 = \hat{x}_0$.

The principle behind Steffensen's Method is that $\hat{x}_0$ is thought to be a better approximation to the fixed point x^* than x_2, so it should be used as the next iterate for Fixed-point Iteration.

Example 8.5.2 *We wish to find the unique fixed point of the function $f(x) = \cos x$ on the interval $[0, 1]$. If we use Fixed-point Iteration with $x_0 = 0.5$, then we obtain the following iterates from the formula $x_{k+1} = g(x_k) = \cos(x_k)$. All iterates are rounded to five decimal places.*

$$\begin{aligned} x_1 &= 0.87758 \\ x_2 &= 0.63901 \\ x_3 &= 0.80269 \\ x_4 &= 0.69478 \\ x_5 &= 0.76820 \\ x_6 &= 0.71917. \end{aligned}$$

These iterates show little sign of converging, as they are oscillating around the fixed point.

If, instead, we use Fixed-point Iteration with acceleration by Aitken's Δ^2 method, we obtain a new sequence of iterates $\{\hat{x}_k\}$, where

$$\begin{aligned} \hat{x}_k &= x_k - \frac{(\Delta x_k)^2}{\Delta^2 x_k} \\ &= x_k - \frac{(x_{k+1} - x_k)^2}{x_{k+2} - 2x_{k+1} + x_k}, \end{aligned}$$

for $k = 0, 1, 2, \ldots$. The first few iterates of this sequence are

$$\begin{aligned} \hat{x}_0 &= 0.73139 \\ \hat{x}_1 &= 0.73609 \\ \hat{x}_2 &= 0.73765 \\ \hat{x}_3 &= 0.73847 \\ \hat{x}_4 &= 0.73880. \end{aligned}$$

Clearly, these iterates are converging much more rapidly than Fixed-point Iteration, as they are not oscillating around the fixed point, but convergence is still linear.

Finally, we try Steffensen's Method. We begin with the first three iterates of Fixed-point Iteration,

$$x_0^{(0)} = x_0 = 0.5, \quad x_1^{(0)} = x_1 = 0.87758, \quad x_2^{(0)} = x_2 = 0.63901.$$

Then, we use the formula from Aitken's Δ^2 *Method to compute*

$$x_0^{(1)} = x_0^{(0)} - \frac{(x_1^{(0)} - x_0^{(0)})^2}{x_2^{(0)} - 2x_1^{(0)} + x_0^{(0)}} = 0.73139.$$

We use this value to restart Fixed-point Iteration and compute two iterates, which are

$$x_1^{(1)} = \cos(x_0^{(1)}) = 0.74425, \quad x_2^{(1)} = \cos(x_1^{(1)}) = 0.73560.$$

Repeating this process, we apply the formula from Aitken's Δ^2 *Method to the iterates* $x_0^{(1)}$, $x_1^{(1)}$ *and* $x_2^{(1)}$ *to obtain*

$$x_0^{(2)} = x_0^{(1)} - \frac{(x_1^{(1)} - x_0^{(1)})^2}{x_2^{(1)} - 2x_1^{(1)} + x_0^{(1)}} = 0.739076.$$

Restarting Fixed-point Iteration with $x_0^{(2)}$ *as the initial iterate, we obtain*

$$x_1^{(2)} = \cos(x_0^{(2)}) = 0.739091, \quad x_2^{(2)} = \cos(x_1^{(2)}) = 0.739081.$$

The most recent iterate $x_2^{(2)}$ *is correct to five decimal places.*

Using all three methods to compute the fixed point to ten decimal digits of accuracy, we find that Fixed-point Iteration requires 57 iterations, so x_{57} *must be computed. Aitken's* Δ^2 *Method requires us to compute 25 iterates of the modified sequence* $\{\hat{x}_k\}$*, which in turn requires 27 iterates of the sequence* $\{x_k\}$*, where the first iterate* x_0 *is given. Steffensen's Method requires us to compute* $x_2^{(3)}$*, which means that only 11 iterates need to be computed, 8 of which require a function evaluation.* □

Exercise 8.5.3 *Write a* MATLAB *function* `[x,niter]=steffensen(g,x0,tol)` *that modifies your function* `fixedpt` *from Exercise 8.3.5 to accelerate convergence using Steffensen's Method. Test your function on the equation from Example 8.3.2. How much more rapidly do the iterates converge?*

8.6 Systems of Nonlinear Equations

The techniques that have been presented in this chapter for solving a single nonlinear equation of the form $f(x) = 0$, or $x = g(x)$, can be generalized to solve a *system* of n nonlinear equations in n unknowns $x_1, x_2, \ldots, x_n$, using concepts and techniques from numerical linear algebra.

8.6.1 Fixed-Point Iteration

Previously, we have learned how to use *fixed-point iteration* to solve a single nonlinear equation of the form

$$f(x) = 0$$

by first transforming the equation into one of the form

$$x = g(x).$$

Then, after choosing an initial guess $x^{(0)}$, we compute a sequence of iterates by

$$x^{(k+1)} = g(x^{(k)}), \quad k = 0, 1, 2, \ldots,$$

that, hopefully, converges to a solution of the original equation.

We have also learned that if the function g is a continuous function that maps an interval I into itself, then g has a *fixed point* (also called a *stationary point*) x^* in I, which is a point that satisfies $x^* = g(x^*)$. That is, a solution to $f(x) = 0$ exists within I. Furthermore, if there is a constant $\rho < 1$ such that

$$|g'(x)| < \rho, \quad x \in I,$$

then this fixed point is *unique*.

It is worth noting that the constant ρ, which can be used to indicate the speed of convergence of fixed-point iteration, corresponds to the spectral radius $\rho(T)$ of the iteration matrix $T = M^{-1}N$ used in a stationary iterative method of the form

$$\mathbf{x}^{(k+1)} = T\mathbf{x}^{(k)} + M^{-1}\mathbf{b}$$

for solving $A\mathbf{x} = \mathbf{b}$, where $A = M - N$.

We now generalize fixed-point iteration to the problem of solving a *system* of n nonlinear equations in n unknowns,

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0, \\ f_2(x_1, x_2, \ldots, x_n) &= 0, \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0. \end{aligned}$$

For simplicity, we express this system of equations in vector form,

$$\mathbf{F}(\mathbf{x}) = \mathbf{0},$$

where $\mathbf{F} : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ is a *vector-valued* function of n variables represented by the vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, and $f_1, f_2, \ldots, f_n$ are the *component functions*, or *coordinate functions*, of $\mathbf{F}$ (see A.1.2).

The notions of limits and continuity for functions of several variables We now define fixed-point iteration for solving a system of nonlinear equations

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}.$$

First, we transform this system of equations into an equivalent system of the form

$$\mathbf{x} = \mathbf{G}(\mathbf{x}).$$

One approach to doing this is to solve the ith equation in the original system for x_i. This is analogous to the derivation of the Jacobi method for solving systems of linear equations. Next, we choose an initial guess $\mathbf{x}^{(0)}$. Then, we compute subsequent iterates by

$$\mathbf{x}^{(k+1)} = \mathbf{G}(\mathbf{x}^{(k)}), \quad k = 0, 1, 2, \ldots.$$

Exercise 8.6.1 *Write a* MATLAB *function* `x=fixedptsys(G,x0,tol)` *that performs fixed-point iteration to solve the equation* $\mathbf{x} = \mathbf{G}(\mathbf{x})$ *with an initial guess* `x0` *and absolute error tolerance* `tol`.

The existence and uniqueness of fixed points of vector-valued functions of several variables can be described in an analogous manner to how it is described in the single-variable case. The function $\mathbf{G}$ has a fixed point in a domain $D \subseteq \mathbb{R}^n$ if $\mathbf{G}$ is continuous on D and $\mathbf{G}$ maps D into D. Furthermore, if $\mathbf{G}$ has continuous first partial derivatives and there exists a constant $\rho < 1$ such that, in some natural matrix norm,

$$\|J_{\mathbf{G}}(\mathbf{x})\| \leq \rho, \quad \mathbf{x} \in D,$$

where $J_{\mathbf{G}}(\mathbf{x})$ is the *Jacobian matrix* of first partial derivatives of $\mathbf{G}$ evaluated at $\mathbf{x}$, then $\mathbf{G}$ has a *unique* fixed point $\mathbf{x}^*$ in D, and fixed-point iteration is guaranteed to converge to $\mathbf{x}^*$ for any initial guess chosen in D. This can be seen by computing a multivariable Taylor expansion of the error $\mathbf{x}^{(k+1)} - \mathbf{x}^*$ around $\mathbf{x}^*$.

Exercise 8.6.2 *Use a multivariable Taylor expansion to prove that if* $\mathbf{G}$ *satisfies the assumptions in the preceding discussion (that it is continuous, maps* D *into itself, has continuous first partial derivatives, and satisfies* $\|J_{\mathbf{G}}(\mathbf{x})\| \leq \rho < 1$ *for* $\mathbf{x} \in D$ *and any natural matrix norm* $\|\cdot\|$*), then* $\mathbf{G}$ *has a unique fixed point* $\mathbf{x}^* \in D$ *and fixed-point iteration will converge to* $\mathbf{x}^*$ *for any initial guess* $\mathbf{x}^{(0)} \in D$.

Exercise 8.6.3 *Under the assumptions of Exercise 8.6.2 to obtain a bound for the error after* k *iterations,* $\|\mathbf{x}^{(k)} - \mathbf{x}^*\|$*, in terms of the initial difference* $\|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\|$.

The constant ρ measures the rate of convergence of fixed-point iteration, as the error approximately decreases by a factor of ρ at each iteration. It is interesting to note that the convergence of fixed-point iteration for functions of several variables can be accelerated by using an approach similar to how the Jacobi method for linear systems is modified to obtain the Gauss-Seidel method. That is, when computing $\mathbf{x}_i^{(k+1)}$ by evaluating $g_i(\mathbf{x}^{(k)})$, we replace $\mathbf{x}_j^{(k)}$, for $j < i$, by $\mathbf{x}_j^{(k+1)}$, since it has already been computed (assuming all components of $\mathbf{x}^{(k+1)}$ are computed in order). Therefore, as in Gauss-Seidel, we are using the most up-to-date information available when computing each iterate.

Example 8.6.1 *Consider the system of equations*

$$\begin{aligned} x_2 &= x_1^2, \\ x_1^2 + x_2^2 &= 1. \end{aligned}$$

The first equation describes a parabola, while the second describes the unit circle. By graphing both equations, it can easily be seen that this system has two solutions, one of which lies in the first quadrant $(x_1, x_2 > 0)$.

To solve this system using fixed-point iteration, we solve the second equation for x_1*, and obtain the equivalent system*

$$\begin{aligned} x_1 &= \sqrt{1 - x_2^2}, \\ x_2 &= x_1^2. \end{aligned}$$

If we consider the rectangle

$$D = \{(x_1, x_2)\,|\,0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1\},$$

we see that the function

$$\mathbf{G}(x_1, x_2) = \left(\sqrt{1 - x_2^2}, x_1^2\right)$$

maps D *into itself. Because* $\mathbf{G}$ *is also continuous on* D*, it follows that* $\mathbf{G}$ *has a fixed point in* D*. However,* $\mathbf{G}$ *has the Jacobian matrix*

$$J_{\mathbf{G}}(\mathbf{x}) = \left[\begin{array}{cc} 0 & -x_2/\sqrt{1 - x_2^2} \\ 2x_1 & 0 \end{array}\right],$$

which cannot satisfy $\|J_{\mathbf{G}}\| < 1$ *on* D*. Therefore, we cannot guarantee that fixed-point iteration with this choice of* $\mathbf{G}$ *will converge, and, in fact, it can be shown that it does* not *converge. Instead, the iterates tend to approach the corners of* D*, at which they remain.*

In an attempt to achieve convergence, we note that $\partial g_2/\partial x_1 = 2x_1 > 1$ *near the fixed point. Therefore, we modify* $\mathbf{G}$ *as follows:*

$$\mathbf{G}(x_1, x_2) = \left(\sqrt{x_2}, \sqrt{1 - x_1^2}\right).$$

For this choice of $\mathbf{G}$*,* $J_{\mathbf{G}}$ *still has partial derivatives that are greater than 1 in magnitude near the fixed point. However, there is one crucial distinction: near the fixed point,* $\rho(J_{\mathbf{G}}) < 1$*, whereas with the original choice of* $\mathbf{G}$*,* $\rho(J_{\mathbf{G}}) > 1$*. Attempting fixed-point iteration with the new* $\mathbf{G}$*, we see that convergence is actually achieved, although it is slow.* □

It can be seen from this example that the conditions for the existence and uniqueness of a fixed point are sufficient, but not necessary.

8.6.2 Newton's Method

Suppose that fixed-point iteration is being used to solve an equation of the form $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, where $\mathbf{F}$ is a vector-valued function of n variables, by transforming it into an equation of the form $\mathbf{x} = \mathbf{G}(\mathbf{x})$. Furthermore, suppose that $\mathbf{G}$ is known to map a domain $D \subseteq \mathbb{R}^n$ into itself, so that a fixed point exists in D. We have learned that the number ρ, where

$$\|J_{\mathbf{G}}(\mathbf{x})\| \leq \rho, \quad \mathbf{x} \in D,$$

provides an indication of the rate of convergence, in the sense that as the iterates converge to a fixed point $\mathbf{x}^*$, if they converge, the error satisfies

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq \rho\|\mathbf{x}^{(k)} - \mathbf{x}^*\|.$$

Furthermore, as the iterates converge, a suitable value for ρ is given by $\rho(J_{\mathbf{G}}(\mathbf{x}^*))$, the spectral radius of the Jacobian matrix at the fixed point.

Therefore, it makes sense to ask: what if this spectral radius is equal to zero? In that case, if the first partial derivatives are continuous near $\mathbf{x}^*$, and the second partial derivatives are continuous and bounded at $\mathbf{x}^*$, then fixed-point iteration converges *quadratically.* That is, there exists a constant M such that

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq M\|\mathbf{x}^{(k)} - \mathbf{x}^*\|^2.$$

Exercise 8.6.4 *Let $\mathbf{G} : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be a twice continuously differentiable function that has a fixed point $\mathbf{x}^{(0)} \in D$. Furthermore, assume that $J_{\mathbf{G}}(\mathbf{x}^*) = 0$ and that the second partial derivatives of the component functions g_i, $i = 1, 2, \ldots, n$ of $\mathbf{G}$ are bounded on D. Use a multivariable Taylor expansion to prove that fixed-point iteration applied to $\mathbf{G}$ converges quadratically for any initial guess $\mathbf{x}^{(0)} \in D$.*

We have previously learned that for a single nonlinear equation $f(x) = 0$, *Newton's method* generally achieves quadratic convergence. Recall that this method computes iterates by

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}, \quad k = 0, 1, 2, \ldots,$$

where $x^{(0)}$ is an initial guess. We now wish to generalize this method to systems of nonlinear equations.

Consider the fixed-point iteration function

$$\mathbf{G}(\mathbf{x}) = \mathbf{x} - [J_{\mathbf{F}}(\mathbf{x})]^{-1}\mathbf{F}(\mathbf{x}). \tag{8.4}$$

Then, it can be shown by direct differentiation that the Jacobian matrix of this function is equal to the zero matrix at $\mathbf{x} = \mathbf{x}^*$, a solution to $\mathbf{F}(\mathbf{x}) = \mathbf{0}$. If we define

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix}, \quad \mathbf{G}(\mathbf{x}) = \begin{bmatrix} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ g_n(x_1, x_2, \ldots, x_n) \end{bmatrix},$$

where f_i and g_i, $i = 1, 2, \ldots, n$ are the coordinate functions of $\mathbf{F}$ and $\mathbf{G}$, respectively, then we have

$$\frac{\partial}{\partial x_k} g_i(\mathbf{x}) = \frac{\partial}{\partial x_k}\left[x_i - \sum_{j=1}^n b_{ij}(\mathbf{x}) f_j(\mathbf{x})\right] = \delta_{ik} - \sum_{j=1}^n b_{ij}(\mathbf{x}) \frac{\partial}{\partial x_k} f_j(\mathbf{x}) - \sum_{j=1}^n \frac{\partial}{\partial x_k} b_{ij}(\mathbf{x}) f_j(\mathbf{x}),$$

where $b_{ij}(\mathbf{x})$ is the (i, j) element of $[J_{\mathbf{F}}(\mathbf{x})]^{-1}$.

Exercise 8.6.5 *Prove that if $\mathbf{G}(\mathbf{x})$ is defined as in (8.4) and $\mathbf{F}(\mathbf{x}^*) = \mathbf{0}$, then $J_{\mathbf{G}}(\mathbf{x}^*) = 0$.*

We see that this choice of fixed-point iteration is a direct generalization of Newton's method to systems of equations, in which the division by $f'(\mathbf{x}^{(k)})$ is replaced by multiplication by the inverse of $J_{\mathbf{F}}(\mathbf{x}^{(k)})$, the total derivative of $\mathbf{F}(\mathbf{x})$.

In summary, Newton's method proceeds as follows: first, we choose an initial guess $\mathbf{x}^{(0)}$. Then, for $k = 0, 1, 2, \ldots$, we iterate as follows:

$$\begin{aligned} \mathbf{y}_k &= -\mathbf{F}(\mathbf{x}^{(k)}) \\ \mathbf{s}_k &= [J_{\mathbf{F}}(\mathbf{x}^{(k)})]^{-1}\mathbf{y}_k \\ \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \mathbf{s}_k \end{aligned}$$

The vector $\mathbf{s}_k$ is computed by solving the system

$$J_{\mathbf{F}}(\mathbf{x}^{(k)})\mathbf{s}_k = \mathbf{y}_k,$$

using a method such as Gaussian elimination with back substitution.

Example 8.6.2 *Recall the system of equations*

$$\begin{aligned} x_2 - x_1^2 &= 0, \\ x_1^2 + x_2^2 - 1 &= 0. \end{aligned}$$

Fixed-point iteration converged rather slowly for this system, if it converged at all. Now, we apply Newton's method to this system. We have

$$\mathbf{F}(x_1, x_2) = \begin{bmatrix} x_2 - x_1^2 \\ x_1^2 + x_2^2 - 1 \end{bmatrix}, \quad J_{\mathbf{F}}(x_1, x_2) = \begin{bmatrix} -2x_1 & 1 \\ 2x_1 & 2x_2 \end{bmatrix}.$$

Using the formula for the inverse of a 2×2 *matrix, we obtain the iteration*

$$\begin{bmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \end{bmatrix} = \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \end{bmatrix} + \frac{1}{4x_1^{(k)}x_2^{(k)} + 2x_1^{(k)}} \begin{bmatrix} 2x_2^{(k)} & -1 \\ -2x_1^{(k)} & -2x_1^{(k)} \end{bmatrix} \begin{bmatrix} x_2^{(k)} - (x_1^{(k)})^2 \\ (x_1^{(k)})^2 + (x_2^{(k)})^2 - 1 \end{bmatrix}.$$

Implementing this iteration in MATLAB, *we see that it converges quite rapidly, much more so than fixed-point iteration. Note that in order for the iteration to not break down, we must have* $\mathbf{x}_1^{(k)} \neq 0$ *and* $\mathbf{x}_2^{(k)} \neq -1/2$. □

Exercise 8.6.6 *Write a* MATLAB *function* `x=newtonsys(F,JF,x0,tol)` *that solves the system of equations* $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ *using Newton's method, where* `JF` *is a function handle that takes* `x` *as input and returns the matrix* $J_{\mathbf{F}}(\mathbf{x})$. *The parameters* `x0` *and* `tol` *are the initial guess and absolute error tolerance, respectively. Test your function on the system from Example 8.6.2.*

One of the drawbacks of Newton's method for systems of nonlinear equations is the need to compute the Jacobian matrix during every iteration, and then solve a system of linear equations, which can be quite expensive. Furthermore, it is not possible to take advantage of information from one iteration to use in the next, in order to save computational effort. These difficulties will be addressed shortly.

8.6.3 Broyden's Method

One of the drawbacks of using Newton's Method to solve a system of nonlinear equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is the computational expense that must be incurred during each iteration to evaluate the partial derivatives of $\mathbf{F}$ at $\mathbf{x}^{(k)}$, and then solve a system of linear equations involving the resulting Jacobian matrix. The algorithm does not facilitate the re-use of data from previous iterations, and in some cases evaluation of the partial derivatives can be unnecessarily costly.

An alternative is to modify Newton's Method so that *approximate* partial derivatives are used, as in the Secant Method for a single nonlinear equation, since the slightly slower convergence is offset by the improved efficiency of each iteration. However, simply replacing the analytical Jacobian matrix of $\mathbf{F}$ with a matrix consisting of finite difference approximations of the partial derivatives does not do much to reduce the cost of each iteration, because the cost of solving the system of linear equations is unchanged.

However, because the Jacobian matrix consists of the partial derivatives evaluated at an element of a convergent sequence, intuitively Jacobian matrices from consecutive iterations are "near" one another in some sense, which suggests that it should be possible to cheaply update an approximate Jacobian matrix from iteration to iteration, in such a way that the *inverse* of the Jacobian matrix can be updated efficiently as well.

This is the case when a matrix has the form

$$B = A + \mathbf{u}\mathbf{v}^T,$$

where $\mathbf{u}$ and $\mathbf{v}$ are given vectors. This modification of A to obtain B is called a *rank-one update*, since $\mathbf{u}\mathbf{v}^T$, an *outer product*, has rank one, since every vector in the range of $\mathbf{u}\mathbf{v}^T$ is a scalar multiple of $\mathbf{u}$. To obtain B^{-1} from A^{-1}, we note that if

$$A\mathbf{x} = \mathbf{u},$$

then

$$B\mathbf{x} = (A + \mathbf{u}\mathbf{v}^T)\mathbf{x} = (1 + \mathbf{v}^T\mathbf{x})\mathbf{u},$$

which yields

$$B^{-1}\mathbf{u} = \frac{1}{1 + \mathbf{v}^T A^{-1}\mathbf{u}} A^{-1}\mathbf{u}.$$

On the other hand, if $\mathbf{x}$ is such that $\mathbf{v}^T A^{-1}\mathbf{x} = 0$, then

$$BA^{-1}\mathbf{x} = (A + \mathbf{u}\mathbf{v}^T)A^{-1}\mathbf{x} = \mathbf{x},$$

which yields

$$B^{-1}\mathbf{x} = A^{-1}\mathbf{x}.$$

This takes us to the following more general problem: given a matrix C, we wish to construct a matrix D such that the following conditions are satisfied:

- $D\mathbf{w} = \mathbf{z}$, for given vectors $\mathbf{w}$ and $\mathbf{z}$
- $D\mathbf{y} = C\mathbf{y}$, if $\mathbf{y}$ is orthogonal to a given vector $\mathbf{g}$.

In our application, $C = A^{-1}$, $D = B^{-1}$, $\mathbf{w} = \mathbf{u}$, $\mathbf{z} = 1/(1 + \mathbf{v}^T A^{-1}\mathbf{u})A^{-1}\mathbf{u}$, and $\mathbf{g} = A^{-T}\mathbf{v}$. To solve this problem, we set

$$D = C + \frac{(\mathbf{z} - C\mathbf{w})\mathbf{g}^T}{\mathbf{g}^T\mathbf{w}}. \tag{8.5}$$

It can be verified directly that D satisfies the above conditions.

Exercise 8.6.7 *Prove that the matrix D defined in (8.5) satisfies $D\mathbf{w} = \mathbf{z}$ and $D\mathbf{y} = C\mathbf{y}$ for $\mathbf{g}^T\mathbf{y} = 0$.*

Applying this definition of D, we obtain

$$\begin{aligned} B^{-1} &= A^{-1} + \frac{\left(\frac{1}{1+\mathbf{v}^T A^{-1}\mathbf{u}} A^{-1}\mathbf{u} - A^{-1}\mathbf{u}\right)\mathbf{v}^T A^{-1}}{\mathbf{v}^T A^{-1}\mathbf{u}} \\ &= A^{-1} - \frac{A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}}{1+\mathbf{v}^T A^{-1}\mathbf{u}}. \end{aligned} \tag{8.6}$$

This formula for the inverse of a rank-one update is known as the *Sherman-Morrison Formula*.

Exercise 8.6.8 *Prove the final form of the Sherman-Morrison formula given in (8.6).*

We now return to the problem of approximating the Jacobian of $\mathbf{F}$, and efficiently obtaining its inverse, at each iterate $\mathbf{x}^{(k)}$. We begin with an exact Jacobian, $A_0 = J_{\mathbf{F}}(\mathbf{x}^{(0)})$, and use A_0 to compute the first iterate, $\mathbf{x}^{(1)}$, using Newton's Method. Then, we recall that for the Secant Method, we use the approximation

$$f'(x_1) \approx \frac{f(x_1) - f(x_0)}{x_1 - x_0}.$$

Generalizing this approach to a system of equations, we seek an approximation A_1 to $J_{\mathbf{F}}(\mathbf{x}^{(1)}$ that has these properties:

- $A_1(\mathbf{x}^{(1)} - \mathbf{x}^{(0)}) = \mathbf{F}(\mathbf{x}^{(1)}) - \mathbf{F}(\mathbf{x}^{(0)})$
- If $\mathbf{z}^T(\mathbf{x}^{(1)} - \mathbf{x}^{(0)}) = 0$, then $A_1\mathbf{z} = J_{\mathbf{F}}(\mathbf{x}^{(0)})\mathbf{z} = A_0\mathbf{z}$.

It follows from previous discussion that

$$A_1 = A_0 + \frac{\mathbf{y}_1 - A_0\mathbf{s}_1}{\mathbf{s}_1^T\mathbf{s}_1}\mathbf{s}_1^T,$$

where

$$\mathbf{s}_1 = \mathbf{x}^{(1)} - \mathbf{x}^{(0)}, \quad \mathbf{y}_1 = F(\mathbf{x}^{(1)}) - \mathbf{F}(\mathbf{x}^{(0)}).$$

Furthermore, once we have computed A_0^{-1}, we have

$$A_1^{-1} = A_0^{-1} - \frac{A_0^{-1}\left(\frac{\mathbf{y}_1 - A_0\mathbf{s}_1}{\mathbf{s}_1^T\mathbf{s}_1}\mathbf{s}_1^T\right)A_0^{-1}}{1 + \mathbf{s}_1^T A_0^{-1}\left(\frac{\mathbf{y}_1 - A_0\mathbf{s}_1}{\mathbf{s}_1^T\mathbf{s}_1}\right)} = A_0^{-1} + \frac{(\mathbf{s}_1 - A_0^{-1}\mathbf{y}_1)\mathbf{s}_1^T A_0^{-1}}{\mathbf{s}_1^T A_0^{-1}\mathbf{y}_1}.$$

Then, as A_1 is an approximation to $J_{\mathbf{F}}(\mathbf{x}^{(1)})$, we can obtain our next iterate $\mathbf{x}^{(2)}$ as follows:

$$A_1\mathbf{s}_2 = -\mathbf{F}(\mathbf{x}^{(1)}), \quad \mathbf{x}^{(2)} = \mathbf{x}^{(1)} + \mathbf{s}_2.$$

Repeating this process, we obtain the following algorithm, which is known as *Broyden's Method*:

```
Choose $\mathbf{x}^{(0)}$
$A_0 = J_{\mathbf{F}}(\mathbf{x}^{(0)})$
$\mathbf{s}_1 = -A_0^{-1}\mathbf{F}(\mathbf{x}^{(0)})$
$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \mathbf{s}_1$
$k = 1$
while not converged do
```

```
    $\mathbf{y}_k = \mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{F}(\mathbf{x}^{(k-1)})$
    $\mathbf{w}_k = A_{k-1}^{-1}\mathbf{y}_k$
    $c = 1/\mathbf{s}_k^T\mathbf{w}_k$
    $A_k^{-1} = A_{k-1}^{-1} + c(\mathbf{s}_k - \mathbf{w}_k)\mathbf{s}_k^T A_{k-1}^{-1}$
    $\mathbf{s}_{k+1} = -A_k^{-1}\mathbf{F}(\mathbf{x}^{(k)})$
    $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{s}_{k+1}$
    $k = k + 1$
end
```

Note that it is not necessary to compute A_k for $k \geq 1$; only A_k^{-1} is needed. It follows that no systems of linear equations need to be solved during an iteration; only matrix-vector multiplications are required, thus saving an order of magnitude of computational effort during each iteration compared to Newton's Method.

Exercise 8.6.9 *Write a* MATLAB *function* `x=broyden(F,JF,x0,tol)` *that solves the system of equations* $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ *using Broyden's method, where* `JF` *is a function handle that takes* `x` *as input and returns the matrix* $J_{\mathbf{F}}(\mathbf{x})$. *The parameters* `x0` *and* `tol` *are the initial guess and absolute error tolerance, respectively. Test your function on the system from Example 8.6.2 and compare the efficiency to that of Newton's method as implemented in Exercise 8.6.6.*

Chapter 9

Initial Value Problems

In this chapter, we begin our exploration of the development of numerical methods for solving *differential equations*, which are equations that depend on derivatives of unknown quantities. Differential equations arise in mathematical models of a wide variety of phenomena, such as propagation of waves, dissipation of heat energy, population growth, or motion of fluids. Solutions of differential equations yield valuable insight about such phenomena, and therefore techniques for solving differential equations are among the most essential methods of applied mathematics.

We now illustrate mathematical models based on differential equations. Newton's Second Law states

$$F = ma = m\frac{dv}{dt},$$

where F, m, a and v represent force, mass, acceleration and velocity, respectively. We use this law to develop a mathematical model for the velocity of a falling object that includes a differential equation. The forces on the falling object include gravity and air resistance, or drag; to simplify the discussion, we neglect any other forces.

The force due to gravity is equal to mg, where g is the acceleration due to gravity, and the drag force is equal to $-\gamma v$, where γ is the drag coefficient. We use downward orientation, so that gravity is acting in the positive (downward) direction and drag is acting in the negative (upward) direction. In summary, we have

$$F = mg - \gamma v.$$

Combining with Newton's Second Law yields the differential equation

$$m\frac{dv}{dt} = mg - \gamma v$$

for the velocity v of the falling object.

Another example of a mathematical model is a differential equation for the population p of a species, which can have the form

$$\frac{dp}{dt} = rp - d,$$

where the constant r is the rate of reproduction of the species. In general, r is called a *rate constant* or *growth rate*. The constant d indicates the number of specimens that die per unit of time, perhaps due to predation or other causes.

A differential equation such as this one does not have a unique solution, as it does not include enough information. Typically, the differential equation is paired with an **initial condition** of the form

$$y(t_0) = y_0,$$

where t_0 represents an **initial time** and y_0 is an **initial value**. The differential equation, together with the intial condition, is called an **initial value problem**. As discussed in the next section, under certain assumptions, it can be proven that an initial value problem has a unique solution. This chapter explores the numerical solution of initial value problems. Chapter 10 investigates the numerical solution of *boundary value problems*, which are differential equations defined on a spatial domain, such as a bounded interval $[a, b]$, paired with *boundary conditions* that ensure a unique solution.

There are many types of differential equations, and a wide variety of solution techniques, even for equations of the same type, let alone different types. We now introduce some terminology that aids in classification of equations and, by extension, selection of solution techniques.

- An *ordinary differential equation*, or ODE, is an equation that depends on one or more derivatives of functions of a single variable. Differential equations given in the preceding examples are all ordinary dfiferential equations, and we will consider these equations exclusively in this book.

- A *partial differential equation*, or PDE, is an equation that depends on one or more *partial* derivatives of functions of several variables. In many cases, PDEs are solved by reducing to multiple ODEs.

  Example 9.0.1 *The* heat equation

  $$\frac{\partial u}{\partial t} = k^2 \frac{\partial^2 u}{\partial x^2},$$

  where k is a constant, is an example of a partial differential equation, as its solution $u(x, t)$ is a function of two independent variables, and the equation includes partial derivatives with respect to both variables. □

- The *order* of a differential equation is the order of the highest derivative of any unknown function in the equation.

  Example 9.0.2 *The differential equation*

  $$\frac{dy}{dt} = ay - b,$$

  where a and b are constants, is a first-order differential equation, as only the first derivative of the solution $y(t)$ appears in the equation. On the other hand, the ODE

  $$y'' + 3y' + 2y = 0$$

  is a second-order differential equation, whereas the PDE known as the beam equation

  $$u_t = u_{xxxx}$$

  is a fourth-order differential equation. □

In this chapter, we limit ourselves to numerical methods for the solution of first-order ODEs. In Section 9.6, we consider systems of first-order ODEs, which allows these numerical methods to be applied to higher-order ODEs.

9.1 Existence and Uniqueness of Solutions

Consider the general first-order *initial value problem*, or *IVP*, that has the form

$$y' = f(t, y), \quad t_0 \leq t \leq T, \tag{9.1}$$

$$y(t_0) = y_0 \tag{9.2}$$

We would like to have an understanding of when this problem can be solved, and whether any solution that can be obtained is unique. The following notion of continuity, applied previously in Section 8.3 to establish convergence criteria for Fixed-Point Iteration, is helpful for this purpose.

Definition 9.1.1 (Lipschitz condition) *A function $f(t, y)$ satisfies a* **Lipschitz condition** *in y on $D \subset \mathbb{R}^2$ if*

$$|f(t, y_2) - f(t, y_1)| \leq L|y_2 - y_1|, \quad (t, y_1), (t, y_2) \in D, \tag{9.3}$$

for some constant $L > 0$, which is called a Lipschitz constant *for f.*

If, in addition, $\partial f/\partial y$ exists on D, we can also conclude that $|\partial f/\partial y| \leq L$ on D.

When solving a problem numerically, it is not sufficient to know that a unique solution exists. As discussed in Section 1.4.4, if a small change in the problem data can cause a substantial change in the solution, then the problem is *ill-conditioned*, and a numerical solution is therefore unreliable, because it could be unduly influenced by roundoff error. The following definition characterizes problems involving differential equations for which numerical solution is feasible.

Definition 9.1.2 (Well-posed problem) *A differential equation of any type, in conjunction with any other information such as an initial condition, is said to describe a* **well-posed problem** *if it satisfies three conditions, known as* **Hadamard's conditions** *for well-posedness:*

- *A solution of the problem exists.*
- *A solution of the problem is unique.*
- *The unique solution* depends continuously *on the problem data*

If a problem is not well-posed, then it is said to be **ill-posed**.

The "problem data" in this definiton may include, for example, initial values or coefficients of the differential equation.

We are now ready to describe a class of initial-value problems that can be solved numerically.

Theorem 9.1.3 (Existence-Uniqueness, Well-Posedness) *Let $D = [t_0, T] \times \mathbb{R}$, and let $f(t, y)$ be continuous on D. If f satisfies a Lipschitz condition on D in y, then the initial value problem (9.1), (9.2) has a unique solution $y(t)$ on $[t_0, T]$. Furthermore, the problem is well-posed.*

This theorem can be proved using Fixed-Point Iteration, in which the Lipschitz condition on f is used to prove that the iteration converges [7, p. 142-155].

Exercise 9.1.1 *Consider the initial value problem*

$$y' = 3y + 2t, \quad 0 < t \leq 1, \quad y(0) = 1.$$

Show that this problem is well-posed.

9.2 One-Step Methods

Numerical methods for the initial-value problem (9.1), (9.2) can be developed using Taylor series. We wish to approximate the solution at times t_n, $n = 1, 2, \ldots$, where

$$t_n = t_0 + nh,$$

with h being a chosen **time step**. Computing approximate solution values in this manner is called **time-stepping** or **time-marching**. Taking a Taylor expansion of the exact solution $y(t)$ at $t = t_{n+1}$ around the center $t = t_n$, we obtain

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(\xi),$$

where $t_n \leq \xi \leq t_{n+1}$.

9.2.1 Euler's Method

Using the fact that $y' = f(t, y)$, we obtain a numerical scheme by truncating the Taylor series after the second term. The result is a **difference equation**

$$y_{n+1} = y_n + hf(t_n, y_n),$$

where each y_n, for $n = 1, 2, \ldots$, is an approximation of $y(t_n)$. This method is called **Euler's method**, the simplest example of what is known as a **one-step method**.

We now need to determine whether this method **converges**; that is, whether

$$\lim_{h\to 0} \max_{0\leq n\leq T/h} |y(t_n) - y_n| = 0.$$

To that end, we attempt to bound the error at time t_n. We begin with a comparison of the difference equation and the Taylor expansion of the exact solution,

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + hf(t_n, y(t_n)) + \frac{h^2}{2}y''(\xi), \\ y_{n+1} &= y_n + hf(t_n, y_n) \end{aligned}$$

It follows that if we define $e_n = y(t_n) - y_n$, then

$$e_{n+1} = e_n + h[f(t_n, y(t_n) - f(t_n, y_n)] + \frac{h^2}{2}y''(\xi).$$

Using the assumption that f satisfies a Lipschitz condition (9.3) in y, we obtain

$$|e_{n+1}| \leq (1+hL)|e_n| + \frac{h^2M}{2}, \tag{9.4}$$

where

$$|y''(t)| \leq M, \quad t_0 \leq t \leq T,$$

and L is the Lipschitz constant for f in y on $[t_0, T] \times \mathbb{R}$.

Applying the relationship (9.4) repeatedly yields

$$\begin{aligned} |e_n| &\leq (1+hL)^n|e_0| + \frac{h^2M}{2}\sum_{i=0}^{n-1}(1+hL)^i \\ &\leq \frac{h^2M}{2}\frac{(1+hL)^n-1}{(1+hL)-1} \\ &\leq \frac{h^2M}{2}\frac{[e^{hL}]^n-1}{hL} \\ &\leq \frac{hM}{2L}[e^{L(t_n-t_0)}-1]. \end{aligned}$$

Here, we have used the formula for the partial sum of a geometric series,

$$1 + r + r^2 + \cdots + r^{n-1} = \frac{r^n-1}{r-1}.$$

We conclude that for $t_0 \leq t_n \leq T$,

$$|y(t_n) - y_n| \leq \frac{hM}{2L}[e^{L(t_n-t_0)}-1] \leq \frac{hM}{2L}[e^{L(T-t_0)}-1].$$

That is, as $h \to 0$, the solution obtained using Euler's method converges to the exact solution, and the convergence is $O(h)$; that is, first-order.

This convergence analysis, however, assumes exact arithmetic. To properly account for roundoff error, we note that the approximate solution values $\tilde{y}_n$, $n = 0, 1, 2, \ldots$, satisfy the modified difference equation

$$\tilde{y}_{n+1} = \tilde{y}_n + hf(t_n, \tilde{y}_n) + \delta_{n+1}, \quad \tilde{y}_0 = y_0 + \delta_0, \tag{9.5}$$

where, for $n = 0, 1, 2, \ldots$, $|\delta_n| \leq \delta$, which is $O(\mathbf{u})$, where $\mathbf{u}$ is the the machine precision (i.e., unit roundoff) introduced in Section 1.5.1. Note that even the initial value $\tilde{y}_0$ has an error term, which arises from representation of y_0 in the floating-point system.

Exercise 9.2.1 *Repeat the convergence analysis for Euler's method on (9.5) to obtain the error bound*

$$|y(t_n) - \tilde{y}_n| \leq \frac{1}{L}\left(\frac{hM}{2} + \frac{\delta}{h}\right)[e^{L(t_n-t_0)}-1] + \delta e^{L(t_n-t_0)}.$$

What happens to this error bound as $h \to 0$? What is an optimal choice of h so that the error bound is minimized?

We conclude our discussion of Euler's method with an example of how the previous convergence analyses can be used to select a suitable time step h.

Example 9.2.1 *Consider the IVP*

$$y' = -y, \quad 0 < t < 10, \quad y(0) = 1.$$

We know that the exact solution is $y(t) = e^{-t}$*. Euler's method applied to this problem yields the difference equation*

$$y_{n+1} = y_n - hy_n = (1-h)y_n, \quad y_0 = 1.$$

We wish to select h *so that the error at time* $T = 10$ *is less than* 0.001*. To that end, we use the error bound*

$$|y(t_n) - y_n| \leq \frac{hM}{2L}[e^{L(t_n - t_0)} - 1],$$

with $M = 1$*, since* $y''(t) = e^{-t}$*, which satisfies* $0 < y''(t) < 1$ *on* $[0, 10]$*, and* $L = 1$*, since* $f(t, y) = -y$ *satisfies* $|\partial f/\partial y| = |-1| \equiv 1$*. Substituting* $t_n = 10$ *and* $t_0 = 0$ *yields*

$$|y(10) - y_n| \leq \frac{h}{2}[e^{10} - 1] \approx 1101.27h.$$

Ensuring that the error at this time is less than 10^{-3} *requires choosing* $h < 9.08 \times 10^{-8}$*. However, the bound on the error at* $t = 10$ *is quite crude. Applying Euler's method with this time step yields a solution whose error at* $t = 10$ *is* 2×10^{-11}*.*

Now, suppose that we include roundoff error in our error analysis. The optimal time step is

$$h = \sqrt{\frac{2\delta}{M}},$$

where δ *is a bound on the roundoff error during any time step. We use* $\delta = 2\mathbf{u}$*, where* $\mathbf{u}$ *is the unit roundoff, because each time step performs only two floating-point operations. Even if* $1 - h$ *is computed once, in advance, its error still propagates to the multiplication with* y_n*. In a typical double-precision floating-point number system,* $\mathbf{u} \approx 1.1 \times 10^{-16}$*. It follows that the optimal time step is*

$$h = \sqrt{\frac{2\delta}{M}} = \sqrt{\frac{2(1.1 \times 10^{-16})}{1}} \approx 1.5 \times 10^{-8}.$$

With this value of h*, we find that the error at* $t = 10$ *is approximately* 3.55×10^{-12}*. This is even more accurate than with the previous choice of time step, which makes sense, because the new value of* h *is smaller.* □

9.2.2 Solving IVPs in MATLAB

MATLAB provides several functions for solving IVPs [35]. To solve an IVP of the form

$$y' = f(t, y), \quad t_0 < t \leq T, \quad y(t_0) = y_0,$$

one can use, for example, the command

```
>> [t,y]=ode23(f,[ t_0 T ],y_0);
```

where `f` is a function handle for $f(t, y)$. The first output `t` is a column vector consisting of times $t_0, t_1, \ldots, t_n = T$, where n is the number of time steps. The second output `y` is a $n \times m$ matrix, where m is the length of `y_0`. The ith row of `y` consists of the values of $y(t_i)$, for $i = 1, 2, \ldots, n$. This is the simplest usage of one of the ODE solvers; additional interfaces are described in the documentation.

Exercise 9.2.2 *(a) Write a* MATLAB *function* `[T,Y]=eulersmethod(f,tspan,y0,h)` *that solves a given IVP of the form (9.1), (9.2) using Euler's method. Assume that* `tspan` *is a vector of the form* $\left[\begin{array}{cc} t_0 & T \end{array}\right]$ *that contains the initial and final times, as in the typical usage of* MATLAB *ODE solvers. The output* `T` *must be a column vector of time values, and the output* `Y` *must be a matrix, each row of which represents the computed solution at the corresponding time value in the same row of* `T`*.*

(b) Test your function on the IVP from Example 9.2.1 with `h=0.1` *and* `h=0.01`*, and compute the error at the final time* t *using the known exact solution. What happens to the error as* `h` *decreases? Is the behavior what you would expect based on theory?*

9.2.3 Runge-Kutta Methods

We have seen that Euler's method is first-order accurate. We would like to use Taylor series to design methods that have a higher order of accuracy. First, however, we must get around the fact that an analysis of the global error, as was carried out for Euler's method, is quite cumbersome. Instead, we will design new methods based on the criteria that their local truncation error, the error committed during a single time step, is higher-order in h.

Using higher-order Taylor series directly to approximate $y(t_{n+1})$ is cumbersome, because it requires evaluating derivatives of f. Therefore, our approach will be to use evaluations of f at carefully chosen values of its arguments, t and y, in order to create an approximation that is just as accurate as a higher-order Taylor series expansion of $y(t + h)$.

To find the right values of t and y at which to evaluate f, we need to take a Taylor expansion of f evaluated at these (unknown) values, and then *match* the resulting numerical scheme to a Taylor series expansion of $y(t + h)$ around t.

We now illustrate our proposed approach in order to obtain a method that is second-order accurate; that is, its local truncation error is $O(h^2)$. The proposed method has the form

$$y_{n+1} = y_n + hf(t + \alpha_1, y + \beta_1),$$

where α_1 and β_1 are to be determined. To ensure second-order accuracy, we must match the Taylor expansion of the exact solution,

$$y(t + h) = y(t) + hf(t, y(t)) + \frac{h^2}{2}\frac{d}{dt}[f(t, y(t))] + \frac{h^3}{6}\frac{d^2}{dt^2}[f(\xi, y)],$$

to

$$y(t + h) = y(t) + hf(t + \alpha_1, y + \beta_1),$$

where $t \leq \xi \leq t + h$. After simplifying by removing terms or factors that already match, we see that we only need to match

$$f(t, y) + \frac{h}{2}\frac{d}{dt}[f(t, y)] + \frac{h^2}{6}\frac{d^2}{dt^2}[f(t, y)]$$

with

$$f(t+\alpha_1, y+\beta_1),$$

at least up to and including terms of $O(h)$, so that the local truncation error will be $O(h^2)$.

Applying the multivariable version of Taylor's theorem to f (see Theorem A.6.6), we obtain

$$\begin{aligned} f(t+\alpha_1, y+\beta_1) &= f(t,y) + \alpha_1 \frac{\partial f}{\partial t}(t,y) + \beta_1 \frac{\partial f}{\partial y}(t,y) + \\ & \quad \frac{\alpha_1^2}{2}\frac{\partial^2 f}{\partial t^2}(\xi,\mu) + \alpha_1\beta_1 \frac{\partial^2 f}{\partial t \partial y}(\xi,\mu) + \frac{\beta_1^2}{2}\frac{\partial^2 f}{\partial y^2}(\xi,\mu), \end{aligned}$$

where ξ is between t and $t+\alpha_1$ and μ is between y and $y+\beta_1$. Meanwhile, computing the full derivatives with respect to t in the Taylor expansion of the solution yields

$$f(t,y) + \frac{h}{2}\frac{\partial f}{\partial t}(t,y) + \frac{h}{2}\frac{\partial f}{\partial y}(t,y) f(t,y) + O(h^2).$$

Comparing terms yields

$$\alpha_1 = \frac{h}{2}, \quad \beta_1 = \frac{h}{2} f(t,y).$$

The resulting numerical scheme is

$$y_{n+1} = y_n + hf\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} f(t_n, y_n)\right).$$

This scheme is known as the **midpoint method**, or the **explicit midpoint method**. Note that it evaluates f at the midpoint of the intervals $[t_n, t_{n+1}]$ and $[y_n, y_{n+1}]$, where the midpoint in y is approximated using Euler's method with time step $h/2$.

The midpoint method is the simplest example of a **Runge-Kutta method**, which is the name given to any of a class of time-stepping schemes that are derived by matching multivaraiable Taylor series expansions of $f(t,y)$ with terms in a Taylor series expansion of $y(t+h)$. Another often-used Runge-Kutta method is the **modified Euler method**

$$y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_n + hf(t_n, y_n))], \tag{9.6}$$

also known as the **explicit trapezoidal method**, as it resembles the Trapezoidal Rule from numerical integration. This method is also second-order accurate.

Exercise 9.2.3 *Derive the explicit trapezoidal method (9.6) by finding a method of the form*

$$y_{n+1} = y_n + h[a_1 f(t_n + \alpha_1, y_n + \beta_1) + a_2 f(t_n + \alpha_2, y_n + \beta_2)]$$

that is second-order accurate.

However, the best-known Runge-Kutta method is the **fourth-order Runge-Kutta method**, which uses *four* evaluations of f during each time step. The method proceeds as follows:

$$\begin{aligned} k_1 &= hf(t_n, y_n), \\ k_2 &= hf\left(t_n + \frac{h}{2}, y_n + \frac{1}{2}k_1\right), \\ k_3 &= hf\left(t_n + \frac{h}{2}, y_n + \frac{1}{2}k_2\right), \\ k_4 &= hf\left(t_{n+1}, y_n + k_3\right), \\ y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned} \tag{9.7}$$

In a sense, this method is similar to Simpson's Rule from numerical integration, which is also fourth-order accurate, as values of f at the midpoint in time are given four times as much weight as values at the endpoints t_n and t_{n+1}.

The values $k_1, \ldots, k_4$ are referred to as **stages**; more precisely, a stage of a Runge-Kutta method is an evaluation of $f(t, y)$, and the number of stages of a Runge-Kutta method is the number of evaluations required per time step. We therefore say that (9.7) is a four-stage, fourth-order method, while the explicit trapezoidal method (9.6) is a two-stage, second-order method. We will see in Section 9.5 that the number of stages does not always correspond to the order of accuracy.

Example 9.2.2 *We compare Euler's method with the fourth-order Runge-Kutta scheme on the initial value problem*

$$y' = -2ty, \quad 0 < t \leq 1, \quad y(0) = 1,$$

which has the exact solution $y(t) = e^{-t^2}$. *We use a time step of* $h = 0.1$ *for both methods. The computed solutions, and the exact solution, are shown in Figure 9.1.*

It can be seen that the fourth-order Runge-Kutta method is far more accurate than Euler's method, which is first-order accurate. In fact, the solution computed using the fourth-order Runge-Kutta method is visually indistinguishable from the exact solution. At the final time $T = 1$, *the relative error in the solution computed using Euler's method is 0.038, while the relative error in the solution computing using the fourth-order Runge-Kutta method is* 4.4×10^{-6}. □

Exercise 9.2.4 *Modify your function* `eulersmethod` *from Exercise 9.2.2 to obtain a new function* `[T,Y]=rk4(f,tspan,y0,h)` *that implements the fourth-order Runge-Kutta method.*

9.2.4 Implicit Methods

Suppose that we approximate the equation

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} y'(s)\, ds$$

by applying the Trapezoidal Rule to the integral. This yields a one-step method

$$y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1})], \tag{9.8}$$

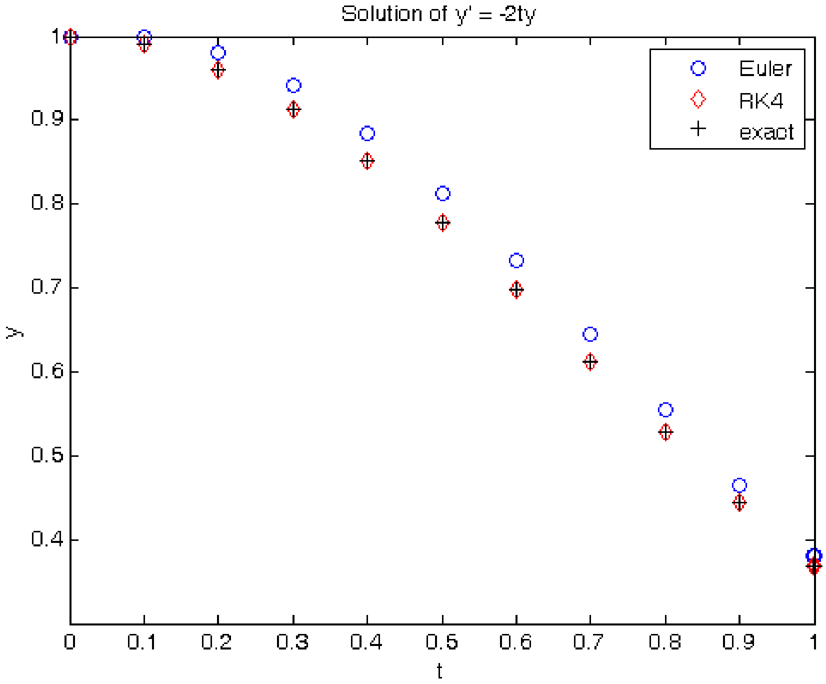


Figure 9.1: Solutions of $y' = -2ty$, $y(0) = 1$ on $[0, 1]$, computed using Euler's method and the fourth-order Runge-Kutta method

known as the **trapezoidal method**.

The trapezoidal method constrasts with Euler's method because it is an **implicit** method, due to the evaluation of $f(t, y)$ at y_{n+1}. It follows that it is generally necessary to solve a nonlinear equation to obtain y_{n+1} from y_n. This additional computational effort is offset by the fact that implicit methods are generally more stable than **explicit** methods such as Euler's method. Another example of an implicit method is **backward Euler's method**

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}). \tag{9.9}$$

Like Euler's method, backward Euler's method is first-order accurate.

Exercise 9.2.5 *Write a* MATLAB *function* `[T,Y]=backwardeuler(f,tspan,y0,h)` *that implements backward Euler's method (9.9). Use the secant method to solve for y_{n+1} at each time step. For initial guesses, use y_n and $y_n + hf(t_n, y_n)$, the approximation of y_{n+1} obtained using (forward) Euler's method.*

Exercise 9.2.6 *Suppose that fixed-point iteration is used to solve for y_{n+1} in backward Euler's method. What is the function g in the equation $y_{n+1} = g(y_{n+1})$? Assuming that g satisfies the condition for a fixed point to exist, how should h be chosen to help ensure convergence of the fixed-point iteration?*

Exercise 9.2.7 *Repeat Exercise 9.2.6 for the trapezoidal method (9.8).*

9.3 Multistep Methods

All of the numerical methods that we have developed for solving initial value problems are classified as *one-step methods*, because they only use information about the solution at time t_n to approximate the solution at time t_{n+1}. As n increases, that means that there are additional values of the solution, at previous times, that could be helpful, but are unused.

Multistep methods are time-stepping methods that do use this information. A general multistep method has the form

$$\sum_{i=0}^{s} \alpha_i y_{n+1-i} = h \sum_{i=0}^{s} \beta_i f(t_{n+1-i}, y_{n+1-i}),$$

where s is the number of steps in the method ($s = 1$ for a one-step method), and h is the time step size, as before.

By convention, $\alpha_0 = 1$, so that y_{n+1} can be conveniently expressed in terms of other values. If $\beta_0 = 0$, the multistep method is said to be *explicit*, because then y_{n+1} can be described using an explicit formula, whereas if $\beta_0 \neq 0$, the method is *implicit*, because then an equation, generally nonlinear, must be solved to compute y_{n+1}.

For a general implicit multistep method, for which $\beta_0 \neq 0$, Newton's method can be applied to the function

$$F(y) = \alpha_0 y + \sum_{i=1}^{s} \alpha_i y_{n+1-i} - h\beta_0 f(t_{n+1}, y) - h \sum_{i=1}^{s} \beta_i f_{n+1-i}.$$

The resulting iteration is

$$\begin{aligned} y_{n+1}^{(k+1)} &= y_{n+1}^{(k)} - \frac{F(y_{n+1}^{(k)})}{F'(y_{n+1}^{(k)})} \\ &= y_{n+1}^{(k)} - \frac{\alpha_0 y_{n+1}^{(k)} + \sum_{i=1}^{s} \alpha_i y_{n+1-i} - h\beta_0 f(t_{n+1}, y_{n+1}^{(k)}) - h\sum_{i=1}^{s} \beta_i f_{n+1-i}}{\alpha_0 - h\beta_0 f_y(t_{n+1}, y_{n+1}^{(k)})}, \end{aligned}$$

with $y_{n+1}^{(0)} = y_n$. If one does not wish to compute f_y, then the Secant Method can be used instead.

9.3.1 Adams Methods

Adams methods [4] involve the integral form of the ODE,

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(s, y(s))\, ds.$$

The general idea behind Adams methods is to approximate the above integral using polynomial interpolation of f at the points $t_{n+1-s}, t_{n+2-s}, \ldots, t_n$ if the method is explicit, and t_{n+1} as well if the method is implicit. In all Adams methods, $\alpha_0 = 1$, $\alpha_1 = -1$, and $\alpha_i = 0$ for $i = 2, \ldots, s$.

Explicit Adams methods are called **Adams-Bashforth methods**. To derive an Adams-Bashforth method, we interpolate f at the points $t_n, t_{n-1}, \ldots, t_{n-s+1}$ with a polynomial of degree $s - 1$. We then integrate this polynomial exactly. It follows that the constants β_i, $i = 1, \ldots, s$, are the integrals of the corresponding Lagrange polynomials from t_n to t_{n+1}, divided by h, because there is already a factor of h in the general multistep formula.

Example 9.3.1 *We derive the three-step Adams-Bashforth method,*

$$y_{n+1} = y_n + h[\beta_1 f(t_n, y_n) + \beta_2 f(t_{n-1}, y_{n-1}) + \beta_3 f(t_{n-2}, y_{n-2}).$$

The constants β_i, $i = 1, 2, 3$, are obtained by evaluating the integral from t_n to t_{n+1} of a polynomial $p_2(t)$ that passes through $f(t_n, y_n)$, $f(t_{n-1}, y_{n-1})$, and $f(t_{n-2}, y_{n-2})$.

Because we can write

$$p_2(t) = \sum_{i=0}^{2} f(t_{n-i}, y_{n-i}) L_i(t),$$

where $L_i(t)$ is the ith Lagrange polynomial for the interpolation points t_n, t_{n-1} and t_{n-2}, and because our final method expresses y_{n+1} as a linear combination of y_n and values of f, it follows that the constants β_i, $i = 1, 2, 3$, are the integrals of the Lagrange polynomials from t_n to t_{n+1}, divided by h.

However, using a change of variable $u = (t_{n+1} - s)/h$, we can instead interpolate at the points $u = 1, 2, 3$, thus simplifying the integration. If we define $\tilde{p}_2(u) = p_2(s) = p_2(t_{n+1} - hu)$ and $\tilde{L}_i(u) = L_i(t_{n+1} - hu)$, then we obtain

$$\begin{aligned}
\int_{t_n}^{t_{n+1}} f(s, y(s))\, ds &= \int_{t_n}^{t_{n+1}} p_2(s)\, ds \\
&= h \int_0^1 \tilde{p}_2(u)\, du \\
&= h \int_0^1 f(t_n, y_n)\tilde{L}_0(u) + f(t_{n-1}, y_{n-1})\tilde{L}_1(u) + f(t_{n-2}, y_{n-2})\tilde{L}_2(u)\, du \\
&= h \left[f(t_n, y_n) \int_0^1 \tilde{L}_0(u)\, du + f(t_{n-1}, y_{n-1}) \int_0^1 \tilde{L}_1(u)\, du + \right. \\
&\qquad \left. f(t_{n-2}, y_{n-2}) \int_0^1 \tilde{L}_2(u)\, du \right] \\
&= h \left[f(t_n, y_n) \int_0^1 \frac{(u-2)(u-3)}{(1-2)(1-3)}\, du + f(t_{n-1}, y_{n-1}) \int_0^1 \frac{(u-1)(u-3)}{(2-1)(2-3)}\, du + \right. \\
&\qquad \left. f(t_{n-2}, y_{n-2}) \int_0^1 \frac{(u-1)(u-2)}{(3-1)(3-2)}\, du \right] \\
&= h \left[\frac{23}{12} f(t_n, y_n) - \frac{4}{3} f(t_{n-1}, y_{n-1}) + \frac{5}{12} f(t_{n-2}, y_{n-2}) \right].
\end{aligned}$$

We conclude that the three-step Adams-Bashforth method is

$$y_{n+1} = y_n + \frac{h}{12}[23f(t_n, y_n) - 16f(t_{n-1}, y_{n-1}) + 5f(t_{n-2}, y_{n-2})]. \tag{9.10}$$

This method is third-order accurate. □

The same approach can be used to derive an implicit Adams method, which is known as an **Adams-Moulton method**. The only difference is that because t_{n+1} is an interpolation point, after the change of variable to u, the interpolation points $0, 1, 2, \ldots, s$ are used. Because the resulting interpolating polynomial is of degree one greater than in the explicit case, the error in an s-step Adams-Moulton method is $O(h^{s+1})$, as opposed to $O(h^s)$ for an s-step Adams-Bashforth method.

Exercise 9.3.1 *Derive the four-step Adams-Bashforth method*

$$y_{n+1} = y_n + \frac{h}{24}[55f(t_n, y_n) - 59f(t_{n-1}, y_{n-1}) + 37f(t_{n-2}, y_{n-2}) - 9f(t_{n-3}, y_{n-3})] \quad (9.11)$$

and the three-step Adams-Moulton method

$$y_{n+1} = y_n + \frac{h}{24}[9f(t_{n+1}, y_{n+1}) + 19f(t_n, y_n) - 5f(t_{n-1}, y_{n-1}) + f(t_{n-2}, y_{n-2})]. \quad (9.12)$$

What is the order of accuracy of each of these methods?

9.3.2 Predictor-Corrector Methods

An Adams-Moulton method can be impractical because, being implicit, it requires an iterative method for solving nonlinear equations, such as fixed-point iteration, and this method must be applied during every time step. An alternative is to pair an Adams-Bashforth method with an Adams-Moulton method to obtain an **Adams-Moulton predictor-corrector method** [25]. Such a method proceeds as follows:

- *Predict:* Use the Adams-Bashforth method to compute a first approximation to y_{n+1}, which we denote by $\tilde{y}_{n+1}$.
- *Evaluate:* Evaluate f at this value, computing $f(t_{n+1}, \tilde{y}_{n+1})$.
- *Correct:* Use the Adams-Moulton method to compute y_{n+1}, but instead of solving an equation, use $f(t_{n+1}, \tilde{y}_{n+1})$ in place of $f(t_{n+1}, y_{n+1})$ so that the Adams-Moulton method can be used as if it was an *explicit* method.
- *Evaluate:* Evaluate f at the newly computed value of y_{n+1}, computing $f(t_{n+1}, y_{n+1})$, to use during the next time step.

Example 9.3.2 *We illustrate the predictor-corrector approach with the two-step Adams-Bashforth method*

$$y_{n+1} = y_n + \frac{h}{2}[3f(t_n, y_n) - f(t_{n-1}, y_{n-1})]$$

and the two-step Adams-Moulton method

$$y_{n+1} = y_n + \frac{h}{12}[5f(t_{n+1}, y_{n+1}) + 8f(t_n, y_n) - f(t_{n-1}, y_{n-1})]. \quad (9.13)$$

First, we apply the Adams-Bashforth method, and compute

$$\tilde{y}_{n+1} = y_n + \frac{h}{2}[3f(t_n, y_n) - f(t_{n-1}, y_{n-1})].$$

Then, we compute $f(t_{n+1}, \tilde{y}_{n+1})$ and apply the Adams-Moulton method, to compute

$$y_{n+1} = y_n + \frac{h}{12}[5f(t_{n+1}, \tilde{y}_{n+1}) + 8f(t_n, y_n) - f(t_{n-1}, y_{n-1})].$$

This new value of y_{n+1} is used when evaluating $f(t_{n+1}, y_{n+1})$ during the next time step. □

One drawback of multistep methods is that because they rely on values of the solution from previous time steps, they cannot be used during the first time steps, because not enough values are available. Therefore, it is necessary to use a one-step method, with at least the same order of accuracy, to compute enough *starting values* of the solution to be able to use the multistep method. For example, to use the three-step Adams-Bashforth method, it is necessary to first use a one-step method such as the fourth-order Runge-Kutta method to compute y_1 and y_2, and then the Adams-Bashforth method can be used to compute y_3 using y_2, y_1 and y_0.

Exercise 9.3.2 *How many starting values are needed to use a s-step multistep method?*

Exercise 9.3.3 *Write a* MATLAB *function* `[T,Y]=adamsbashforth3(f,tspan,y0,h)` *that implements the 3-step Adams-Bashforth method (9.10) to solve the given IVP (9.1), (9.2). Use the fourth-order Runge-Kutta to generate the necessary starting values. Use different values of* `h` *on a sample IVP to confirm that your method is third-order accurate.*

Exercise 9.3.4 *Write a* MATLAB *function* `[T,Y]=predictcorrect(f,tspan,y0,h)` *that implements an Adams-Moulton predictor-corrector method using a four-step predictor (9.11) and three-step corrector (9.12). Use the fourth-order Runge-Kutta to generate the necessary starting values. Use different values of* `h` *on a sample IVP to confirm that your method is fourth-order accurate.*

9.3.3 Backward Differentiation Formulae

Another class of multistep methods, known as **Backward differentiation formulas (BDF)** [36, p. 349], can be derived using polynomial interpolation as in Adams methods, but for a different purpose–to approximate the *derivative* of y at t_{n+1}. This approximation is then equated to $f(t_{n+1}, y_{n+1})$. It follows that all methods based on BDFs are implicit, and they all satisfy $\beta_0 = 1$, with $\beta_i = 0$ for $i = 1, 2, \ldots, s$.

More precisely, a BDF has the form

$$\sum_{i=0}^{s} \alpha_i y_{n+1-i} = hf(t_{n+1}, y_{n+1}),$$

where

$$\alpha_i = \mathcal{L}'_{s,i}(t_{n+1}),$$

with $\mathcal{L}_{s,0}(t), \mathcal{L}_{s,1}(t), \ldots, \mathcal{L}_{s,s}(t)$ being the Lagrange polynomials for $t_{n+1-s}, \ldots, t_n, t_{n+1}$.

Exercise 9.3.5 *Show that a 1-step BDF is simply backward Euler's method.*

Exercise 9.3.6 *Derive a 2-step BDF. How do the coefficients α_i, $i = 0, 1, 2$, relate to those of the 3-point second-order numerical differentiation formula (7.3)?*

9.4 Convergence Analysis

We have previously determined that when applying Euler's method

$$y_{n+1} = y_n + hf(t_n, y_n)$$

to the initial value problem (9.1), (9.2), the error in the computed solution satisfies the error bound

$$|y(t_n) - y_n| \leq \frac{Mh}{2L}(e^{L(t_n - t_0)} - 1),$$

where L is the Lipschitz constant for f and M is an upper bound on $|y''(t)|$. This error bound indicates that the numerical solution *converges* to the exact solution at $h \to 0$; that is,

$$\lim_{h\to 0} \max_{0\leq n\leq (T-t_0)/h} |y(t_n) - y_n| = 0.$$

It would be desirable to be able to prove that a numerical method converges without having to proceed through the same detailed error analysis that was carried out with Euler's method, since other methods are more complex and such analysis would require more assumptions to obtain a similar bound on the error in y_n.

To that end, we define two properties that a numerical method should have in order to be convergent.

Definition 9.4.1

- **Consistency:** *a numerical method for the initial-value problem (9.1) is said to be* **consistent** *if*

  $$\lim_{h\to 0} \max_{0\leq n\leq (T-t_0)/h} |\tau_n(h)| = 0,$$

  where $\tau_n(h)$ is the local truncation error at time t_n.

- **Stability:** *a numerical method is said to be* **stable** *if there exists a constant α such that for any two numerical solutions y_n and $\tilde{y}_n$*

  $$|y_{n+1} - \tilde{y}_{n+1}| \leq (1 + \alpha h)|y_n - \tilde{y}_n|, \quad 0 \leq n \leq (T - t_0)/h.$$

  It follows from this relation that

  $$|y_n - \tilde{y}_n| \leq e^{\alpha(T-t_0)}|y_0 - \tilde{y}_0|.$$

Informally, a stable method converges to the differential equation as $h \to 0$, and the solution computed using a stable method is not overly sensitive to perturbations in the initial data. While the difference in solutions is allowed to grow over time, it is "controlled" growth, meaning that the rate of growth is independent of the step size h.

9.4.1 Consistency

The definition of consistency in Definition 9.4.1 can be cumbersome to apply directly to a given method. Therefore, we consider one-step and multistep methods separately to obtain simple approaches for determining whether a given method is consistent, and if so, its order of accuracy.

9.4.1.1 One-Step Methods

We have learned that the numerical solution obtained from Euler's method,

$$y_{n+1} = y_n + hf(t_n, y_n), \quad t_n = t_0 + nh,$$

converges to the exact solution $y(t)$ of the initial value problem

$$y' = f(t, y), \quad y(t_0) = y_0,$$

as $h \to 0$.

We now analyze the convergence of a general one-step method of the form

$$y_{n+1} = y_n + h\Phi(t_n, y_n, h), \tag{9.14}$$

for some continuous function $\Phi(t, y, h)$. We define the **local truncation error** of this one-step method by

$$\tau_n(h) = \frac{y(t_{n+1}) - y(t_n)}{h} - \Phi(t_n, y(t_n), h).$$

That is, the local truncation error is the result of substituting the exact solution into the approximation of the ODE by the numerical method.

Exercise 9.4.1 *Find the local truncation error of the modified Euler method (9.6).*

As $h \to 0$ and $n \to \infty$, in such a way that $t_0 + nh = t \in [t_0, T]$, we obtain

$$\tau_n(h) \to y'(t) - \Phi(t, y(t), 0).$$

It follows from Defintion 9.4.1 that the one-step method is **consistent** if

$$\Phi(t, y, 0) = f(t, y).$$

Recall that a consistent one-step method is one that converges to the ODE as $h \to 0$.

Example 9.4.2 *Consider the midpoint method*

$$y_{n+1} = y_n + hf\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}f(t_n, y_n)\right).$$

To check consistency, we compute

$$\phi(t, y, 0) = f\left(t + \frac{0}{2}, y + \frac{0}{2}f(t, y)\right) = f(t, y),$$

so it is consistent. □

Exercise 9.4.2 *Verify that the fourth-order Runge-Kutta method (9.7) is consistent.*

9.4.1.2 Multistep Methods

For multistep methods, we must define consistency slightly differently, because we must account for the fact that a multistep method requires starting values that are computed using another method.

Therefore, we say that a multistep method is consistent if its own local truncation error $\tau_n(h)$ approaches zero as $h \to 0$, *and* if the one-step method used to compute its starting values is also consistent. We also say that a s-step multistep method is stable, or **zero-stable**, if there exists a constant K such that for any two sequences of values $\{y_k\}$ and $\{z_k\}$ produced by the method with step size h from different sets of starting values $\{y_0, y_1, \ldots, y_{s-1}\}$ and $\{z_0, z_1, \ldots, z_{s-1}\}$,

$$|y_n - z_n| \leq K \max_{0\leq j\leq s-1} |y_j - z_j|,$$

as $h \to 0$.

To compute the local truncation error of Adams methods, integrate the error in the polynomial interpolation used to derive the method from t_n to t_{n+1}. For the explicit s-step method, this yields

$$\tau_n(h) = \frac{1}{h}\int_{t_n}^{t_{n+1}} \frac{f^{(s)}(\xi, y(\xi))}{s!}(t - t_n)(t - t_{n-1}) \cdots (t - t_{n-s+1})\,dt.$$

Using the substitution $u = (t_{n+1} - t)/h$, and the Weighted Mean Value Theorem for Integrals, yields

$$\tau_n(h) = \frac{1}{h}\frac{f^{(s)}(\xi, y(\xi))}{s!}h^{s+1}(-1)^s \int_0^1 (u - 1)(u - 2) \cdots (u - s)\,du.$$

Evaluating the integral yields the constant in the error term. We also use the fact that $y' = f(t, y)$ to replace $f^{(s)}(\xi, y(\xi))$ with $y^{(s+1)}(\xi)$. Obtaining the local truncation error for an implicit, Adams-Moulton method can be accomplished in the same way, except that t_{n+1} is also used as an interpolation point.

For a general multistep method, we substitute the exact solution into the method, as in one-step methods, and obtain

$$\tau_n(h) = \frac{\sum_{j=0}^s \alpha_j y(t_{n+1-j}) - h\sum_{j=0}^s \beta_j f(t_{n+1-j}, y(t_{n+1-j}))}{h\sum_{j=0}^s \beta_j},$$

where the scaling by $h\sum_{j=0}^s \beta_j$ is designed to make this definition of local truncation error consistent with that of one-step methods.

By replacing each evaluation of $y(t)$ by a Taylor series expansion around t_{n+1}, we obtain

$$\begin{aligned}
\tau_n(h) &= \frac{1}{h\sum_{j=0}^s \beta_j}\sum_{j=0}^s \left[\alpha_j \sum_{k=0}^\infty \frac{1}{k!}y^{(k)}(t_{n+1})(-jh)^k - h\beta_j \sum_{k=0}^\infty \frac{d^k}{dt^k}[f(t_{n+1}, y(t_{n+1}))](-jh)^k\right] \\
&= \frac{1}{h\sum_{j=0}^s \beta_j}\sum_{j=0}^s \left[\sum_{k=0}^\infty (-1)^k \frac{h^k}{k!}\alpha_j y^{(k)}(t_{n+1})j^k + \sum_{k=1}^\infty (-1)^k \frac{h^k}{(k-1)!}\beta_j y^{(k)}(t_{n+1})j^{k-1}\right] \\
&= \frac{1}{h\sum_{j=0}^s \beta_j}\left\{y(t_{n+1})\sum_{j=0}^s \alpha_j + \sum_{k=1}^\infty (-h)^k y^{(k)}(t_{n+1})\left[\frac{1}{k!}\sum_{j=1}^s j^k\alpha_j + \frac{1}{(k-1)!}\sum_{j=0}^s j^{k-1}\beta_j\right]\right\} \\
&= \frac{1}{h\sum_{j=0}^s \beta_j}\left[y(t_{n+1})C_0 + \sum_{k=1}^\infty (-h)^k y^{(k)}(t_{n+1})C_k\right]
\end{aligned}$$

where

$$C_0 = \sum_{j=0}^{s} \alpha_j, \quad C_k = \frac{1}{k!}\sum_{j=1}^{s} j^k \alpha_j + \frac{1}{(k-1)!}\sum_{j=0}^{s} j^{k-1}\beta_j, \quad k = 1, 2, \ldots.$$

We find that $\tau_n(h) \to 0$ as $h \to 0$ only if $C_0 = C_1 = 0$. Furthermore, the method is of order p if and only if

$$C_0 = C_1 = C_2 = \cdots = C_p = 0, \quad C_{p+1} \neq 0. \tag{9.15}$$

Finally, we can conclude that the local truncation error for a method of order p is

$$\tau_n(h) = \frac{1}{\sum_{j=0}^{s} \beta_j}(-h)^p y^{(p+1)}(t_{n+1})C_{p+1} + O(h^{p+1}).$$

Exercise 9.4.3 *Use the conditions (9.15) to verify the order of accuracy of the four-step Adams-Bashforth method (9.11). What is the local truncation error?*

Further analysis is required to obtain the local truncation error of a predictor-corrector method that is obtained by combining two Adams methods. The result of this analysis is the following theorem, which is proved in [19, p. 387-388].

Theorem 9.4.3 *Let the solution of the initial value problem*

$$y' = f(t, y), \quad t_0 < t \leq T, \quad y(t_0) = y_0$$

be approximated by the Adams-Moulton s-step predictor-corrector method with predictor

$$\tilde{y}_{n+1} = y_n + h\sum_{i=1}^{s} \tilde{\beta}_i f_{n+1-i}$$

and corrector

$$y_{n+1} = y_n + h\left[\beta_0 f(t_{n+1}, \tilde{y}_{n+1}) + \sum_{i=1}^{s} \beta_i f_{n+1-i}\right].$$

Then the local truncation error of the predictor-corrector method is

$$S_n(h) = \tilde{T}_n(h) + T_n(h)\beta_0 \frac{\partial f}{\partial y}(t_{n+1}, y(t_{n+1}) + \xi_{n+1})$$

where $T_n(h)$ *and* $\tilde{T}_n(h)$ *are the local truncation errors of the predictor and corrector, respectively, and* ξ_{n+1} *is between 0 and* $hT_n(h)$*. Furthermore, there exist constant* α *and* β *such that*

$$|y(t_n) - y_n| \leq \left[\max_{0 \leq i \leq s-1} |y(t_i) - y_i| + \beta S(h)\right] e^{\alpha(t_n - t_0)},$$

where $S(h) = \max_{s \leq n \leq (T-t_0)/h} |S_n(h)|$.

A single time step of a predictor-corrector method, as we have described it, can be viewed as an instance of Fixed-Point Iteration in which only one iteration is performed, with the initial guess being the prediction $\tilde{y}_{n+1}$ and the function $g(y)$ being the corrector. If desired, the iteration can be continued until convergence is achieved.

Exercise 9.4.4 *Show that an s-step predictor-corrector method, in which the corrector is repeatedly applied until* y_{n+1} *converges, has local truncation error* $O(h^{s+1})$.

9.4.2 Stability

We now specialize the definition of stability from Definition 9.4.1 to one-step and multistep methods, so that their stability (or lack thereof) can readily be determined.

9.4.2.1 One-Step Methods

From Definition 9.4.1, a one-step method of the form (9.14) is **stable** if $\Phi(t, y, h)$ is Lipschitz continuous in y. That is,

$$|\Phi(t, u, h) - \Phi(t, v, h)| \leq L_\Phi |u - v|, \quad t \in [t_0, T], \quad u, v \in \mathbb{R}, \quad h \in [0, h_0], \tag{9.16}$$

for some constant L_Φ.

Example 9.4.4 *Consider the midpoint method*

$$y_{n+1} = y_n + hf\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} f(t_n, y_n)\right).$$

First, we check whether

$$\phi(t, y, h) = f\left(t + \frac{h}{2}, y + \frac{h}{2} f(t, y)\right)$$

satisfies a Lipschitz condition in y. *We assume that* $f(t, y)$ *satisifes a Lipschitz condition in* y *on* $[t_0, T] \times (-\infty, \infty)$ *with Lipschitz constant* L. *Then we have*

$$\begin{aligned}
|\phi(t, y, h) - \phi(t, \tilde{y}, h)| &= \left| f\left(t + \frac{h}{2}, y + \frac{h}{2} f(t, y)\right) - f\left(t + \frac{h}{2}, \tilde{y} + \frac{h}{2} f(t, \tilde{y})\right)\right| \\
&\leq L\left|\left(y + \frac{h}{2} f(t, y)\right) - \left(\tilde{y} + \frac{h}{2} f(t, \tilde{y})\right)\right| \\
&\leq L\left[|y - \tilde{y}| + \frac{hL}{2}|y - \tilde{y}|\right] \\
&\leq \left(L + \frac{1}{2} hL^2\right)|y - \tilde{y}|.
\end{aligned}$$

It follows that $\phi(t, y, h)$ *satifies a Lipschitz condition on the domain* $[t_0, T] \times (-\infty, \infty) \times [0, h_0]$ *with Lipschitz constant* $\tilde{L} = L + \frac{1}{2} h_0 L^2$. *Therefore it is stable.* □

Exercise 9.4.5 *Prove that the modified Euler method (9.6) is stable.*

9.4.2.2 Multistep Methods

We now examine the stability of a general s-step multistep method of the form

$$\sum_{i=0}^{s} \alpha_i y_{n+1-i} = h \sum_{i=0}^{s} \beta_i f(t_{n+1-i}, y_{n+1-i}).$$

If this method is applied to the initial value problem

$$y' = 0, \quad y(t_0) = y_0, \quad y_0 \neq 0,$$

for which the exact solution is $y(t) = y_0$, then for the method to be stable, the computed solution must remain bounded.

It follows that the computed solution satisfies the m-term recurrence relation

$$\sum_{i=0}^{s} \alpha_i y_{n+1-i} = 0,$$

which has a solution of the form

$$y_n = \sum_{i=0}^{s} c_i n^{p_i} \lambda_i^n,$$

where the c_i and p_i are constants, and the λ_i are the roots of the **characteristic equation**

$$\alpha_0 \lambda^s + \alpha_1 \lambda^{s-1} + \cdots + \alpha_{s-1}\lambda + \alpha_s = 0.$$

When a root λ_i is distinct, $p_i = 0$. Therefore, to ensure that the solution does not grow exponentially, the method must satisfy the **root condition:**

- All roots must satisfy $|\lambda_i| \leq 1$.
- If $|\lambda_i| = 1$ for any i, then it must be a *simple root*, meaning that its multiplicity is one.

It can be shown that a multistep method is zero-stable if and only if it satisfies the root condition. Furthermore, $\lambda = 1$ is always a root, because in order to be consistent, a multistep method must have the property that $\sum_{i=0}^{s} \alpha_i = 0$. If this is the only root that has absolute value 1, then we say that the method is **strongly stable**, whereas if there are multiple roots that are distinct from one another, but have absolute value 1, then the method is said to be **weakly stable**.

Because all Adams methods have the property that $\alpha_0 = 1$, $\alpha_1 = -1$, and $\alpha_i = 0$ for $i = 2, 3, \ldots, s$, it follows that the roots of the characteristic equation are all zero, except for one root that is equal to 1. Therefore, all Adams methods are strongly stable. The same is not true for BDFs; they are zero-unstable for $s > 6$ [36, p. 349].

Example 9.4.5 *A multistep method that is neither an Adams method, nor a backward differentiation formula, is an implicit 2-step method known as* **Simpson's method***:*

$$y_{n+1} = y_{n-1} + \frac{h}{3}[f_{n+1} + 4f_n + f_{n-1}].$$

Although it is only a 2-step method, it is fourth-order accurate, due to the high degree of accuracy of Simpson's Rule.

This method is obtained from the relation satisfied by the exact solution,

$$y(t_{n+1}) = y(t_{n-1}) + \int_{t_{n-1}}^{t_{n+1}} f(t, y(t))\, dt.$$

Since the integral is over an interval of width $2h$*, it follows that the coefficients* β_i *obtained by polynomial interpolation of* f *must satisfy the condition*

$$\sum_{i=0}^{s} \beta_i = 2,$$

as opposed to summing to 1 for Adams methods.

For this method, we have $s = 2$, $\alpha_0 = 1$, $\alpha_1 = 0$ *and* $\alpha_2 = -1$*, which yields the characteristic polynomial* $\lambda^2 - 1$*. This polynomial has two distinct roots, 1 and* -1*, that both have absolute value 1. It follows that Simpson's method is only weakly stable.* □

Exercise 9.4.6 *Determine whether the 2-step BDF from Exercise (9.3.6) is strongly stable, weakly stable, or unstable.*

9.4.3 Convergence

It can be shown that a consistent and stable one-step method of the form (9.14) is convergent. Using the same approach and notation as in the convergence proof of Euler's method, and the fact that the method is stable, we obtain the following bound for the **global error** $e_n = y(t_n) - y_n$:

$$|e_n| \leq \left(\frac{e^{L_\Phi (T - t_0)} - 1}{L_\Phi} \right) \max_{0 \leq m \leq n-1} |\tau_m(h)|,$$

where L_Φ is the Lipschitz constant for Φ, as in (9.16).

Because the method is consistent, we have

$$\lim_{h \to 0} \max_{0 \leq n \leq T/h} |\tau_n(h)| = 0.$$

It follows that as $h \to 0$ and $n \to \infty$ in such a way that $t_0 + nh = t$, we have

$$\lim_{n \to \infty} |e_n| = 0,$$

and therefore the method is convergent.

In the case of Euler's method, we have

$$\Phi(t, y, h) = f(t, y), \quad \tau_n(h) = \frac{h}{2} y''(\tau), \quad \tau \in (t_0, T).$$

Therefore, there exists a constant K such that

$$|\tau_n(h)| \leq Kh, \quad 0 < h \leq h_0,$$

for some sufficiently small h_0. We say that Euler's method is *first-order accurate*. More generally, we say that a one-step method has **order of accuracy** p if, for any sufficiently smooth solution $y(t)$, there exists constants K and h_0 such that

$$|\tau_n(h)| \leq Kh^p, \quad 0 < h \leq h_0.$$

Exercise 9.4.7 *Prove that the modified Euler method (9.6) is convergent and second-order accurate.*

As for multistep methods, a consistent multistep method is convergent if and only if it is stable. Because Adams methods are always strongly stable, it follows that all Adams-Moulton predictor-corrector methods are convergent.

9.4.4 Stiff Differential Equations

To this point, we have evaluated the accuracy of numerical methods for initial-value problems in terms of the rate at which the error approaches zero, when the step size h approaches zero. However, this characterization of accuracy is not always informative, because it neglects the fact that the local truncation error of any one-step or multistep method also depends on higher-order derivatives of the solution. In some cases, these derivatives can be quite large in magnitude, even when the solution itself is relatively small, which requires that h be chosen particularly small in order to achieve even reasonable accuracy.

This leads to the concept of a *stiff differential equation*. A differential equation of the form $y' = f(t, y)$ is said to be **stiff** if its exact solution $y(t)$ includes a term that decays exponentially to zero as t increases, but whose derivatives are much greater in magnitude than the term itself. An example of such a term is e^{-ct}, where c is a large, positive constant, because its kth derivative is $c^k e^{-ct}$. Because of the factor of c^k, this derivative decays to zero much more slowly than e^{-ct} as t increases. Because the error includes a term of this form, evaluated at a time less than t, the error can be quite large if h is not chosen sufficiently small to offset this large derivative. Furthermore, the larger c is, the smaller h must be to maintain accuracy.

Example 9.4.6 *Consider the initial value problem*

$$y' = -100y, \quad t > 0, \quad y(0) = 1.$$

The exact solution is $y(t) = e^{-100t}$*, which rapidly decays to zero as* t *increases. If we solve this problem using Euler's method, with step size* $h = 0.1$*, then we have*

$$y_{n+1} = y_n - 100hy_n = -9y_n,$$

which yields the exponentially growing *solution* $y_n = (-9)^n$*. On the other hand, if we choose* $h = 10^{-3}$*, we obtain the computed solution* $y_n = (0.9)^n$*, which is much more accurate, and correctly captures the qualitative behavior of the exact solution, in that it rapidly decays to zero.* □

The ODE in the preceding example is a special case of the **test equation**

$$y' = \lambda y, \quad y(0) = 1, \quad \operatorname{Re} \lambda < 0.$$

The exact solution to this problem is $y(t) = e^{\lambda t}$. However, as λ increases in magnitude, the problem becomes increasingly stiff. By applying a numerical method to this problem, we can determine how small h must be, for a given value of λ, in order to obtain a qualitatively accurate solution.

When applying a one-step method to the test equation, the computed solution has the form

$$y_{n+1} = Q(h\lambda)y_n,$$

where $Q(h\lambda)$ is a polynomial in $h\lambda$ if the method is explicit, and a rational function if it is implicit. This polynomial is meant to approximate $e^{h\lambda}$, since the exact solution satisfies $y(t_{n+1}) = e^{h\lambda}y(t_n)$. However, to obtain a qualitatively correct solution, that decays to zero as t increases, we must choose h so that $|Q(h\lambda)| < 1$.

Example 9.4.7 *Consider the modified Euler method*

$$y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_n + h, y_n + hf(t_n, y_n))].$$

Setting $f(t, y) = \lambda y$ yields the computed solution

$$y_{n+1} = y_n + \frac{h}{2}[\lambda y_n + \lambda(y_n + h\lambda y_n)] = \left(1 + h\lambda + \frac{1}{2}h^2\lambda^2\right) y_n,$$

so $Q(h\lambda) = 1 + h\lambda + \frac{1}{2}(h\lambda)^2$. If we assume λ is real, then in order to satisfy $|Q(h\lambda)| < 1$, we must have $-2 < h\lambda < 0$. It follows that the larger $|\lambda|$ is, the smaller h must be. □

The test equation can also be used to determine how to choose h for a multistep method. The process is similar to the one used to determine whether a multistep method is stable, except that we use $f(t, y) = \lambda y$, rather than $f(t, y) \equiv 0$.

Given a general multistep method of the form

$$\sum_{i=0}^{s} \alpha_i y_{n+1-i} = h \sum_{i=0}^{s} \beta_i f_{n+1-i},$$

we substitute $f_n = \lambda y_n$ and obtain the recurrence relation

$$\sum_{i=0}^{s} (\alpha_i - h\lambda\beta_i) y_{n+1-i} = 0.$$

It follows that the computed solution has the form

$$y_n = \sum_{i=1}^{s} c_i n^{p_i} \mu_i^n,$$

where each μ_i is a root of the **stability polynomial**

$$Q(\mu, h\lambda) = (\alpha_0 - h\lambda\beta_0)\mu^s + (\alpha_1 - h\lambda\beta_1)\mu^{s-1} + \cdots + (\alpha_{s-1} - h\lambda\beta_{s-1})\mu + (\alpha_s - h\lambda\beta_s).$$

The exponents p_i range from 0 to the multiplicity of μ_i minus one, so if the roots are all distinct, all p_i are equal to zero. In order to ensure that the numerical solution y_n decays to zero as n increases, we must have $|\mu_i| < 1$ for $i = 1, 2, \ldots, s$. Otherwise, the solution will either converge to a nonzero value, or grow in magnitude.

Example 9.4.8 *Consider the 3-step Adams-Bashforth method*

$$y_{n+1} = y_n + \frac{h}{12}[23f_n - 16f_{n-1} + 5f_{n-2}].$$

Applying this method to the test equation yields the stability polynomial

$$Q(\mu, h\lambda) = \mu^3 + \left(-1 - \frac{23}{12}h\lambda\right)\mu^2 + \frac{4}{3}h\lambda\mu - \frac{5}{12}h\lambda.$$

Let $\lambda = -100$. *If we choose* $h = 0.1$, *so that* $\lambda h = -10$, *then* $Q(\mu, h\lambda)$ *has a root approximately equal to* -18.884, *so* h *is too large for this method. On the other hand, if we choose* $h = 0.005$, *so that* $h\lambda = -1/2$, *then the largest root of* $Q(\mu, h\lambda)$ *is approximately* -0.924, *so* h *is sufficiently small to produce a qualitatively correct solution.*

Next, we consider the 2-step Adams-Moulton method

$$y_{n+1} = y_n + \frac{h}{12}[5f_{n+1} + 8f_n - f_{n-1}].$$

In this case, we have

$$Q(\mu, h\lambda) = \left(1 - \frac{5}{12}h\lambda\right)\mu^2 + \left(-1 - \frac{2}{3}h\lambda\right)\mu + \frac{1}{12}h\lambda.$$

Setting $h = 0.05$, *so that* $h\lambda = -5$, *the largest root of* $Q(\mu, h\lambda)$ *turns out to be approximately* -0.906, *so a larger step size can safely be chosen for this method.* □

In general, larger step sizes can be chosen for implicit methods than for explicit methods. However, the savings achieved from having to take fewer time steps can be offset by the expense of having to solve a nonlinear equation during every time step.

9.4.4.1 Region of Absolute Stability

The **region of absolute stability** of a one-step method or a multistep method is the region R of the complex plane such that if $h\lambda \in R$, then a solution of the test equation computed using h and λ will decay to zero, as desired. That is, for a one-step method, $|Q(h\lambda)| < 1$ for $h\lambda \in R$, and for a multistep method, the roots $\mu_1, \mu_2, \ldots, \mu_s$ of $Q(\mu, h\lambda)$ satisfy $|\mu_i| < 1$.

Because a larger region of absolute stability allows a larger step size h to be chosen for a given value of λ, it is preferable to use a method that has as large a region of absolute stability as possible. The ideal situation is when a method is **A-stable**, which means that its region of absolute stability contains the entire left half-plane, because then, the solution will decay to zero regardless of the choice of h.

An example of an A-stable one-step method is the *Backward Euler method*

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}),$$

an implicit method. For this method,

$$Q(h\lambda) = \frac{1}{1 - h\lambda},$$

and since $\operatorname{Re}\lambda < 0$, it follows that $|Q(h\lambda)| < 1$ regardless of the value of h. The only A-stable multistep method is the *implicit trapezoidal method*

$$y_{n+1} = y_n + \frac{h}{2}[f_{n+1} + f_n],$$

because

$$Q(\mu, h\lambda) = \left(1 - \frac{h\lambda}{2}\right)\mu + \left(-1 - \frac{h\lambda}{2}\right),$$

which has the root

$$\mu = \frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}}.$$

The numerator and denominator have imaginary parts of the same magnitude, but because $\operatorname{Re}\lambda < 0$, the real part of the denominator has a larger magnitude than that of the numerator, so $|\mu| < 1$, regardless of h.

Implicit multistep methods, such as the implicit trapezoidal method, are often used for stiff differential equations because of their larger regions of absolute stability. However, as the next exercises illustrate, it is important to properly estimate the largest possible value of λ for a given ODE in order to select an h such that $h\lambda$ actually lies within the region of absolute stability.

Exercise 9.4.8 *Form the stability polynomial for the 2-step Adams-Moulton method*

$$y_{n+1} = y_n + \frac{h}{12}[5f_{n+1} + 8f_n - f_{n-1}]. \tag{9.17}$$

Exercise 9.4.9 *Suppose the 2-step Adams-Moulton method (9.17) is applied to the IVP*

$$y' = -2y, \quad y(0) = 1.$$

How small must h be so that a bounded solution can be ensured?

Exercise 9.4.10 *Now, suppose the same Adams-Moulton method is applied to the IVP*

$$y' = -2y + e^{-100t}, \quad y(0) = 1.$$

How does the addition of the source term e^{-100t} affect the choice of h?

Exercise 9.4.11 *In general, for an ODE of the form $y' = f(t, y)$, how should the value of λ be determined for the purpose of choosing an h such that $h\lambda$ lies within the region of absolute stability?*

9.4.4.2 Dahlquist's Theorems

We conclue our discussion of multistep methods with some important results, due to Germund Dahlquist, concerning the consistency, zero-stability, and convergence of multistep methods.

Theorem 9.4.9 (Dahlquist's Equivalence Theorem) *A consistent multistep method with local truncation error $O(h^p)$ is convergent with global error $O(h^p)$ if and only if it is zero-stable.*

This theorem shows that local error provides an indication of global error only for zero-stable methods. A proof can be found in [15, Theorem 6.3.4].

The second theorem imposes a limit on the order of accuracy of zero-stable methods.

Theorem 9.4.10 (Dahlquist's Barrier Theorem) *The order of accuracy of a zero-stable s-step method is at most $s+1$, if s is odd, or $s+2$, if s is even.*

For example, because of this theorem, it can be concluded that a 6th-order accurate three-step method cannot be zero stable, whereas a 4th-order accurate, zero-stable two-step method has the highest order of accuracy that can be achieved. A proof can be found in [15, Section 4.2].

Finally, we state a result, proved in [11], concerning absolute stability that highlights the trade-off between explicit and implicit methods.

Theorem 9.4.11 (Dahlquist's Second Barrier Theorem) *No explicit multistep method is A-stable. Furthermore, no A-stable multistep method can have an order of accuracy greater than 2. The second-order accurate, A-stable multistep method with the smallest asymptotic error constant is the trapezoidal method.*

In order to obtain A-stable methods with higher-order accuracy, it is necessary to relax the condition of A-stability. Backward differentiation formulae (BDF), mentioned previously in our initial discussion of multistep methods, are efficient implicit methods that are high-order accurate and have a region of absolute stability that includes a large portion of the negative half-plane, including the entire negative real axis.

Exercise 9.4.12 *Find a BDF of order greater than 1 that has a region of absolute stability that includes the entire negative real axis.*

9.5 Adaptive Methods

So far, we have assumed that the time-stepping methods that we have been using for solving $y' = f(t, y)$ on the interval $t_0 < t < T$ compute the solution at times $t_1, t_2, \ldots$ that are equally spaced. That is, we define $t_{n+1} - t_n = h$ for some value of h that is *fixed* over the entire interval $[t_0, T]$ on which the problem is being solved. However, in practice, this is ill-advised because

- the chosen time step may be too large to resolve the solution with sufficient accuracy, especially if it is highly oscillatory, or

- the chosen time step may be too small when the solution is particularly smooth, thus wasting computational effort required for evaluations of f.

This is reminiscent of the problem of choosing appropriate subintervals when applying composite quadrature rules to approximate definite integrals. In that case, adaptive quadrature rules were designed to get around this problem. These methods used estimates of the error in order to determine whether certain subintervals should be divided. In this section, we seek to develop an analogous strategy for time-stepping to solve initial value problems.

9.5.1 Error Estimation

The counterpart to this approach for initial value problems would involve estimating the *global* error, perhaps measured by $\max_{0\leq n\leq T/h}|y(t_n)-y_n|$, and then, if it is too large, repeating the time-stepping process with a smaller value of h. However, this is impractical, because it is difficult to obtain a sharp estimate of global error, and much of the work involved would be wasted due to overwriting of solution values, unlike with adaptive quadrature, where each subinterval can be integrated independently.

Instead, we propose to estimate the *local truncation error* at each time step, and use that estimate to determine whether h should be varied for the next time step. This approach minimizes the amount of extra work that is required to implement this kind of *adaptive* time-stepping, and it relies on an error estimate that is easy to compute.

We first consider error estimation for one-step methods. This error estimation is accomplished using a pair of one-step methods,

$$\begin{aligned} y_{n+1} &= y_n + h\Phi_p(t_n, y_n, h), && (9.18)\\ \tilde{y}_{n+1} &= \tilde{y}_n + h\Phi_{p+1}(t_n, \tilde{y}_n, h), && (9.19) \end{aligned}$$

of orders p and $p+1$, respectively. Recall that their local truncation errors are

$$\begin{aligned} \tau_{n+1}(h) &= \frac{1}{h}[y(t_{n+1})-y(t_n)] - \Phi_p(t_n, y(t_n), h),\\ \tilde{\tau}_{n+1}(h) &= \frac{1}{h}[y(t_{n+1})-y(t_n)] - \Phi_{p+1}(t_n, y(t_n), h). \end{aligned}$$

We make the assumption that both methods are exact at time t_n; that is, $y_n = \tilde{y}_n = y(t_n)$. It then follows from (9.18) and (9.19) that

$$\tau_{n+1}(h) = \frac{1}{h}[y(t_{n+1}) - y_{n+1}], \quad \tilde{\tau}_{n+1}(h) = \frac{1}{h}[y(t_{n+1}) - \tilde{y}_{n+1}].$$

Subtracting these equations yields

$$\tau_{n+1}(h) = \tilde{\tau}_{n+1}(h) + \frac{1}{h}[\tilde{y}_{n+1} - y_{n+1}].$$

Because $\tau_{n+1}(h)$ is $O(h^p)$ while $\tilde{\tau}_{n+1}(h)$ is $O(h^{p+1})$, we neglect $\tilde{\tau}_{n+1}(h)$ and obtain the simple error estimate

$$\tau_{n+1}(h) = \frac{1}{h}(\tilde{y}_{n+1} - y_{n+1}).$$

The approach for multistep methods is similar. We use a pair of Adams methods, consisting of an s-step Adams-Bashforth (explicit) method,

$$\sum_{i=0}^{s} \alpha_i y_{n+1-i} = h\sum_{i=1}^{s} \beta_i f_{n+1-i},$$

and an $(s-1)$-step Adams-Moulton (implicit) method,

$$\sum_{i=0}^{s} \tilde{\alpha}_i \tilde{y}_{n+1-i} = h\sum_{i=0}^{s} \tilde{\beta}_i \tilde{f}_{n+1-i},$$

where $\tilde{f}_i = f(t_i, \tilde{y}_i)$, so that both are $O(h^s)$-accurate. We then have

$$\begin{aligned}\sum_{i=0}^{s} \alpha_i y(t_{n+1-i}) &= h\sum_{i=1}^{s} \beta_i f(t_{n+1-i}, y(t_{n+1-i}) + h\tau_{n+1}(h), \\ \sum_{i=0}^{s} \tilde{\alpha}_i y(t_{n+1-i}) &= h\sum_{i=0}^{s} \tilde{\beta}_i f(t_{n+1-i}, y(t_{n+1-i}) + h\tilde{\tau}_{n+1}(h),\end{aligned}$$

where $\tau_{n+1}(h)$ and $\tilde{\tau}_{n+1}(h)$ are the local truncation errors of the explicit and implicit methods, respectively.

As before, we assume that $y_{n+1-s}, \ldots, y_n$ are exact, which yields

$$\tau_{n+1}(h) = \frac{1}{h}[y(t_{n+1}) - y_{n+1}], \quad \tilde{\tau}_{n+1}(h) = \frac{1}{h}[y(t_{n+1}) - \tilde{y}_{n+1}],$$

as in the case of one-step methods. It follows that

$$\tilde{y}_{n+1} - y_{n+1} = h[\tau_{n+1}(h) - \tilde{\tau}_{n+1}(h)].$$

The local truncation errors have the form

$$\tau_{n+1}(h) = Ch^s y^{(s+1)}(\xi_n), \quad \tilde{\tau}_{n+1}(h) = \tilde{C}h^s y^{(s+1)}(\tilde{\xi}_n),$$

where $\xi_n, \tilde{\xi}_n \in [t_{n+1-s}, t_{n+1}]$. We assume that these unknown values are equal, which yields

$$\tilde{\tau}_{n+1}(h) \approx \frac{\tilde{C}}{h(C - \tilde{C})}[\tilde{y}_{n+1} - y_{n+1}]. \tag{9.20}$$

Exercise 9.5.1 *Formulate an error estimate of the form (9.20) for the case $s = 4$; that is, estimate the error in the 3-step Adams-Moulton method (9.12) using the 4-step Adams-Bashforth method (9.11).* Hint: *use the result of Exercise 9.4.3.*

9.5.2 Adaptive Time-Stepping

Our goal is to determine how to modify h so that the local truncation error is approximately *equal* to a prescribed tolerance ε, and therefore is not too large nor too small.

When using two one-step methods as previously discussed, because $\tau_{n+1}(h)$ is the local truncation error of a method that is pth-order accurate, it follows that if we replace h by qh for some scaling factor q, the error is multiplied by q^p. Therefore, we relate the error obtained with step size qh to our tolerance, and obtain

$$|\tau_{n+1}(qh)| \approx \left|\frac{q^p}{h}(\tilde{y}_{n+1} - y_{n+1})\right| \leq \varepsilon.$$

Solving for q yields

$$q \leq \left(\frac{\varepsilon h}{|\tilde{y}_{n+1} - y_{n+1}|}\right)^{1/p}.$$

In practice, though, the step size is kept bounded by chosen values $h_{\min}$ and $h_{\max}$ in order to avoid missing sensitive regions of the solution by using excessively large time steps, as well as expending too much computational effort on regions where $y(t)$ is oscillatory by using step sizes that are too small [10].

For one-step methods, if the error is deemed small enough so that y_{n+1} can be accepted, but $\tilde{y}_{n+1}$ is obtained using a higher-order method, then it makes sense to instead use $\tilde{y}_{n+1}$ as input for the next time step, since it is ostensibly more accurate, even though the error estimate applies to y_{n+1}. Using $\tilde{y}_{n+1}$ instead is called **local extrapolation**.

The **Runge-Kutta-Fehlberg method** [14] is an example of an adaptive time-stepping method. It uses a four-stage, fourth-order Runge-Kutta method and a five-stage, fifth-order Runge-Kutta method. These two methods *share* some evaluations of $f(t, y)$, in order to reduce the number of evaluations of f per time step to six, rather than the nine that would normally be required from a pairing of fourth- and fifth-order methods. A pair of Runge-Kutta methods that can share stages in this way is called an **embedded pair**.

The **Bogacki-Shampine method** [8], which is used in the MATLAB function `ode23`, is an embedded pair consisting of a four-stage, second-order Runge-Kutta method and a three-stage, third-order Runge-Kutta method. As in the Runge-Kutta-Fehlberg metod, evaluations of $f(t, y)$ are shared, reducing the number of evaluations per time step from seven to four. However, unlike the Runge-Kutta Felhberg method, its last stage, $k_4 = f(t_{n+1}, y_{n+1})$, is the same as the first stage of the next time step, $k_1 = f(t_n, y_n)$, if y_{n+1} is accepted, as local extrapolation is used. This reduces the number of *new* evaluations of f per time step from four to three. A Runge-Kutta method that shares stages across time-steps in this manner is called a **FSAL** (First Same as Last) method.

Exercise 9.5.2 *Find the definitions of the two Runge-Kutta methods used in the Bogacki-Shampine method (they can easily be found online). Use these definitions to write a* MATLAB *function* `[t,y]=rk23(f,tspan,y0,h)` *that implements the Bogacki-Shampine method, using an initial step size specified in the input argument* `h`*. How does the performance of your method compare to that of* `ode23`*?*

The MATLAB ODE solver `ode45` uses the **Dormand-Prince method** [13], which consists of a 5-stage, 5th-order Runge-Kutta method and a 6-stage, 4th-order Runge-Kutta method. By sharing stages, the number of evaluations of $f(t, y)$ is reduced from eleven to seven. Like the Bogacki-Shampine method, the Dormand-Prince method is FSAL, so in fact only six new evaluations per time step are required.

Exercise 9.5.3 *Find the definitions of the two Runge-Kutta methods used in the Dormand-Prince method (they can easily be found online). Use these definitions to write a* MATLAB *function* `[t,y]=rk23(f,tspan,y0,h)` *that implements the Dormand-Prince method, using an initial step size specified in the input argument* `h`*. How does the performance of your method compare to that of* `ode45`*?*

For multistep methods, we assume as before that an s-step predictor and $(s-1)$-step corrector are used. Recall that the error estimate $\tau_{n+1}(h)$ for the corrector is given in (9.20). As with one-step methods, we relate the error estimate $\tau_{n+1}(qh)$ to the error tolerance ε and solve for q, which yields

$$q \approx \left(\frac{\varepsilon}{\tau_{n+1}(h)}\right)^{1/s}.$$

Then, the time step can be adjusted to qh, but as with one-step methods, q is constrained to avoid drastic changes in the time step. Unlike one-step methods, a change in the time step is computationally expensive, as it requires the computation of new starting values at equally spaced times.

Exercise 9.5.4 *Implement an adaptive multistep method based on the 4-step Adams Bashforth method (9.11) and 3-step Adams-Moulton method (9.12). Use the fourth-order Runge-Kutta method to obtain starting values.*

9.6 Higher-Order Equations and Systems of Differential Equations

Numerical methods for solving a single, first-order ODE of the form $y' = f(t, y)$ can also be applied to more general ODE, including systems of first-order equations, and equations with higher-order derivatives. We will now learn how to generalize these methods to such problems.

9.6.1 Systems of First-Order Equations

We consider a system of m first-order equations, that has the form

$$\begin{aligned} y_1' &= f_1(t, y_1, y_2, \ldots, y_m), \\ y_2' &= f_2(t, y_1, y_2, \ldots, y_m), \\ &\vdots \\ y_m' &= f_m(t, y_1, y_2, \ldots, y_m), \end{aligned}$$

where $t_0 < t \leq T$, with initial conditions

$$y_1'(t_0) = y_{1,0}, \quad y_2'(t_0) = y_{2,0}, \quad \cdots \quad y_m'(t_0) = y_{m,0}.$$

This problem can be written more conveniently in vector form

$$\mathbf{y}' = \mathbf{f}(y, \mathbf{y}), \quad \mathbf{y}'(t_0) = \mathbf{y}_0,$$

where $\mathbf{y}(t)$ is a vector-valued function with component functions

$$\mathbf{y}(t) = \langle y_1(t), y_2(t), \ldots, y_m(t) \rangle,$$

$\mathbf{f}$ is a vector-valued function of t and $\mathbf{y}$, with component functions

$$\mathbf{f}(t, y) = \langle f_1(t, \mathbf{y}), f_2(t, \mathbf{y}), \ldots, f_m(t, \mathbf{y}),$$

and $\mathbf{y}_0$ is the vector of initial values,

$$\mathbf{y}_0 = \langle y_{1,0}, y_{2,0}, \ldots, y_{m,0} \rangle.$$

This initial-value problem has a unique solution $\mathbf{y}(t)$ on $[t_0, T]$ if $\mathbf{f}$ is continuous on the domain $D = [t_0, T] \times (-\infty, \infty)^m$, and satisfies a Lipschitz condition on D in each of the variables $y_1, y_2, \ldots, y_m$.

Applying a one-step method of the form

$$y_{n+1} = y_n + h\phi(t_n, y_n, h)$$

to a system is straightforward. It simply requires generalizing the function $\phi(t_n, y_n, h)$ to a vector-valued function that evaluates $\mathbf{f}(t, \mathbf{y})$ in the same way as it evaluates $f(t, y)$ in the case of a single equation, with its arguments obtained from t_n and $\mathbf{y}_n$ in the same way as they are from t_n and y_n for a single equation.

Example 9.6.1 *Consider the modified Euler method*

$$y_{n+1} = y_n + \frac{h}{2}\left[f(t_n, y_n) + f\left(t_n + h, y_n + \frac{h}{2}f(t_n, y_n)\right)\right].$$

To apply this method to a system of m *equations of the form* $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$*, we compute*

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2}\left[\mathbf{f}(t_n, \mathbf{y}_n) + \mathbf{f}\left(t_n + h, \mathbf{y}_n + \frac{h}{2}\mathbf{f}(t_n, \mathbf{y}_n)\right)\right],$$

where $\mathbf{y}_n$ *is an approximation to* $\mathbf{y}(t_n)$*. The vector* $\mathbf{y}_n$ *has components*

$$\mathbf{y}_n = \langle y_{1,n}, y_{2,n}, \ldots, y_{m,n}\rangle,$$

where, for $i = 1, 2, \ldots, m$*,* $y_{i,n}$ *is an approximation to* $y_i(t_n)$*.*

We illustrate this method on the system of two equations

$$\begin{aligned} y_1' &= f_1(t, y_1, y_2) = -2y_1 + 3ty_2, && (9.21) \\ y_2' &= f_2(t, y_1, y_2) = t^2 y_1 - e^{-t} y_2. && (9.22) \end{aligned}$$

First, we rewrite the method in the more convenient form

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h, y_n + k_1) \\ y_{n+1} &= y_n + \frac{1}{2}[k_1 + k_2]. \end{aligned}$$

Then, the modified Euler method, applied to this system of ODE, takes the form

$$\begin{aligned} k_{1,1} &= hf_1(t_n, y_{1,n}, y_{2,n}) \\ &= h[-2y_{1,n} + 3t_n y_{2,n}] \\ k_{2,1} &= hf_2(t_n, y_{1,n}, y_{2,n}) \\ &= h[t_n^2 y_{1,n} - e^{-t_n} y_{2,n} \\ k_{1,2} &= hf_1(t_n + h, y_{1,n} + k_{1,1}, y_{2,n} + k_{2,1}) \\ &= h[-2(y_{1,n} + k_{1,1}) + 3(t_n + h)(y_{2,n} + k_{2,2})] \\ k_{2,2} &= hf_2(t_n + h, y_{1,n} + k_{1,1}, y_{2,n} + k_{2,1}) \\ &= h[(t_n + h)^2(y_{1,n} + k_{1,1}) - e^{-(t_n+h)}(y_{2,n} + k_{2,1})] \\ y_{1,n+1} &= y_{1,n} + \frac{1}{2}[k_{1,1} + k_{1,2}] \\ y_{2,n+1} &= y_{2,n} + \frac{1}{2}[k_{2,1} + k_{2,2}]. \end{aligned}$$

This can be written in vector form as follows:

$$\begin{aligned}\mathbf{k}_1 &= h\mathbf{f}(t_n, \mathbf{y}_n)\\ \mathbf{k}_2 &= h\mathbf{f}(t_n + h, \mathbf{y}_n + \mathbf{k}_1)\\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{1}{2}[\mathbf{k}_1 + \mathbf{k}_2].\end{aligned}$$

□

Exercise 9.6.1 *Try your* `rk4` *function from Exercise 9.2.4 on the system (9.21), (9.22) with initial conditions* $y_1(0) = 1$, $y_2(0) = -1$. *Write your time derivative function* `yp=f(t,y)` *for this system so that the input argument* `y` *and the value* `yp` *returned by* `f` *are both* column *vectors, and pass a column vector containing the initial values as the input argument* `y0`. *Do you even need to modify* `rk4`*?*

Multistep methods generalize in a similar way. A general s-step multistep method for a system of first-order ODE $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ has the form

$$\sum_{i=0}^{s} \alpha_i \mathbf{y}_{n+1-i} = h \sum_{i=0}^{s} \beta_i \mathbf{f}(t_{n+1-i}, \mathbf{y}_{n+1-i}),$$

where the constants α_i and β_i, for $i = 0, 1, \ldots, s$, are determined in the same way as in the case of a single equation.

Example 9.6.2 *The explicit 3-step Adams-Bashforth method applied to the system in the previous example has the form*

$$\begin{aligned}y_{1,n+1} &= y_{1,n} + \frac{h}{12}[23f_{1,n} - 16f_{1,n-1} + 5f_{1,n-2}],\\ y_{2,n+1} &= y_{2,n} + \frac{h}{12}[23f_{2,n} - 16f_{2,n-1} + 5f_{2,n-2}],\end{aligned}$$

where

$$f_{1,i} = -2y_{1,i} + 3t_i y_{2,i}, \quad f_{2,i} = t_i^2 y_{1,i} - e^{-t_i} y_{2,i}, \quad i = 0, \ldots, n.$$

□

The order of accuracy for a one-step or multistep method, when applied to a system of equations, is the same as when it is applied to a single equation. For example, the modified Euler's method is second-order accurate for systems, and the 3-step Adams-Bashforth method is third-order accurate. However, when using adaptive step size control for any of these methods, it is essential that the step size h is selected so that *all* components of the solution are sufficiently accurate, or it is likely that *none* of them will be.

9.6.2 Higher-Order Equations

The numerical methods we have learned are equally applicable to differential equations of higher order, that have the form

$$y^{(m)} = f(t, y, y', y'', \dots, y^{(m-1)}), \quad t_0 < t \le T,$$

because such equations are equivalent to systems of first-order equations, in which new variables are introduced that correspond to lower-order derivatives of y. Specifically, we define the variables

$$u_1 = y, \quad u_2 = y', \quad u_3 = y'', \quad \cdots \quad u_m = y^{(m-1)}.$$

Then, the above ODE of order m is equivalent to the system of first-order ODEs

$$\begin{aligned} u_1' &= u_2 \\ u_2' &= u_3 \\ &\vdots \\ u_m' &= f(t, u_1, u_2, \dots, u_m). \end{aligned}$$

The initial conditions of the original higher-order equation,

$$y(t_0) = y_0^{(0)}, \quad y'(t_0) = y_0^{(1)}, \dots, \quad y^{(m-1)}(t_0) = y_0^{(m-1)},$$

are equivalent to the following initial conditions of the first order system

$$u_1(t_0) = y_0^{(0)}, \quad u_2(t_0) = y_0^{(1)}, \dots, u_m(t_0) = y_0^{(m-1)}.$$

We can then apply any one-step or multistep method to this first-order system.

Example 9.6.3 *Consider the second-order equation*

$$y'' + 3y' + 2y = \cos t, \quad y(0) = 2, \quad y'(0) = -1.$$

By defining $u_1 = y$ *and* $u_2 = y'$, *we obtain the equivalent the first-order system*

$$\begin{aligned} u_1' &= u_2 \\ u_2' &= -3u_2 - 2u_1 + \cos t, \end{aligned}$$

with initial conditions

$$u_1(0) = 2, \quad u_2(0) = -1.$$

To apply the 4th-order Runge-Kutta method,

$$\begin{aligned} \mathbf{k}_1 &= h\mathbf{f}(t_n, \mathbf{u}_n) \\ \mathbf{k}_2 &= h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{u}_n + \frac{1}{2}\mathbf{k}_1\right) \\ \mathbf{k}_3 &= h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{u}_n + \frac{1}{2}\mathbf{k}_2\right) \\ \mathbf{k}_4 &= h\mathbf{f}(t_n + h, \mathbf{u}_n + \mathbf{k}_3) \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4), \end{aligned}$$

to this system, we compute

$$\begin{aligned}
k_{1,1} &= hf_1(t_n, u_{1,n}, u_{2,n}) = hu_{2,n} \\
k_{2,1} &= hf_2(t_n, u_{1,n}, u_{2,n}) = h[-3u_{2,n} - 2u_{1,n} + \cos t_n] \\
k_{1,2} &= hf_1\left(t_n + \frac{h}{2}, u_{1,n} + \frac{1}{2}k_{1,1}, u_{2,n} + \frac{1}{2}k_{2,1}\right) = h\left(u_{2,n} + \frac{1}{2}k_{2,1}\right) \\
k_{2,2} &= hf_2\left(t_n + \frac{h}{2}, u_{1,n} + \frac{1}{2}k_{1,1}, u_{2,n} + \frac{1}{2}k_{2,1}\right) \\
&= h\left[-3\left(u_{2,n} + \frac{1}{2}k_{2,1}\right) - 2\left(u_{1,n} + \frac{1}{2}k_{1,1}\right) + \cos\left(t_n + \frac{h}{2}\right)\right] \\
k_{1,3} &= hf_1\left(t_n + \frac{h}{2}, u_{1,n} + \frac{1}{2}k_{1,2}, u_{2,n} + \frac{1}{2}k_{2,2}\right) = h\left(u_{2,n} + \frac{1}{2}k_{2,2}\right) \\
k_{2,3} &= hf_2\left(t_n + \frac{h}{2}, u_{1,n} + \frac{1}{2}k_{1,2}, u_{2,n} + \frac{1}{2}k_{2,2}\right) \\
&= h\left[-3\left(u_{2,n} + \frac{1}{2}k_{2,2}\right) - 2\left(u_{1,n} + \frac{1}{2}k_{1,2}\right) + \cos\left(t_n + \frac{h}{2}\right)\right] \\
k_{1,4} &= hf_1(t_n + h, u_{1,n} + k_{1,3}, u_{2,n} + k_{2,3}) = h(u_{2,n} + k_{2,3}) \\
k_{2,4} &= hf_2(t_n + h, u_{1,n} + k_{1,3}, u_{2,n} + k_{2,3}) \\
&= h[-3(u_{2,n} + k_{2,3}) - 2(u_{1,n} + k_{1,3}) + \cos(t_n + h)] \\
u_{1,n+1} &= u_{1,n} + \frac{1}{6}(k_{1,1} + 2k_{1,2} + 2k_{1,3} + k_{1,4}) \\
u_{2,n+1} &= u_{2,n} + \frac{1}{6}(k_{2,1} + 2k_{2,2} + 2k_{2,3} + k_{2,4}).
\end{aligned}$$

□

Exercise 9.6.2 *Modify your* `rk4` *function from Exercise 9.2.4 so that it solves a single ODE of the form*

$$y^{(m)} = f(t, y, y', y'', \ldots, y^{(m-1)})$$

with initial conditions

$$y(t_0) = y_0, \quad y'(t_0) = y_0', \quad y''(t_0) = y_0'', \quad \cdots \quad y^{(m-1)}(t_0) = y_0^{(m)}.$$

Assume that the input argument `ym = f(t,y)` *treats the input argument* `y` *as a* row *vector consisting of the values of* $y, y', y'', \ldots, y^{(m-1)}$ *at time* `t`*, and that* `f` *returns a* scalar *value* `ym` *that represents the value of* $y^{(m)}$*. Your function should also assume that the argument* `y0` *containing the initial values is a row vector. The value of* m *indicating the order of the ODE can be automatically inferred from* `length(y0)`*, rather than passed as a parameter.*

Exercise 9.6.3 *How would you use your function from the previous exercise to solve a* system *of ODEs of order* m*?*

Chapter 10

Two-Point Boundary Value Problems

We now consider the *two-point boundary value problem* (BVP)

$$y'' = f(x, y, y'), \quad a < x < b, \tag{10.1}$$

a second-order ODE, with *boundary conditions*

$$y(a) = \alpha, \quad y(b) = \beta. \tag{10.2}$$

This problem is guaranteed to have a unique solution if the following conditions hold:

- f, f_y, and $f_{y'}$ are continuous on the domain

$$D = \{(x, y, y') \,|\, a \leq x \leq b, -\infty < y < \infty, -\infty < y' < \infty\}.$$

- $f_y > 0$ on D
- $f_{y'}$ is bounded on D.

In this chapter, we will introduce several methods for solving this kind of problem, most of which can be generalized to *partial differential equations* (PDEs) on higher-dimensional domains. A comprehensive treatment of numerical methods for two-point BVPs can be found in [20].

10.1 The Shooting Method

There are several approaches to solving this type of problem. The first method that we will examine is called the *shooting method*. It treats the two-point boundary value problem as an *initial value problem* (IVP), in which x plays the role of the time variable, with a being the "initial time" and b being the "final time". Specifically, the shooting method solves the initial value problem

$$y'' = f(x, y, y'), \quad a < x < b,$$

with initial conditions

$$y(a) = \alpha, \quad y'(a) = t,$$

where t must be chosen so that the solution satisfies the remaining boundary condition, $y(b) = \beta$. Since t, being the first derivative of $y(x)$ at $x = a$, is the "initial slope" of the solution, this approach requires selecting the proper slope, or “trajectory”, so that the solution will “hit the target” of $y(x) = \beta$ at $x = b$. This viewpoint indicates how the shooting method earned its name. Note that since the ODE associated with the IVP is of second-order, it must normally be rewritten as a system of first-order equations before it can be solved by standard numerical methods such as Runge-Kutta or multistep methods.

10.1.1 Linear Problems

In the case where $y'' = f(x, y, y')$ is a *linear* ODE of the form

$$y'' = p(x)y' + q(x)y + r(x), \quad a < x < b, \tag{10.3}$$

selecting the slope t is relatively simple. Let $y_1(x)$ be the solution of the IVP

$$y'' = p(x)y' + q(x)y + r(x), \quad a < x < b, \quad y(a) = \alpha, \quad y'(a) = 0, \tag{10.4}$$

and let $y_2(x)$ be the solution of the IVP

$$y'' = p(x)y' + q(x)y, \quad a < x < b, \quad y(a) = 0, \quad y'(a) = 1. \tag{10.5}$$

Then, the solution of the original BVP has the form

$$y(x) = y_1(x) + ty_2(x), \tag{10.6}$$

where t is the correct slope, since any linear combination of solutions of the ODE also satisfies the ODE, and the initial values are linearly combined in the same manner as the solutions themselves.

Exercise 10.1.1 *Assume $y_2(b) \neq 0$. Find the value of t in (10.6) such that the boundary conditions (10.2) are satisfied.*

Exercise 10.1.2 *Explain why the condition $y_2(b)$ is guaranteed to be satisfied, due to the previously stated assumptions about $f(x, y, y')$ that guarantee the existence and uniqueness of the solution.*

Exercise 10.1.3 *Write a* MATLAB *function* `y=shootlinear(p,q,r,a,b,alpha,beta,n)` *that solves the linear BVP (10.3), (10.2) using the shooting method. Use the fourth-order Runge-Kutta method to solve the IVPs (10.4), (10.5).* Hint: *consult Section 9.6 on solving second-order ODEs. The input arguments* `p`, `q`, *and* `r` *are function handles for the coefficients $p(x)$, $q(x)$ and $r(x)$, respectively, of (10.3). The input arguments* `a`, `b`, `alpha` *and* `beta` *specify the boundary conditions (10.2), and* `n` *refers to the number of* interior *grid points; that is, a time step of $h = (b - a)/(n + 1)$ is to be used. The output* `y` *is a vector consisting of* $\mathtt{n} + 2$ *values, including both the boundary and interior values of the approximation of the solution $y(x)$ on $[a, b]$. Test your function on the BVP from Example 10.2.1.*

10.1.2 Nonlinear Problems

If the ODE is nonlinear, however, then t satisfies a nonlinear equation of the form

$$y(b, t) = 0,$$

where $y(b,t)$ is the value of the solution, at $x = b$, of the IVP specified by the shooting method, with initial sope t. This nonlinear equation can be solved using an iterative method such as the bisection method, fixed-point iteration, Newton's Method, or the Secant Method. The only difference is that each evaluation of the function $y(b,t)$, at a new value of t, is relatively expensive, since it requires the solution of an IVP over the interval $[a,b]$, for which $y'(a) = t$. The value of that solution at $x = b$ is taken to be the value of $y(b,t)$.

If Newton's Method is used, then an additional complication arises, because it requires the derivative of $y(b,t)$, with respect to t, during each iteration. This can be computed using the fact that $z(x,t) = \partial y(x,t)/\partial t$ satisfies the ODE

$$z'' = f_y z + f_{y'} z', \quad a < x < b, \quad z(a,t) = 0, \quad z'(a,t) = 1,$$

which can be obtained by differentiating the original BVP and its boundary conditions with respect to t. Therefore, each iteration of Newton's Method requires *two* IVPs to be solved, but this extra effort can be offset by the rapid convergence of Newton's Method.

Suppose that Euler's method,

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{f}(x, \mathbf{y}_i, h),$$

for the IVP $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$, $\mathbf{y}(x_0) = \mathbf{y}_0$, is to be used to solve any IVPs arising from the Shooting Method in conjunction with Newton's Method. Because each IVP, for $y(x,t)$ and $z(x,t)$, is of second order, we must rewrite each one as a first-order system. We first define

$$y^1 = y, \quad y^2 = y', \quad z^1 = z, \quad z^2 = z'.$$

We then have the systems

$$\begin{aligned} \frac{\partial y^1}{\partial x} &= y^2, \\ \frac{\partial y^2}{\partial x} &= f(x, y^1, y^2), \\ \frac{\partial z^1}{\partial x} &= z^2, \\ \frac{\partial z^2}{\partial x} &= f_y(x, y^1, y^2) z^1 + f_{y'}(x, y^1, y^2) z^2, \end{aligned}$$

with initial conditions

$$y^1(a) = \alpha, \quad y^2(a) = t, \quad z^1(a) = 0, \quad z^2(a) = 1.$$

The algorithm then proceeds as follows:

```
Choose $t^{(0)}$
Choose $h$ such that $b - a = hN$, where $N$ is the number of steps
for $k = 0, 1, 2, \ldots$ until convergence do
    $i = 0$, $y_0^1 = \alpha$, $y_0^2 = t^{(k)}$, $z_0^1 = 0$, $z_0^2 = 1$
    for $i = 0, 1, 2, \ldots, N - 1$ do
        $x_i = a + ih$
```

```
            $y^1_{i+1} = y^1_i + hy^2_i$
            $y^2_{i+1} = y^2_i + hf(x_i, y^1_i, y^2_i)$
            $z^1_{i+1} = z^1_i + hz^2_i$
            $z^2_{i+1} = z^2_i + h[f_y(x_i, y^1_i, y^2_i)z^1_i + f_{y'}(x_i, y^1_i, y^2_i)z^2_i]$
        end
        $t^{(k+1)} = t^{(k)} - (y^1_N - \beta)/z^1_N$
    end
```

Exercise 10.1.4 *What would be a logical choice of initial guess for the slope* $t^{(0)}$*, that would not require any information about the function* $f(x, y, y')$*?*

Exercise 10.1.5 *Implement the above algorithm to solve the BVP from Example 10.2.2.*

Changing the implementation to use a different IVP solver, such as a Runge-Kutta method or multistep method, in place of Euler's method only changes the inner loop.

Exercise 10.1.6 *Modify your code from Exercise 10.1.5 to use the fourth-order Runge-Kutta method in place of Euler's method. How does this affect the convergence of the Newton iteration?*

Exercise 10.1.7 *Modify your code from Exercise 10.1.5 to use the Secant Method instead of Newton's Method. How can the function* $f(x, y, y')$ *from the ODE be used to obtain a logical second initial guess* $t^{(1)}$*?* Hint: *consider a solution that is a parabola. How is the efficiency of the iteration affected by the change to the Secant Method?*

Exercise 10.1.8 *Write a* MATLAB *function* `SHOOTBVP(f,a,b,alpha,beta,N)` *that solves the general nonlinear BVP (10.1), (10.2) using the shooting method in conjunction with the Secant Method. The input arguments* `f` *is a function handle for the functions* f*. Test your function on the BVP from Exercise 10.1.5. What happens to the performance as* `N`*, the number of time steps, increases?*

10.2 Finite Difference Methods

The shooting method for a two-point boundary value problem of the form

$$y'' = f(x, y, y'), \quad a < x < b, \quad y(a) = \alpha, \quad y(b) = \beta,$$

while taking advantage of effective methods for initial value problems, can not readily be generalized to boundary value problems in higher spatial dimensions. We therefore consider an alternative approach, in which the first and second derivatives of the solution $y(x)$ are approximated by *finite differences*.

We discretize the problem by dividing the interval $[a, b]$ into $N + 1$ subintervals of equal width $h = (b-a)/(N+1)$. Each subinterval is of the form $[x_{i-1}, x_i]$, for $i = 1, 2, \ldots, N$, where $x_i = a+ih$. We denote by y_i an approximation of the solution at x_i; that is, $y_i \approx y(x_i)$. Then, assuming $y(x)$ is at least four times continuously differentiable, we approximate y' and y'' at each x_i, $i = 1, 2, \ldots, N$, by the finite differences

$$y'(x_i) = \frac{y(x_{i+1}) - y(x_i)}{2h} - \frac{h^2}{6}y'''(\eta_i),$$

$$y''(x_i) = \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1})}{h^2} - \frac{h^2}{12}y^{(4)}(\xi_i),$$

where η_i and ξ_i lie in $[x_{i-1}, x_{i+1}]$.

If we substitute these finite differences into the boundary value problem, and apply the boundary conditions to impose

$$y_0 = \alpha, \quad y_{N+1} = \beta,$$

then we obtain a system of equations

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = f\left(x_i, y_i, \frac{y_{i+1} - y_{i-1}}{2h}\right), \quad i = 1, 2, \dots, N,$$

for the values of the solution at each x_i, in which the local truncation error is $O(h^2)$.

10.2.1 Linear Problems

We first consider the case in which the boundary value problem includes a linear ODE of the form

$$y'' = p(x)y' + q(x)y + r(x). \tag{10.7}$$

Then, the above system of equations is also linear, and can therefore be expressed in matrix-vector form

$$A\mathbf{y} = \mathbf{r},$$

where A is a tridiagonal matrix, since the approximations of y' and y'' at x_i only use y_{i-1}, y_i and y_{i+1}, and $\mathbf{r}$ is a vector that includes the values of $r(x)$ at the grid points, as well as additional terms that account for the boundary conditions.

Specifically,

$$\begin{aligned}
a_{ii} &= 2 + h^2 q(x_i), \quad i = 1, 2, \dots, N, \\
a_{i,i+1} &= -1 + \frac{h}{2}p(x_i), \quad i = 1, 2, \dots, N-1, \\
a_{i+1,i} &= -1 - \frac{h}{2}p(x_{i+1}), \quad i = 1, 2, \dots, N-1, \\
r_1 &= -h^2 r(x_1) + \left(1 + \frac{h}{2}p(x_1)\right)\alpha, \\
r_i &= -h^2 r(x_i), \quad i = 2, 3, \dots, N-1, \\
r_N &= -h^2 r(x_N) + \left(1 - \frac{h}{2}p(x_N)\right)\beta.
\end{aligned}$$

This system of equations is guaranteed to have a unique solution if A is diagonally dominaint, which is the case if $q(x) \geq 0$ and $h < 2/L$, where L is an upper bound on $|p(x)|$.

Example 10.2.1 *We solve the BVP*

$$y'' = 2y' - y + xe^x - x, \quad 0 < x < 2, \quad y(0) = 0, \quad y(2) = -4. \tag{10.8}$$

The following script uses the function `FDBVP` *(see Exercise 10.2.1) to solve this problem with* $N = 10$ *interior grid points, and then visualize the exact and approximate solutions, as well as the error.*

```
% coefficients
p=@(x)(2*ones(size(x)));
q=@(x)(-ones(size(x)));
r=@(x)(x.*exp(x)-x);
% boundary conditions
a=0;
b=2;
alpha=0;
beta=-4;
% number of interior grid points
N=10;
% solve using finite differences
[x,y]=FDBVP(p,q,r,a,b,alpha,beta,N);
% exact solution: y = x^3 e^x/6 - 5xe^x/3 + 2e^x - x - 2
yexact=x.^3.*exp(x)/6-5*x.*exp(x)/3+2*exp(x)-x-2;
% plot exact and approximate solutions for comparison
subplot(121)
plot(x,yexact,'b-')
hold on
plot(x,y,'r--o')
hold off
xlabel('x')
ylabel('y')
subplot(122)
plot(x,abs(yexact-y))
xlabel('x')
ylabel('error')
```

The plots are shown in Figure 10.1. □

Exercise 10.2.1 *Write a* MATLAB *function* `FDBVP(p,q,r,a,b,alpha,beta,N)` *that solves the linear BVP (10.7), (10.2). The input arguments* `p`, `q` *and* `r` *must be function handles that represent the functions $p(x)$, $q(x)$ and $r(x)$, respectively. Test your function on the BVP from Example 10.2.1 for different values of N. How does the error behave as N increases? Specifically, if the error is $O(h^p)$, then what is the value of p? Does this value match the theoretical expectation?* Hint: *use the* `diag` *function to set up the matrix A.*

Exercise 10.2.2 *After evaluating the coefficients $p(x)$, $q(x)$ and $r(x)$ from (10.7) at the grid points x_i, $i = 1, 2, \ldots, N$, how many floating-point operations are necessary to solve the system $A\mathbf{y} = \mathbf{r}$? If the boundary conditions are changed but the ODE remains the same, how many additional floating-point operations are needed?* Hint: *review the material in Chapter 2 on the solution of banded systems.*

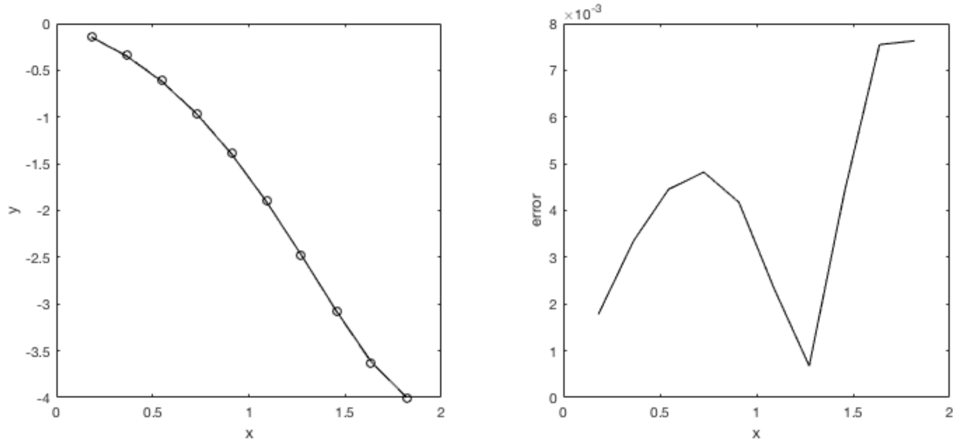


Figure 10.1: Left plot: exact (solid curve) and approximate (dashed curve with circles) solutions of the BVP (10.8) computed using finite differences. Right plot: error in the approximate solution.

10.2.2 Nonlinear Problems

If the ODE is nonlinear, then we must solve a system of nonlinear equations of the form

$$\mathbf{F}(\mathbf{y}) = \mathbf{0},$$

where $\mathbf{F}(\mathbf{y})$ is a vector-valued function with coordinate functions $f_i(\mathbf{y})$, for $i = 1, 2, \ldots, N$. These coordinate functions are defined as follows:

$$\begin{aligned}
F_1(\mathbf{y}) &= y_2 - 2y_1 + \alpha - h^2 f\left(x_1, y_1, \frac{y_2 - \alpha}{2h}\right), \\
F_2(\mathbf{y}) &= y_3 - 2y_2 + y_1 - h^2 f\left(x_2, y_2, \frac{y_3 - y_1}{2h}\right), \\
&\vdots \\
F_{N-1}(\mathbf{y}) &= y_N - 2y_{N-1} + y_{N-2} - h^2 f\left(x_{N-1}, y_{N-1}, \frac{y_N - y_{N-2}}{2h}\right), \\
F_N(\mathbf{y}) &= \beta - 2y_N + y_{N-1} - h^2 f\left(x_N, y_N, \frac{\beta - y_{N-1}}{2h}\right).
\end{aligned} \tag{10.9}$$

This system of equations can be solved approximately using an iterative method such as Fixed-point Iteration, Newton's Method, or the Secant Method.

For example, if Newton's Method is used, then, by the Chain Rule, the entries of the Jacobian matrix $J_{\mathbf{F}}(\mathbf{y})$, a tridiagonal matrix, are defined as follows:

$$\begin{aligned} J_{\mathbf{F}}(\mathbf{y})_{ii} &= \frac{\partial f_i}{\partial y_i}(\mathbf{y}) = -2 - h^2 f_y\left(x_i, y_i, \frac{y_{i+1}-y_{i-1}}{2h}\right), \quad i = 1, 2, \ldots, N, \\ J_{\mathbf{F}}(\mathbf{y})_{i,i+1} &= \frac{\partial f_i}{\partial y_{i+1}}(\mathbf{y}) = 1 - \frac{h}{2} f_{y'}\left(x_i, y_i, \frac{y_{i+1}-y_{i-1}}{2h}\right), \quad i = 1, 2, \ldots, N-1, \\ J_{\mathbf{F}}(\mathbf{y})_{i,i-1} &= \frac{\partial f_i}{\partial y_{i-1}}(\mathbf{y}) = 1 + \frac{h}{2} f_{y'}\left(x_i, y_i, \frac{y_{i+1}-y_{i-1}}{2h}\right), \quad i = 2, 3, \ldots, N, \end{aligned} \qquad (10.10)$$

where, for convenience, we use $y_0 = \alpha$ and $y_{N+1} = \beta$. Then, during each iteration of Newton's Method, the system of equations

$$J_{\mathbf{F}}(\mathbf{y}^{(k)})\mathbf{s}_{k+1} = -\mathbf{F}(\mathbf{y}^{(k)})$$

is solved in order to obtain the next iterate

$$\mathbf{y}^{(k+1)} = \mathbf{y}^{(k)} + \mathbf{s}_{k+1}$$

from the previous iterate. An appropriate initial guess is the unique linear function that satisfies the boundary conditions,

$$\mathbf{y}^{(0)} = \alpha + \frac{\beta - \alpha}{b - a}(\mathbf{x} - a),$$

where $\mathbf{x}$ is the vector with coordinates $x_1, x_2, \ldots, x_N$.

Exercise 10.2.3 *Derive the formulas (10.10) from (10.9).*

Example 10.2.2 *We solve the BVP*

$$y'' = y^3 - yy', \quad 1 < x < 2, \quad y(1) = \frac{1}{2}, \quad y(2) = \frac{1}{3}. \qquad (10.11)$$

This BVP has the exact solution $y(x) = 1/(1+x)$. To solve this problem using finite differences, we apply Newton's Method to solve the equation $\mathbf{F}(\mathbf{y}) = \mathbf{0}$, where

$$\begin{aligned} F_1(\mathbf{y}) &= y_2 - 2y_1 + \frac{1}{2} - h^2\left(y_1^3 - y_1\frac{y_2 - 1/2}{2h}\right), \\ F_i(\mathbf{y}) &= y_{i+1} - 2y_i + y_{i-1} - h^2\left(y_i^3 - y_i\frac{y_{i+1}-y_{i-1}}{2h}\right), \quad i = 2, 3, \ldots, N-1, \\ F_N(\mathbf{y}) &= \frac{1}{3} - 2y_N + y_{N-1} - h^2\left(y_N^3 - y_N\frac{1/3 - y_{N-1}}{2h}\right). \end{aligned}$$

The following MATLAB *function can be used to evaluate $\mathbf{F}(\mathbf{y})$ for a general BVP of the form (10.1). Its arguments are assumed to be vectors of x- and y-values,* including *boundary values, along with a function handle for the right-hand side $f(x, y, y')$ of the ODE (10.1) and the spacing h.*

```
% newtF: evaluates F(y) for solving ODE
% y'' = f(x,y,y') with Newton's Method
function F=newtF(x,y,f,h)
```

```
% use only interior x-values
xi=x(2:end-1);
% y_i
yi=y(2:end-1);
% y_{i+1}
yip1=y(3:end);
% y_{i-1}
yim1=y(1:end-2);
% centered difference approximation of y':
% (y_{i+1} - y_{i-1})/(2h)
ypi=(yip1-yim1)/(2*h);
% evaluate F(y)
F=yip1-2*yi+yim1-h^2*f(xi,yi,ypi);
```

Using $f_y = 3y^2 - y'$ and $f_{y'} = -y$, we obtain

$$\begin{aligned}
J_{\mathbf{F}}(\mathbf{y})_{ii} &= -2 - h^2\left(3y_i^2 - \frac{y_{i+1} - y_{i-1}}{2h}\right), \quad i = 1, 2, \ldots, N, \\
J_{\mathbf{F}}(\mathbf{y})_{i,i+1} &= 1 + \frac{h}{2}y_i, \quad i = 1, 2, \ldots, N-1, \\
J_{\mathbf{F}}(\mathbf{y})_{i,i-1} &= 1 - \frac{h}{2}y_i, \quad i = 2, 3, \ldots, N.
\end{aligned}$$

A MATLAB function similar to `newtF` can be written to construct $J_{\mathbf{F}}(\mathbf{y})$ for a general ODE of the form (10.1). This is left to Exercise 10.2.4.

The following script sets up this BVP, calls `FDNLBVP` (see Exercise 10.2.5 to compute an approximate solution, and then visualizes the approximate solution and exact solution.

```
% set up BVP y'' = f(x,y,y')
f=@(x,y,yp)(y.^3-y.*yp);
fy=@(x,y,yp)(3*y.^2-yp);
fyp=@(x,y,yp)(-y);
% boundary conditons: y(a)=alpha, y(b)=beta
a=1;
b=2;
alpha=1/2;
beta=1/3;
% N: number of interior nodes
N=10;
% use Newton's method
[x,y]=FDNLBVP(f,fy,fyp,a,b,alpha,beta,N);
% compare to exact solution
yexact=1./(x+1);
plot(x,yexact,'b-')
hold on
plot(x,y,'r--o')
hold off
```

```
xlabel('x')
ylabel('y')
```

Using an absolute error tolerance of 10^{-8}, Newton's Method converges in just three iterations, and does so quadratically. The resulting plot is shown in Figure 10.2. □

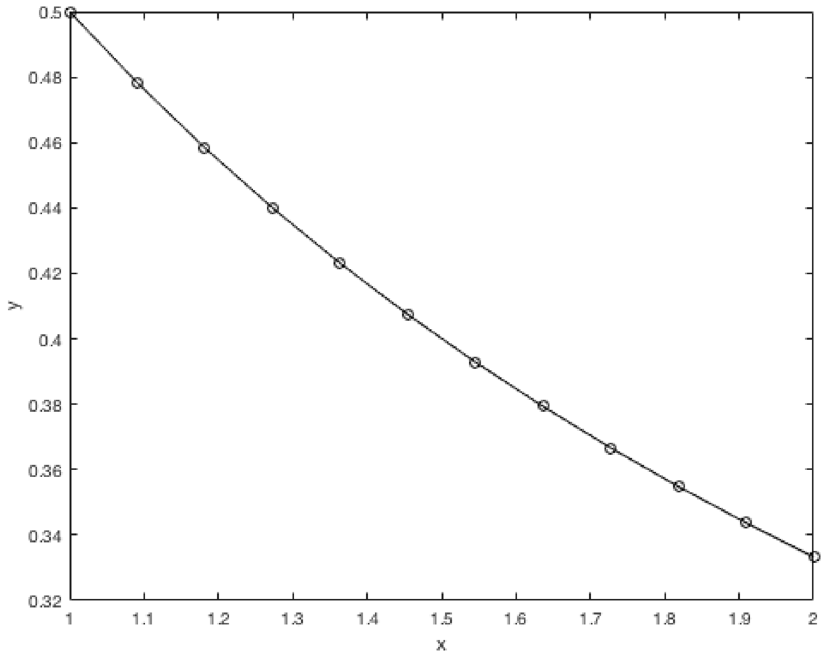


Figure 10.2: Exact (solid curve) and approximate (dashed curve with circles) solutions of the BVP (10.11) from Example 10.2.2.

Exercise 10.2.4 *Write a* Matlab *function* `newtJ(x,y,fy,fyp,h)` *that uses (10.10) to construct the Jacobian matrix* $J_{\mathbf{F}}(\mathbf{y})$ *for use with Newton's Method. The input arguments* `x` *and* `y` *contain x- and y-values, respectively, including boundary values. The input arguments* `fy` *and* `fyp` *are assumed to be function handles that implement* $f_y(x, y, y')$ *and* $f_{y'}(x, y, y')$*, respectively. Use* `newtF` *from Example 10.2.2 as a model.*

Exercise 10.2.5 *Write a* Matlab *function* `FDNLBVP(f,fy,fyp,a,b,alpha,beta,N)` *that solves the general nonlinear BVP (10.1), (10.2) using finite differences in conjunction with Newton's Method. The input arguments* `f`*,* `fy` *and* `fyp` *are function handles for the functions* f*,* f_y *and* $f_{y'}$*, respectively. Use* `newtF` *from Example 10.2.2 and* `newtJ` *from Exercise 10.2.4 as helper functions. Test your function on the BVP from Example 10.2.2. What happens to the error as* `N`*, the number of interior grid points, increases?*

It is worth noting that for two-point boundary value problems that are discretized by finite differences, it is much more practical to use Newton's Method, as opposed to a quasi-Newton

Method such as the Secant Method or Broyden's Method, than for a general system of nonlinear equations because the Jacobian matrix is tridiagaonal. This reduces the expense of the computation of $\mathbf{s}_{k+1}$ from $O(N^3)$ operations in the general case to only $O(N)$ for two-point boundary value problems.

Exercise 10.2.6 *Modify your function* `FDNLBVP` *from Exercise 10.2.5 to use Broyden's Method instead of Newton's Method. How does this affect the efficiency, when applied to the BVP from Example 10.2.2?*

Exercise 10.2.7 *Although Newton's Method is much more efficient for such a problem than for a general system of nonlinear equations, what is an advantage of using the Secant Method over Newton's Method or Broyden's Method?*

It can be shown that regardless of the choice of iterative method used to solve the system of equations arising from discretization, the local truncation error of the finite difference method for nonlinear problems is $O(h^2)$, as in the linear case. The order of accuracy can be increased by applying Richardson extrapolation.

10.3 Collocation

While the finite-difference approach from the previous section is generally effective for two-point boundary value problems, and is more flexible than the Shooting Method as it can be applied to higher-dimensional BVPs, it does have its drawbacks.

- First, the accuracy of finite difference approximations relies on the existence of the higher-order derivatives that appear in their error formulas. Unfortunately, the existence of these higher-order derivatives is not assured.
- Second, a matrix obtained from a finite-difference approximation can be ill-conditioned, and this conditioning worsens as the spacing h decreases.
- Third, it is best suited for problems in which the domain is relatively simple, such as a rectangular domain.

We now consider an alternative approach that, in higher dimensions, is more readily applied to problems on domains with complicated geometries.

First, we assume that the solution $y(x)$ is approximated by a function $y_N(x)$ that is a *linear combination* of chosen linearly independent functions $\phi_1(x), \phi_2(x), \ldots, \phi_N(x)$, called *basis functions* as they form a basis for an N-dimensional vector space. We then have

$$y_N(x) = \sum_{i=1}^{N} c_i\phi_i(x), \tag{10.12}$$

where the constants $c_1, c_2, \ldots, c_N$ are unknown. Substituting this form of the solution into (10.1), (10.2) yields the equations

$$\sum_{j=1}^{N} c_j\phi_j''(x) = f\left(x, \sum_{j=1}^{N} c_j\phi_j(x), \sum_{j=1}^{N} c_j\phi_j'(x)\right), \quad a < x < b, \tag{10.13}$$

$$\sum_{j=1}^{N} c_j\phi_j(a) = \alpha, \quad \sum_{j=1}^{N} c_j\phi_i(b) = \beta. \tag{10.14}$$

Already, the convenience of this assumption is apparent: instead of solving for a function $y(x)$ on the interval (a, b), we are instead having to solve for the N coefficients $c_1, c_2, \ldots, c_N$. However, it is not realistic to think that there is any choice of these coefficients that satisfies (10.13) on the *entire* interval (a, b), as well as the boundary conditions (10.14). Rather, we need to impose N conditions on these N unknowns, in the hope that the resulting system of N equations will have a unique solution that is also an accurate approximation of the exact solution $y(x)$.

To that end, we require that (10.13) is satisfied at $N-2$ points in (a, b), denoted by $x_1, x_2, \ldots, x_{N-2}$, and that the boundary conditions (10.14) are satisfied. The points $a = x_0, x_1, x_2, \ldots, x_{N-2}, x_{N-1} = b$ are called *collocation points*. This approach of approximating $y(x)$ by imposing (10.12) and solving the system of N equations given by

$$\sum_{j=1}^{N} c_j\phi_j''(x_i) = f\left(x_i, \sum_{j=1}^{N} c_j\phi_j(x_i), \sum_{j=1}^{N} c_j\phi_j'(x_i)\right), \quad i = 1, 2, \ldots, N-2 \tag{10.15}$$

and (10.14), is called *collocation.*

For simplicity, we assume that the BVP (10.1) is linear. We are then solving a problem of the form

$$y''(x) = p(x)y'(x) + q(x)y(x) + r(x), \quad a < x < b. \tag{10.16}$$

From (10.15), we obtain the system of linear equations

$$\sum_{j=1}^{N} c_j\phi_j''(x_i) = r(x_i) + \sum_{j=1}^{N} c_j q(x_i)\phi_j(x_i) + \sum_{j=1}^{N} c_j p(x_i)\phi_j'(x_i), \quad i = 1, 2, \ldots, N-2, \tag{10.17}$$

along with (10.14). This system can be written in the form $A\mathbf{c} = \mathbf{b}$, where $\mathbf{c} = \begin{bmatrix} c_1 & \cdots & c_N \end{bmatrix}^T$. We can then solve this system using any of the methods from Chapter 2.

Exercise 10.3.1 *Express the system of linear equations (10.17), (10.14) in the form* $A\mathbf{c} = \mathbf{b}$*, where* $\mathbf{c}$ *is defined as above. What are the entries* a_{ij} *and* b_i *of the matrix* A *and right-hand side vector* $\mathbf{b}$*, respectively?*

Example 10.3.1 *Consider the BVP*

$$y'' = x^2, \quad 0 < x < 1, \tag{10.18}$$

$$y(0) = 0, \quad y(0) = 1. \tag{10.19}$$

We assume that our approximation of $y(x)$ *has the form*

$$y_4(x) = c_1 + c_2x + c_3x^2 + c_4x^3.$$

That is, $N = 4$*, since we are assuming that* $y(x)$ *is a linear combination of the four functions* $1, x, x^2$ *and* x^3*. Substituting this form into the BVP yields*

$$2c_3 + 6c_4x_i = x_i^2, \quad i = 1, 2,$$

$$c_1 = 0, \quad c_1 + c_2 + c_3 + c_4 = 1.$$

Writing this system of equations in matrix-vector form, we obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 6x_1 \\ 0 & 0 & 2 & 6x_2 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 0 \\ x_1^2 \\ x_2^2 \\ 1 \end{bmatrix}. \tag{10.20}$$

For the system to be specified completely, we need to choose the two collocation points $x_1, x_2 \in (0, 1)$. As long as these points are chosen to be distinct, the matrix of the system will be nonsingular. For this example, we choose $x_1 = 1/3$ and $x_2 = 2/3$. We then have the system

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 1/9 \\ 4/9 \\ 1 \end{bmatrix}. \tag{10.21}$$

We can now solve this system in Matlab*:*

```
>> x=[ 1/3 2/3 ];
>> A=[ 1 0 0 0;
       0 0 2 6*x(1);
       0 0 2 6*x(2);
       1 1 1 1 ];
>> b=[ 0; x(1)^2; x(2)^2; 1 ];
>> format rat
>> c=A\b

c =

        0
       17/18
       -1/9
        1/6
```

The `format rat` statement was used to obtain exact values of the entries of **c**, since these entries are guaranteed to be rational numbers in this case. It follows that our approximate solution $y_N(x)$ is

$$y_4(x) = \frac{17}{18}x - \frac{1}{9}x^2 + \frac{1}{6}x^3.$$

The exact solution of the original BVP is easily obtained by integration:

$$y(x) = \frac{1}{12}x^4 + \frac{11}{12}x.$$

From these formulas, though, it is not easy to gauge how accurate $y_4(x)$ is. To visualize the error, we plot both solutions:

```
>> xp=0:0.01:1;
>> y4p=c(1)+c(2)*xp+c(3)*xp.^2+c(4)*xp.^3;
>> yp=xp.^4/12+11*xp/12;
>> plot(xp,yp)
>> hold on
>> plot(xp,y4p,'r--')
>> xlabel('x')
>> ylabel('y')
>> legend('exact','approximate')
```

The result is shown in Figure 10.3. As we can see, this approximate solution is reasonably accurate.

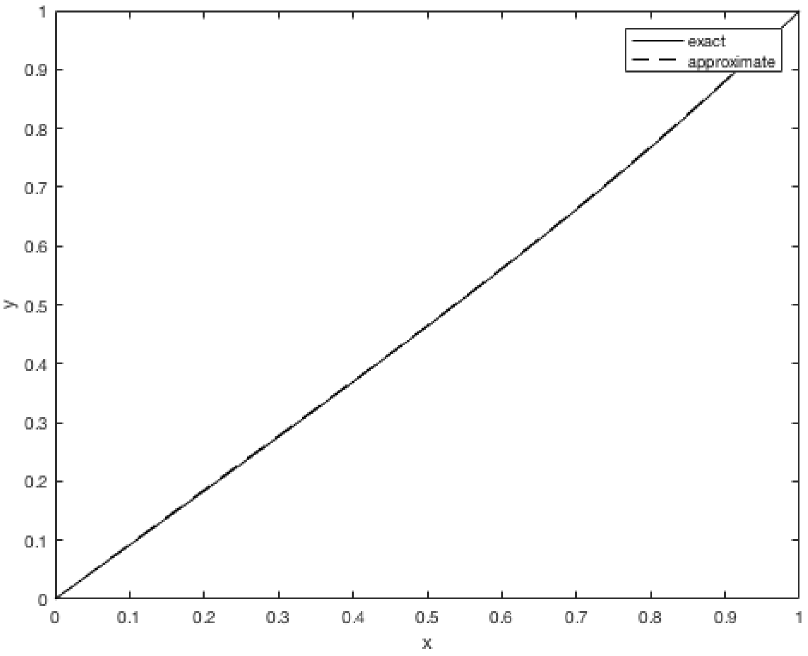


Figure 10.3: Exact (blue curve) and approximate (dashed curve) solutions of (10.18), (10.19) from Example 10.3.1.

To get a numerical indication of the accuracy, we can measure the error at the points in `xp` that were used for plotting:

```
>> format short
>> norm(yp-y4p,'inf')

ans =

    0.0023
```

Since the exact solution and approximation are both polynomials, we can also compute the L^2 norm of the error:

```
>> py=[ 1/12 0 0 11/12 0 ];
>> py4=c(end:-1:1)';
>> err4=py-[ 0 py4 ];
>> err42=conv(err4,err4);
>> Ierr42=polyint(err42);
>> norm22=polyval(Ierr42,1)-polyval(Ierr42,0);
>> norm2=sqrt(norm22)

norm2 =

    0.0019
```

□

Exercise 10.3.2 *Solve the BVP from Example 10.3.1 again, but with different collocation points* $x_1, x_2 \in (0, 1)$*. What happens to the error?*

Exercise 10.3.3 *Use* Matlab *to compute the* relative *error in the* ∞*-norm and* L^2*-norm from the preceding example.*

Exercise 10.3.4 *What would happen if* $N = 5$ *collocation points were used, along with the functions* $\phi_j(x) = x^{j-1}$*,* $j = 1, 2, \ldots, 5$*?*

Exercise 10.3.5 *Write a* Matlab *function* `[x,y]=linearcollocation(p,q,r,a,b,alpha,beta,N)` *that uses collocation to solve the linear BVP (10.16), (10.2). The input arguments* `p`*,* `q` *and* `r` *must be function handles for the functions* $p(x)$*,* $q(x)$ *and* $r(x)$*, respectively, from (10.16). Use* N *equally spaced collocation points, which must be returned in the output* `x`*. The output* `y` *must contain the values of the approximate solution* $y_N(x)$ *at the collocation points. Use the basis functions* $\phi_j(x) = x^{j-1}$*,* $j = 1, 2, \ldots, N$*. Test your function on the BVP from Example 10.3.1.*

Exercise 10.3.6 *Use your function* `linearcollocation` *from Exercise 10.3.5 to solve the BVP*

$$y'' = e^x, \quad 0 < x < 1, \quad y(0) = 0, \quad y(1) = 1.$$

What happens to the error in the approximate solution as the number of collocation points, N*, increases? Plot the error as a function of* N*, using logarithmic scales.*

The choice of functions $\phi_j(x)$, $j = 1, 2, \ldots, N$, can significantly affect the process of solving the resulting system of equations. The choice used in Example 10.3.1, $\phi_j(x) = x^{j-1}$, while convenient, is not a good choice, especially when N is large. As illustrated in Section 6.2, these functions can be nearly linearly dependent on the interval $[a, b]$, which can lead to ill-conditioned systems.

Exercise 10.3.7 *What happens to the condition number of the matrix used by your function* `linearcollocation` *from Exercise 10.3.5 as* N *increases?*

Alternatives include orthogonal polynomials, such as Chebyshev polynomials, or trigonometric polynomials, as discussed in Section 6.4.

Exercise 10.3.8 *Modify your function* `linearcollocation` *from Exercise 10.3.5 to use Chebyshev polynomials instead of the monomial basis, and the Chebyshev points as the collocation points instead of equally spaced points. What happens to the condition number of the matrix as N increases?*

Collocation can be used for either linear or nonlinear BVP. In the nonlinear case, choosing the functions $\phi_j(x)$, $j = 1, 2, \ldots, N$, and the collocation points x_i, $i = 0, 1, \ldots, N-1,$, yields a system of nonlinear equations for the unknowns $c_1, c_2, \ldots, c_N$. This system can then be solved using any of the techniques from Section 8.6, just as when using finite differences.

Exercise 10.3.9 *Describe the system of nonlinear equations $\mathbf{F}(\mathbf{c}) = \mathbf{0}$ that must be solved at each iteration when using Newton's method to solve a general nonlinear BVP of the form (10.1), (10.2). What is the Jacobian of $\mathbf{F}$, $J_{\mathbf{F}}(\mathbf{c})$?*

Exercise 10.3.10 *Write a* MATLAB *function* `[x,y]=nonlinearcollocation(f,a,b,alpha,beta,N)` *that solves a BVP of the form (10.1), (10.2) using Newton's method. The input argument* `f` *must be a function handle for the function $f(x, y, y')$ from (10.1), and* `N` *indicates the number of collocation points. Use the Chebyshev points as the collocation points, and the Chebyshev polynomials as the basis functions. For the initial guess, use the coefficients corresponding to the unique linear function that satisfies the boundary conditions (10.2). Test your function on the BVP*

$$y'' = y^2, \quad y(1) = 6, \quad y(2) = 3/2.$$

What is the exact solution? Hint: *Solve the simpler ODE $y' = y^2$; the form of its solution suggests the form of the solution of $y'' = y^2$.*

10.4 The Finite Element Method

In collocation, the approximate solution $y_N(x)$ is defined to be an element of an N-dimensional function space, which is the span of the chosen basis functions $\phi_j(x)$, $j = 1, 2, \ldots, N$. In this section, we describe another method for solving a BVP in which the approximate solution is again restricted to an N-dimensional function space, but instead of requiring the *residual* $R(x, y_N, y_N', y_N'') \equiv y_N' - f(x, y_N, y_N')$ to vanish at selected points in (a, b), as in collocation, we require that the residual is orthogonal to a given function space, consisting of functions called *test functions*. That is, we require the residual to be zero in an "average" sense, rather than a pointwise sense. In fact, this approach is called the *weighted mean residual method*.

For concreteness and simplicity, we consider the linear boundary value problem

$$-u''(x) = f(x), \quad 0 < x < 1, \tag{10.22}$$

with boundary conditions

$$u(0) = 0, \quad u(1) = 0. \tag{10.23}$$

This equation can be used to model, for example, transverse vibration of a string due to an external force $f(x)$, or longitudinal displacement of a beam subject to a load $f(x)$. In either case, the boundary conditions prescribe that the endpoints of the object in question are fixed.

If we multiply both sides of (10.22) by a *test function* $w(x)$, and then integrate over the domain $[0, 1]$, we obtain

$$\int_0^1 -w(x)u''(x)\,dx = \int_0^1 w(x)f(x)\,dx.$$

Applying integration by parts, we obtain

$$\int_0^1 w(x)u''(x)\,dx = w(x)u'(x)\big|_0^1 - \int_0^1 w'(x)u'(x)\,dx.$$

Let $C^2[0, 1]$ be the space of all functions with two continuous derivatives on $[0, 1]$, and let $C_0^2[0, 1]$ be the space of all functions in $C^2[0, 1]$ that are equal to zero at the endpoints $x = 0$ and $x = 1$. If we require that our test function $u(x)$ belongs to $C_0^2[0, 1]$, then $w(0) = w(1) = 0$, and the boundary term in the above application of integration by parts vanishes. We then have

$$\int_0^1 w'(x)u'(x)\,dx = \int_0^1 w(x)f(x)\,dx.$$

This is called the *weak form* of the boundary value problem (10.22), (10.23), known as the *strong form* or *classical form*, because it only requires that the *first* derivative of $u(x)$ exist, as opposed to the original boundary value problem, that requires the existence of the *second* derivative. The weak form also known as the *variational form*. It can be shown that both the weak form and strong form have the same solution $u \in C_0^2[0, 1]$.

To find an approximate solution of the weak form, we restrict ourselves to a N-dimensional subspace V_N of $C_0^2[0, 1]$ by requiring that the approximate solution, denoted by $u_N(x)$, satisfies

$$u_N(x) = \sum_{j=1}^{N} c_j\phi_j(x), \tag{10.24}$$

where the *trial functions* $\phi_1, \phi_2, \ldots, \phi_n$ form a *basis* for V_N. For now, we only assume that these trial functions belong to $C_0^2[0, 1]$, and are linearly independent. Substituting this form into the weak form yields

$$\sum_{j=1}^{N} \left[\int_0^1 w(x)\phi_j'(x)\,dx\right] c_j = \int_0^1 w(x)f(x)\,dx.$$

Since our trial functions and test functions come from the same space, this version of the weighted mean residual method is known as the *Galerkin method*. As in collocation, we need N equations to uniquely determine the N unknowns $c_1, c_2, \ldots, c_N$. To that end, we use the basis functions $\phi_1, \phi_2, \ldots, \phi_N$ as test functions. This yields the system of equations

$$\sum_{j=1}^{N} \left[\int_0^1 \phi_i'(x)\phi_j'(x)\,dx\right] c_j = \int_0^1 \phi_i(x)f(x)\,dx, \quad i = 1, 2, \ldots, N.$$

This system can be written in matrix-vector form

$$A\mathbf{c} = \mathbf{f}$$

where $\mathbf{u}$ is a vector of the unknown coefficients $c_1, c_2, \ldots, c_N$ and

$$a_{ij} = \int_0^1 \phi_i'(x)\phi_j'(x)\,dx, \quad i, j = 1, 2, \ldots, N,$$

$$f_i = \int_0^1 \phi_i(x)f(x)\,dx, \quad i = 1, 2, \ldots, N.$$

By finding the coefficients $u_1, u_2, \ldots, u_N$ that satisfy these equations, we ensure that the *residual* $R(x, u_N, u_N', u_N'') = f(x) + u_N''(x)$ satisfies

$$\langle w, R \rangle = 0, \quad w \in V_N,$$

as each $w \in V_N$ can be expressed as a linear combination of the test functions $\phi_1, \phi_2, \ldots, \phi_N$. It follows that the residual is *orthogonal* to V_N.

We must now choose trial (and test) functions $\phi_1, \phi_2, \ldots, \phi_N$. A simple choice is a set of piecewise linear "hat" functions, or "tent" functions, so named because of the shapes of their graphs, which are illustrated in Figure 10.4. We divide the interval $[0, 1]$ into $N + 1$ subintervals $[x_{i-1}, x_i]$, for $i = 1, 2, \ldots, N + 1$, with uniform spacing $h = 1/(N + 1)$, where $x_0 = 0$ and $x_{N+1} = 1$.

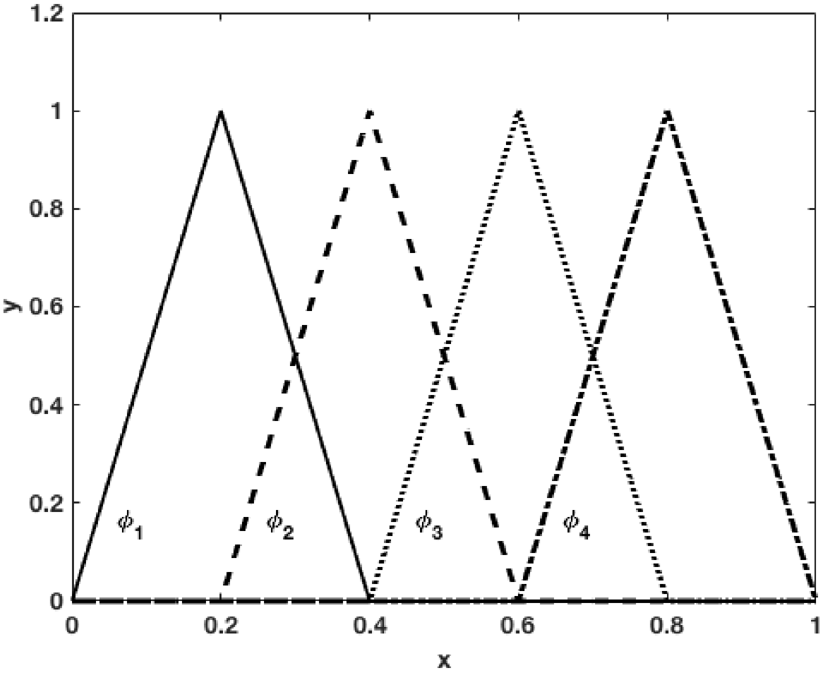


Figure 10.4: Piecewise linear basis functions $\phi_j(x)$, as defined in (10.25), for $j = 1, 2, 3, 4$, with $N = 4$

Then we define

$$\phi_j(x) = \begin{cases} 0 & 0 \le x \le x_{j-1} \\ \frac{1}{h}(x - x_{j-1}) & x_{j-1} < x \le x_j \\ \frac{1}{h}(x_{j+1} - x) & x_j < x \le x_{j+1} \\ 0 & x_{j+1} < x \le 1 \end{cases}, \quad j = 1, 2, \ldots, N. \tag{10.25}$$

These functions automatically satisfy the boundary conditions. Because they are only piecewise linear, their derivatives are discontinuous. They are

$$\phi_j'(x) = \begin{cases} 0 & 0 \le x \le x_{j-1} \\ \frac{1}{h} & x_{j-1} < x \le x_j \\ -\frac{1}{h} & x_j < x \le x_{j+1} \\ 0 & x_{j+1} < x \le 1 \end{cases}, \quad j = 1, 2, \ldots, N.$$

It follows from these definitions that $\phi_i(x)$ and $\phi_j(x)$ cannot simultaneously be nonzero at any point in $[0,1]$ unless $|i-j| \le 1$. This yields a symmetric tridiagonal matrix A with entries

$$\begin{aligned} a_{ii} &= \left(\frac{1}{h}\right)^2 \int_{x_{i-1}}^{x_i} 1\,dx + \left(-\frac{1}{h}\right)^2 \int_{x_i}^{x_{i+1}} 1\,dx = \frac{2}{h}, \quad i = 1, 2, \ldots, N, \\ a_{i,i+1} &= -\frac{1}{h^2} \int_{x_i}^{x_{i+1}} 1\,dx = -\frac{1}{h}, \quad i = 1, 2, \ldots, N-1, \\ a_{i+1,i} &= a_{i,i+1}, \quad i = 1, 2, \ldots, N-1. \end{aligned}$$

For the right-hand side vector $\mathbf{f}$, known as the *load vector*, we have

$$f_i = \frac{1}{h} \int_{x_{i-1}}^{x_i} (x - x_{i-1}) f(x)\,dx + \frac{1}{h} \int_{x_i}^{x_{i+1}} (x_{i+1} - x) f(x)\,dx, \quad i = 1, 2, \ldots, N. \tag{10.26}$$

When the Galerkin method is used with basis functions such as these, that are only nonzero within a small portion of the spatial domain, the method is known as the *finite element method.* In this context, the subintervals $[x_{i-1}, x_i]$ are called *elements*, and each x_i is called a *node*. As we have seen, an advantage of this choice of trial function is that the resulting matrix A, known as the *stiffness matrix*, is sparse.

It can be shown that the matrix A with entries defined from these approximate integrals is not only symmetric and tridiagonal, but also positive definite. It follows that the system $A\mathbf{c} = \mathbf{f}$ is stable with respect to roundoff error, and can be solved using methods such as the conjugate gradient method that are appropriate for sparse symmetric positive definite systems.

Example 10.4.1 *We illustrate the finite element method by solving (10.22), (10.23) with $f(x) = x$, with $N = 4$. The following* MATLAB *commands are used to help specify the problem.*

```
>> % solve -u'' = f on (0,1), u(0)=u(1)=0, f polynomial
>> % represent f(x) = x as a polynomial
>> fx=[ 1 0 ];
>> % set number of interior nodes
>> N=4;
>> h=1/(N+1);
>> % compute vector containing all nodes, including boundary nodes
>> x=h*(0:N+1)';
```

This vector of nodes will be convenient when constructing the load vector **f** and performing other tasks.

We need to solve the system $A\mathbf{c} = \mathbf{f}$, where

$$A = \frac{1}{h}\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix},$$

with $h = 1/5$. The following Matlab commands set up the stiffness matrix for a general value of N.

```
>> % construct stiffness matrix:
>> e=ones(N-1,1);
>> % use diag to place entries on subdiagonal and superdiagonal
>> A=1/h*(2*eye(N)-diag(e,1)-diag(e,-1));
```

The load vector **f** has elements

$$f_i = \frac{1}{h}\int_{x_{i-1}}^{x_i} (x - x_{i-1})x\,dx + \frac{1}{h}\int_{x_i}^{x_{i+1}} (x_{i+1} - x)x\,dx, \quad i = 1, 2, \ldots, N.$$

The following statements compute these elements when f is a polynomial.

```
% construct load vector:
% pre-allocate column vector
f=zeros(N,1);
for i=1:N
    % note that in text, 0-based indexing is used
    % for x-values, while Matlab uses 1-based indexing
    % phi_{i-1}(x) = (x - x_{i-1})/h
    hat1=[ 1 -x(i) ]/h;
    % multiply hat function by f(x)
    integrand1=conv(fx,hat1);
    % anti-differentiate
    antideriv1=polyint(integrand1);
    % substitute limits of integration into antiderivative
    integral1=polyval(antideriv1,x(i+1))-polyval(antideriv1,x(i));
    % phi_i(x) = (x_{i+1} - x)/h
    hat2=[ -1 x(i+2) ]/h;
    % repeat integration process on [x_i,x_{i+1}]
    integrand2=conv(fx,hat2);
    % anti-differentiate
    antideriv2=polyint(integrand2);
    % substitute limits of integration into antiderivative
    integral2=polyval(antideriv2,x(i+2))-polyval(antideriv2,x(i+1));
    f(i)=integral1+integral2;
end
```

Now that the system $A\mathbf{c} = \mathbf{f}$ is set up, we can easily solve it in MATLAB using the command `c=A\f`.

For this simple BVP, we can obtain the exact solution analytically, to help gauge the accuracy of our approximate solution. The following statements accomplish this for the BVP $-u'' = f$ on $(0, 1)$, $u(x_0) = u(x_{N+1}) = 1$, when f is a polynomial.

```
% obtain exact solution by integration
u=-polyint(polyint(fx));
% to solve for constants of integration:
% u + d1 x + d2 = 0 at x = x_0, x_{N+1}
% in matrix-vector form:
% [ x_0     1 ] [ d1 ] = [ -u(x0)      ]
% [ x_{N+1} 1 ] [ d2 ]   [ -u(x_{N+1}) ]
V=[ x(1) 1; x(end) 1 ];
b=[ -polyval(u,x(1)); -polyval(u,x(end)) ];
d=V\b;
u=u+[ 0 0 d(1) d(2) ];
```

Now, we can visualize the exact solution $u(x)$ and approximate solution $u_N(x)$, which is a piecewise linear function due to the use of piecewise linear trial functions.

```
% make vector of x-values for plotting exact solution
xp=x(1):h/100:x(end);
plot(xp,polyval(u,xp),'b')
hold on
plot(x,[ 0; c; 0 ],'r--o')
hold off
xlabel('x')
ylabel('y')
```

Because each of the trial functions $\phi_j(x)$, $j = 1, 2, 3, 4$, is equal to 1 at x_j and equal to 0 at x_i for $i \neq j$, each element c_j of $\mathbf{c}$ is equal to the value of $u_4(x)$ at the corresponding node x_j. The exact and approximate solutions are shown in Figure 10.5. It can be seen that there is very close agreement between the exact and approximate solutions at the nodes; in fact, in this example, they are exactly equal, though this does not occur in general. □

In the preceding example, the integrals in (10.26) could be evaluated exactly. Generally, however, they must be approximated, using techniques such as those presented in Chapter 7.

Exercise 10.4.1 *What is the value of f_i if the Trapezoidal Rule is used on each of the integrals in (10.26)? What if Simpson's Rule is used?*

Exercise 10.4.2 *Write a* MATLAB *function* `[x,u]=FEMBVP(f,N)` *that solves the BVP (10.22), (10.23) with* `N` interior *nodes. The input argument* `f` *must be a function handle. The outputs* `x` *and* `u` *must be column vectors consisting of the nodes and values of the approximate solution at the nodes, respectively. Use the Trapezoidal rule to approximate the integrals (10.26). Test your function with $f(x) = e^x$. What happens to the error as N increases?*

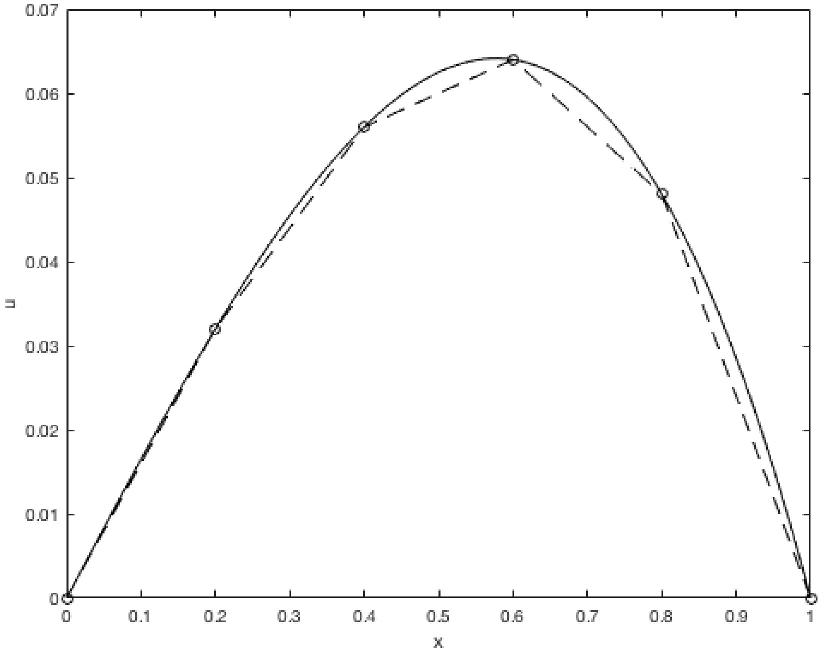


Figure 10.5: Exact (solid curve) and approximate (dashed curve) solutions of (10.22), (10.23) with $f(x) = x$ and $N = 4$

Exercise 10.4.3 *Generalize your function* `FEMBVP` *from Exercise 10.4.2 so that it can be used to solve the BVP* $-u'' + q(x)u = f(x)$ *on* $(0,1)$*, with boundary conditions* $u(0) = u(1) = 0$*, for a given function* $q(x)$ *that must be passed as an input argument that is a function handle.* Hint: *re-derive the weak form of the BVP to determine how the matrix* A *must be modified. Use the Trapezoidal Rule to approximate any integrals involving* $q(x)$*.*

Exercise 10.4.4 *Modify your function* `FEMBVP` *from Exercise 10.4.3 so that it can be used to solve the BVP* $-u'' + q(x)u = f(x)$ *on* $(0,1)$*, with boundary conditions* $u(0) = u_0$*,* $u(1) = u_1$*, where the scalars* u_0 *and* u_1 *must be passed as input arguments.* Hint: *Modify (10.24) to include additional basis functions* $\phi_0(x)$ *and* $\phi_{N+1}(x)$*, that are equal to 1 at* $x = x_0$ *and* $x = x_{N+1}$*, respectively, and equal to 0 at all other nodes. How must the load vector* $\mathbf{f}$ *be modified to account for these nonhomogeneous boundary conditions?*

Exercise 10.4.5 *Modify your function* `FEMBVP` *from Exercise 10.4.4 so that it can be used to solve the BVP* $-(p(x)u')' + q(x)u = f(x)$ *on* $(0,1)$*, with boundary conditions* $u(0) = u_0$*,* $u(1) = u_1$*, where the coefficient* $p(x)$ *must be passed as an input argument that is a function handle.* Hint: *re-derive the weak form of the BVP to determine how the matrix* A *must be modified. Use the Trapezoidal Rule to approximate any integrals involving* $p(x)$*.*

It can be shown that when using the finite element method with piecewise linear trial functions, the error in the approximate solution is $O(h^2)$. Higher-order accuracy can be achieved by using higher-degree piecewise polynomials as basis functions, such as cubic B-splines. Such a choice also helps to ensure that the approximate solution is differentiable, unlike the solution computed using piecewise linear basis functions, which are continuous but not differentiable at the points x_i, $i = 1, 2, \ldots, N$. With cubic B-splines, the error in the computed solution is $O(h^4)$ as opposed to $O(h^2)$ in the piecewise linear case, due to the two additional degrees of differentiability. However, the drawback is that the matrix arising form the use of higher-degree basis functions is no longer tridiagonal; the upper and lower bandwidth are each equal to the degree of the piecewise polynomial that is used.

10.5 Further Reading

Part V

Appendices

Appendix A

Review of Calculus

Among the mathematical problems that can be solved using techniques from numerical analysis are the basic problems of differential and integral calculus:

- computing the instantaneous rate of change of one quantity with respect to another, which is a *derivative*, and

- computing the total change in a function over some portion of its domain, which is a *definite integral.*

Calculus also plays an essential role in the development and analysis of techniques used in numerical analysis, including those techniques that are applied to problems not arising directly from calculus. Therefore, it is appropriate to review some basic concepts from calculus before we begin our study of numerical analysis.

A.1 Limits and Continuity

A.1.1 Limits

The basic problems of differential and integral calculus described in the previous paragraph can be solved by computing a sequence of approximations to the desired quantity and then determining what value, if any, the sequence of approximations approaches. This value is called a *limit* of the sequence. As a sequence is a function, we begin by defining, precisely, the concept of the limit of a function.

Definition A.1.1 *We write*

$$\lim_{x\to a} f(x) = L$$

if for any open interval I_1 containing L, there is some open interval I_2 containing a such that $f(x) \in I_1$ whenever $x \in I_2$, and $x \neq a$. We say that L is the **limit of** $f(x)$ **as** x **approaches** a.

We write

$$\lim_{x\to a^-} f(x) = L$$

if, for any open interval I_1 containing L, there is an open interval I_2 of the form (c, a), where $c < a$, such that $f(x) \in I_1$ whenever $x \in I_2$. We say that L is the **limit of** $f(x)$ **as** x **approaches** a **from the left**, *or the* **left-hand limit of** $f(x)$ **as** x **approaches** a.

Similarly, we write

$$\lim_{x\to a^+} f(x) = L$$

if, for any open interval I_1 containing L, there is an open interval I_2 of the form (a, c), where $c > a$, such that $f(x) \in I_1$ whenever $x \in I_2$. We say that L is the **limit of** $f(x)$ **as** x **approaches** a **from the right**, *or the* **right-hand limit of** $f(x)$ **as** x **approaches** a.

We can make the definition of a limit a little more concrete by imposing sizes on the intervals I_1 and I_2, as long as the interval I_1 can still be of arbitrary size. It can be shown that the following definition is equivalent to the previous one.

Definition A.1.2 *We write*

$$\lim_{x\to a} f(x) = L$$

if, for any $\epsilon > 0$, there exists a number $\delta > 0$ such that $|f(x) - L| < \epsilon$ whenever $0 < |x - a| < \delta$.

Similar definitions can be given for the left-hand and right-hand limits.

Note that in either definition, the point $x = a$ is specifically excluded from consideration when requiring that $f(x)$ be close to L whenever x is close to a. This is because the concept of a limit is only intended to describe the behavior of $f(x)$ *near* $x = a$, as opposed to its behavior *at* $x = a$. Later in this appendix we discuss the case where the two distinct behaviors coincide.

To illustrate limits, we consider

$$L = \lim_{x\to 0^+} \frac{\sin x}{x}. \tag{A.1}$$

We will visualize this limit in MATLAB. First, we construct a vector of x-values that are near zero, but excluding zero. This can readily be accomplished using the colon operator:

```
>> dx=0.01;
>> x=dx:dx:1;
```

Then, the vector `x` contains the values $x_i = i\Delta x$, $i = 1, 2, \ldots, 100$, where $\Delta x = 0.01$.

Exercise A.1.1 *Use the vector* `x` *to plot* $\sin x/x$ *on the interval* $(0, 1]$. *What appears to be the value of the limit L in (A.1)?*

The preceding exercise can be completed using a `for` loop, but it is much easier to use componentwise operators. Since `x` is a vector, the expression `sin(x)/x` would cause an error. Instead, the `./` operator can be used to perform componentwise division of the vectors `sin(x)` and `x`. The `.` can be used with several other arithmetic operators to perform componentwise operations on matrices and vectors. For example, if `A` is a matrix, then `A.^2` is a matrix in which each entry is the square of the corresponding entry of `A`.

Exercise A.1.2 *Use* one *statement to plot* $\sin x/x$ *on the interval* $(0, 1]$.

A.1.2 Limits of Functions of Several Variables

The notions of limit and continuity generalize to vector-valued functions and functions of several variables in a straightforward way.

Definition A.1.3 *Given a function* $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ *and a point* $\mathbf{x}_0 \in D$*, we write*

$$\lim_{\mathbf{x}\to\mathbf{x}_0} f(\mathbf{x}) = L$$

if, for any $\epsilon > 0$*, there exists a* $\delta > 0$ *such that*

$$|f(\mathbf{x}) - L| < \epsilon$$

whenever $\mathbf{x} \in D$ *and*

$$0 < \|\mathbf{x} - \mathbf{x}_0\| < \delta.$$

In this definition, we can use any appropriate vector norm $\|\cdot\|$, as discussed in Section B.11.

Definition A.1.4 *We also say that* f *is* continuous *at a point* $\mathbf{x}_0 \in D$ *if*

$$\lim_{\mathbf{x}\to\mathbf{x}_0} f(\mathbf{x}) = f(\mathbf{x}_0).$$

It can be shown f is continuous at $\mathbf{x}_0$ if its partial derviatives are bounded near $\mathbf{x}_0$.

Having defined limits and continuity for scalar-valued functions of several variables, we can now define these concepts for vector-valued functions. Given a vector-valued function $\mathbf{F} : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$, and $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in D$, we write

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix}$$

where $f_1, f_2, \ldots, f_n$ are the *component functions*, or *coordinate functions*, of $\mathbf{F}$.

Definition A.1.5 *Given* $\mathbf{F} : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ *and* $\mathbf{x}_0 \in D$*, we say that*

$$\lim_{\mathbf{x}\to\mathbf{x}_0} \mathbf{F}(\mathbf{x}) = \mathbf{L}$$

if and only if

$$\lim_{\mathbf{x}\to\mathbf{x}_0} f_i(\mathbf{x}) = L_i, \quad i = 1, 2, \ldots, n.$$

Similarly, we say that $\mathbf{F}$ *is* continuous *at* $\mathbf{x}_0$ *if and only if each coordinate function* f_i *is continuous at* $\mathbf{x}_0$*. Equivalently,* $\mathbf{F}$ *is continuous at* $\mathbf{x}_0$ *if*

$$\lim_{\mathbf{x}\to\mathbf{x}_0} \mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{x}_0).$$

A.1.3 Limits at Infinity

The concept of a limit defined above is useful for describing the behavior of a function $f(x)$ as x approaches a *finite* value a. However, suppose that the function f is a *sequence*, which is a function that maps $\mathbb{N}$, the set of natural numbers, to $\mathbb{R}$, the set of real numbers. We will denote such a sequence by $\{f_n\}_{n=0}^{\infty}$, or simply $\{f_n\}$. In numerical analysis, it is sometimes necessary to determine the value that the terms of a sequence $\{f_n\}$ approach as $n \to \infty$. Such a value, if it exists, is not a limit, as defined previously. However, it is natural to use the notation of limits to describe this behavior of a function. We therefore define what it means for a sequence $\{f_n\}$ to have a limit as n becomes infinite.

Definition A.1.6 (Limit at Infinity) *Let* $\{f_n\}$ *be a sequence defined for all integers not less than some integer* n_0*. We say that the* **limit of** $\{f_n\}$ **as** n **approaches** ∞ *is equal to* L*, and write*

$$\lim_{n\to\infty} f_n = L,$$

if for any open interval I *containing* L*, there exists a number* M *such that* $f_n \in I$ *whenever* $x > M$*.*

Example A.1.7 *Let the sequence* $\{f_n\}_{n=1}^{\infty}$ *be defined by* $f_n = 1/n$ *for every positive integer* n*. Then*

$$\lim_{n\to\infty} f_n = 0,$$

since for any $\epsilon > 0$*, no matter how small, we can find a positive integer* n_0 *such that* $|f_n| < \epsilon$ *for all* $n \geq n_0$*. In fact, for any given* ϵ*, we can choose* $n_0 = \lceil 1/\epsilon \rceil$*, where* $\lceil x \rceil$*, known as the* ceiling function*, denotes the smallest integer that is greater than or equal to* x*.* □

A.1.4 Continuity

In many cases, the limit of a function $f(x)$ as x approached a could be obtained by simply computing $f(a)$. Intuitively, this indicates that f has to have a graph that is one continuous curve, because any "break" or "jump" in the graph at $x = a$ is caused by f approaching one value as x approaches a, only to actually assume a different value at a. This leads to the following precise definition of what it means for a function to be continuous at a given point.

Definition A.1.8 (Continuity) *We say that a function f is* **continuous at** *a if*

$$\lim_{x \to a} f(x) = f(a).$$

We also say that $f(x)$ has the **Direct Subsitution Property** *at $x = a$.*
We say that a function f is **continuous from the right** *at a if*

$$\lim_{x \to a^+} f(x) = f(a).$$

Similarly, we say that f is **continuous from the left** *at a if*

$$\lim_{x \to a^-} f(x) = f(a).$$

The preceding definition describes continuity at a single point. In describing where a function is continuous, the concept of continuity over an interval is useful, so we define this concept as well.

Definition A.1.9 (Continuity on an Interval) *We say that a function f is* **continuous on the interval** *(a, b) if f is continuous at every point in (a, b). Similarly, we say that f is continuous on*

1. *$[a, b)$ if f is continuous on (a, b), and continuous from the right at a.*
2. *$(a, b]$ if f is continuous on (a, b), and continuous from the left at b.*
3. *$[a, b]$ if f is continuous on (a, b), continuous from the right at a, and continuous from the left at b.*

In numerical analysis, it is often necessary to construct a continuous function, such as a polynomial, based on data obtained by measurements and problem-dependent constraints. In this course, we will learn some of the most basic techniques for constructing such continuous functions by a process called *interpolation*.

A.1.5 The Intermediate Value Theorem

Suppose that a function f is continuous on some closed interval $[a, b]$. The graph of such a function is a continuous curve connecting the points $(a, f(a))$ with $(b, f(b))$. If one were to draw such a graph, their pen would not leave the paper in the process, and therefore it would be impossible to "avoid" any y-value between $f(a)$ and $f(b)$. This leads to the following statement about such continuous functions.

Theorem A.1.10 (Intermediate Value Theorem) *Let f be continuous on $[a, b]$. Then, on (a, b), f assumes every value between $f(a)$ and $f(b)$; that is, for any value y between $f(a)$ and $f(b)$, $f(c) = y$ for some c in (a, b).*

The Intermediate Value Theorem has a very important application in the problem of finding solutions of a general equation of the form $f(x) = 0$, where x is the solution we wish to compute and f is a given continuous function. Often, methods for solving such an equation try to identify an interval $[a, b]$ where $f(a) > 0$ and $f(b) < 0$, or vice versa. In either case, the Intermediate Value

Theorem states that f must assume every value between $f(a)$ and $f(b)$, and since 0 is one such value, it follows that the equation $f(x) = 0$ must have a solution somewhere in the interval (a, b).

We can find an approximation to this solution using a procedure called *bisection*, which repeatedly applies the Intermediate Value Theorem to smaller and smaller intervals that contain the solution. We will study bisection, and other methods for solving the equation $f(x) = 0$, in this course.

A.2 Derivatives

The basic problem of differential calculus is computing the instantaneous rate of change of one quantity y with respect to another quantity x. For example, y may represent the position of an object and x may represent time, in which case the instantaneous rate of change of y with respect to x is interpreted as the velocity of the object.

When the two quantities x and y are related by an equation of the form $y = f(x)$, it is certainly convenient to describe the rate of change of y with respect to x in terms of the function f. Because the instantaneous rate of change is so commonplace, it is practical to assign a concise name and notation to it, which we do now.

Definition A.2.1 (Derivative) *The* **derivative** *of a function $f(x)$ at $x = a$, denoted by $f'(a)$, is*

$$f'(a) = \lim_{h\to 0} \frac{f(a+h) - f(a)}{h},$$

provided that the above limit exists. When this limit exists, we say that f is **differentiable** *at a.*

Remark Given a function $f(x)$ that is differentiable at $x = a$, the following numbers are all equal:

- the derivative of f at $x = a$, $f'(a)$,
- the slope of the tangent line of f at the point $(a, f(a))$, and
- the instantaneous rate of change of $y = f(x)$ with respect to x at $x = a$.

This can be seen from the fact that all three numbers are defined in the same way. □

Exercise A.2.1 *Let $f(x) = x^2 - 3x + 2$. Use the* MATLAB *function* `polyder` *to compute the coefficients of $f'(x)$. Then use the* `polyval` *function to obtain the equation of the tangent line of $f(x)$ at $x = 2$. Finally, plot the graph of $f(x)$ and this tangent line, on the same graph, restricted to the interval* $[0, 4]$.

Many functions can be differentiated using differentiation rules such as those learned in a calculus course. However, many functions cannot be differentiated using these rules. For example, we may need to compute the instantaneous rate of change of a quantity $y = f(x)$ with respect to another quantity x, where our only knowledge of the function f that relates x and y is a set of pairs of x-values and y-values that may be obtained using measurements. In this course we will learn how to approximate the derivative of such a function using this limited information. The most common methods involve constructing a continuous function, such as a polynomial, based on the given data, using interpolation. The polynomial can then be differentiated using differentiation rules. Since the polynomial is an approximation to the function $f(x)$, its derivative is an approximation to $f'(x)$.

A.2.1 Differentiability and Continuity

Consider a tangent line of a function f at a point $(a, f(a))$. When we consider that this tangent line is the limit of secant lines that can cross the graph of f at points on *either side* of a, it seems impossible that f can fail to be continuous at a. The following result confirms this: a function that is differentiable at a given point (and therefore has a tangent line at that point) *must* also be continuous at that point.

Theorem A.2.2 *If f is differentiable at a, then f is continuous at a.*

It is important to keep in mind, however, that the *converse* of the above statement, "if f is continuous, then f is differentiable", is not true. It is actually very easy to find examples of functions that are continuous at a point, but fail to be differentiable at that point. As an extreme example, it is known that there is a function that is continuous everywhere, but is differentiable *nowhere.*

Example A.2.3 *The functions $f(x) = |x|$ and $g(x) = x^{1/3}$ are examples of functions that are continuous for all x, but are not differentiable at $x = 0$. The graph of the* absolute value function *$|x|$ has a sharp corner at $x = 0$, since the one-sided limits*

$$\lim_{h\to 0^-} \frac{f(h) - f(0)}{h} = -1, \quad \lim_{h\to 0^+} \frac{f(h) - f(0)}{h} = 1$$

do not agree, but in general these limits must agree in order for $f(x)$ to have a derivative at $x = 0$.

The cube root function $g(x) = x^{1/3}$ is not differentiable at $x = 0$ because the tangent line to the graph at the point $(0, 0)$ is vertical, so it has no finite slope. We can also see that the derivative does not exist at this point by noting that the function $g'(x) = (1/3)x^{-2/3}$ has a vertical asymptote at $x = 0$.

Exercise A.2.2 *Plot both of the functions from this example on the interval $[-1, 1]$. Use the colon operator to create a vector of x-values and use the dot for performing componentwise operations on a vector, where applicable. From the plot, identify the non-differentiability in these continuous functions.* □

A.3 Extreme Values

In many applications, it is necessary to determine where a given function attains its minimum or maximum value. For example, a business wishes to maximize profit, so it can construct a function that relates its profit to variables such as payroll or maintenance costs. We now consider the basic problem of finding a maximum or minimum value of a general function $f(x)$ that depends on a single independent variable x. First, we must precisely define what it means for a function to *have* a maximum or minimum value.

Definition A.3.1 (Absolute extrema) *A function f has a* **absolute maximum** *or* **global maximum** *at c if $f(c) \geq f(x)$ for all x in the domain of f. The number $f(c)$ is called the* **maximum value** *of f on its domain. Similarly, f has a* **absolute minimum** *or* **global minimum** *at c if $f(c) \leq f(x)$ for all x in the domain of f. The number $f(c)$ is then called the* **minimum value** *of f on its domain. The maximum and minimum values of f are called the* **extreme values** *of f, and the absolute maximum and minimum are each called an* **extremum** *of f.*

Before computing the maximum or minimum value of a function, it is natural to ask whether it is possible to determine in advance whether a function even has a maximum or minimum, so that effort is not wasted in trying to solve a problem that has no solution. The following result is very helpful in answering this question.

Theorem A.3.2 (Extreme Value Theorem) *If f is continuous on $[a, b]$, then f has an absolute maximum and an absolute minimum on $[a, b]$.*

Now that we can easily determine whether a function has a maximum or minimum on a closed interval $[a, b]$, we can develop an method for actually finding them. It turns out that it is easier to find points at which f attains a maximum or minimum value in a "local" sense, rather than a "global" sense. In other words, we can best find the absolute maximum or minimum of f by finding points at which f achieves a maximum or minimum with respect to "nearby" points, and then determine which of these points is the absolute maximum or minimum. The following definition makes this notion precise.

Definition A.3.3 (Local extrema) *A function f has a* **local maximum** *at c if $f(c) \geq f(x)$ for all x in an open interval containing c. Similarly, f has a* **local minimum** *at c if $f(c) \leq f(x)$ for all x in an open interval containing c. A local maximum or local minimum is also called a* **local extremum**.

At each point at which f has a local maximum, the function either has a horizontal tangent line, or no tangent line due to not being differentiable. It turns out that this is true in general, and a similar statement applies to local minima. To state the formal result, we first introduce the following definition, which will also be useful when describing a method for finding local extrema.

Definition A.3.4 (Critical Number) *A number c in the domain of a function f is a* **critical number** *of f if $f'(c) = 0$ or $f'(c)$ does not exist.*

The following result describes the relationship between critical numbers and local extrema.

Theorem A.3.5 (Fermat's Theorem) *If f has a local minimum or local maximum at c, then c is a critical number of f; that is, either $f'(c) = 0$ or $f'(c)$ does not exist.*

This theorem suggests that the maximum or minimum value of a function $f(x)$ can be found by solving the equation $f'(x) = 0$. As mentioned previously, we will be learning techniques for solving such equations in this course. These techniques play an essential role in the solution of problems in which one must compute the maximum or minimum value of a function, subject to constraints on its variables. Such problems are called *optimization problems*.

The following exercise highlights the significance of critical numbers. It relies on some of the

MATLAB functions for working with polynomials that were introduced in Section 1.2.

Exercise A.3.1 *Consider the polynomial* $f(x) = x^3 - 4x^2 + 5x - 2$*. Plot the graph of this function on the interval* $[0, 3]$*. Use the colon operator to create a vector of* x*-values and use the dot for componentwise operations on vectors wherever needed. Then, use the* `polyder` *and* `roots` *functions to compute the critical numbers of* $f(x)$*. How do they relate to the absolute maximum and minimum values of* $f(x)$ *on* $[0, 3]$*, or any local maxima or minima on this interval?*

A.4 Integrals

There are many cases in which some quantity is defined to be the product of two other quantities. For example, a rectangle of width w has uniform height h, and the area A of the rectangle is given by the formula $A = wh$. Unfortunately, in many applications, we cannot necessarily assume that certain quantities such as height are constant, and therefore formulas such as $A = wh$ cannot be used directly. However, they can be used indirectly to solve more general problems by employing the notation known as *integral calculus*.

Suppose we wish to compute the area of a shape that is not a rectangle. To simplify the discussion, we assume that the shape is bounded by the vertical lines $x = a$ and $x = b$, the x-axis, and the curve defined by some continuous function $y = f(x)$, where $f(x) \geq 0$ for $a \leq x \leq b$. Then, we can approximate this shape by n rectangles that have width $\Delta x = (b - a)/n$ and height $f(x_i)$, where $x_i = a + i\Delta x$, for $i = 0, \ldots, n$. We obtain the approximation

$$A \approx A_n = \sum_{i=1}^{n} f(x_i)\Delta x.$$

Intuitively, we can conclude that as $n \to \infty$, the approximate area A_n will converge to the exact area of the given region. This can be seen by observing that as n increases, the n rectangles defined above comprise a more accurate approximation of the region.

More generally, suppose that for each $n = 1, 2, \ldots$, we define the quantity R_n by choosing points $a = x_0 < x_1 < \cdots < x_n = b$, and computing the sum

$$R_n = \sum_{i=1}^{n} f(x_i^*)\Delta x_i, \quad \Delta x_i = x_i - x_{i-1}, \quad x_{i-1} \leq x_i^* \leq x_i.$$

The sum that defines R_n is known as a *Riemann sum*. Note that the interval $[a, b]$ need not be divided into subintervals of equal width, and that $f(x)$ can be evaluated at *arbitrary* points belonging to each subinterval.

If $f(x) \geq 0$ on $[a, b]$, then R_n converges to the area under the curve $y = f(x)$ as $n \to \infty$, provided that the widths of all of the subintervals $[x_{i-1}, x_i]$, for $i = 1, \ldots, n$, approach zero. This behavior is ensured if we require that

$$\lim_{n\to\infty} \delta(n) = 0, \quad \text{where} \quad \delta(n) = \max_{1\leq i\leq n} \Delta x_i.$$

This condition is necessary because if it does not hold, then, as $n \to \infty$, the region formed by the n rectangles will not converge to the region whose area we wish to compute. If f assumes negative values on $[a, b]$, then, under the same conditions on the widths of the subintervals, R_n converges to the *net* area between the graph of f and the x-axis, where area below the x-axis is counted negatively.

Definition A.4.1 *We define the* definite integral *of $f(x)$ from a to b by*

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} R_n,$$

where the sequence of Riemann sums $\{R_n\}_{n=1}^\infty$ is defined so that $\delta(n) \to 0$ as $n \to \infty$, as in the previous discussion. The function $f(x)$ is called the integrand*, and the values a and b are the* lower *and* upper limits of integration*, respectively. The process of computing an integral is called* integration*.*

In Chapter 7, we will study the problem of computing an approximation to the definite integral of a given function $f(x)$ over an interval $[a, b]$. We will learn a number of techniques for computing such an approximation, and all of these techniques involve the computation of an appropriate Riemann sum.

Exercise A.4.1 *Let $f(x) = e^{-x^2}$. Write a* Matlab *function that takes as input a parameter n, and computes the Riemann sum R_n for $f(x)$ using n rectangles. Use a* `for` *loop to compute the Riemann sum. First use $x_i^* = x_{i-1}$, then use $x_i^* = x_i$, and then use $x_i^* = (x_{i-1} + x_i)2$. For each case, compute Riemann sums for several values of n. What can you observe about the convergence of the Riemann sum in these three cases?*

A.5 The Mean Value Theorem

While the derivative describes the behavior of a function at a point, we often need to understand how the derivative influences a function's behavior on an interval. This understanding is essential in numerical analysis because, it is often necessary to approximate a function $f(x)$ by a function $g(x)$ using knowledge of $f(x)$ and its derivatives at various points. It is therefore natural to ask how well $g(x)$ approximates $f(x)$ away from these points.

The following result, a consequence of Fermat's Theorem, gives limited insight into the relationship between the behavior of a function on an interval and the value of its derivative at a point.

Theorem A.5.1 (Rolle's Theorem) *If f is continuous on a closed interval $[a, b]$ and is differentiable on the open interval (a, b), and if $f(a) = f(b)$, then $f'(c) = 0$ for some number c in (a, b).*

By applying Rolle's Theorem to a function f, then to its derivative f', its second derivative f'', and so on, we obtain the following more general result, which will be useful in analyzing the accuracy of methods for approximating functions by polynomials.

Theorem A.5.2 (Generalized Rolle's Theorem) *Let x_0, x_1, x_2, …, x_n be distinct points in an interval $[a, b]$. If f is n times differentiable on (a, b), and if $f(x_i) = 0$ for $i = 0, 1, 2, \ldots, n$, then $f^{(n)}(c) = 0$ for some number c in (a, b).*

A more fundamental consequence of Rolle's Theorem is the Mean Value Theorem itself, which we now state.

Theorem A.5.3 (Mean Value Theorem) *If f is continuous on a closed interval $[a, b]$ and is differentiable on the open interval (a, b), then*

$$\frac{f(b) - f(a)}{b - a} = f'(c)$$

for some number c in (a, b).

Remark The expression

$$\frac{f(b) - f(a)}{b - a}$$

is the slope of the secant line passing through the points $(a, f(a))$ and $(b, f(b))$. The Mean Value Theorem therefore states that under the given assumptions, the slope of this secant line is equal to the slope of the tangent line of f at the point $(c, f(c))$, where $c \in (a, b)$. □

The Mean Value Theorem has the following practical interpretation: the average rate of change of $y = f(x)$ with respect to x on an interval $[a, b]$ is equal to the instantaneous rate of change y with respect to x at some point in (a, b).

A.5.1 The Mean Value Theorem for Integrals

Suppose that $f(x)$ is a continuous function on an interval $[a, b]$. Then, by the Fundamental Theorem of Calculus, $f(x)$ has an antiderivative $F(x)$ defined on $[a, b]$ such that $F'(x) = f(x)$. If we apply the Mean Value Theorem to $F(x)$, we obtain the following relationship between the integral of f over $[a, b]$ and the value of f at a point in (a, b).

Theorem A.5.4 (Mean Value Theorem for Integrals) *If f is continuous on $[a, b]$, then*

$$\int_a^b f(x)\,dx = f(c)(b - a)$$

for some c in (a, b).

In other words, f assumes its average value over $[a, b]$, defined by

$$f_{ave} = \frac{1}{b - a}\int_a^b f(x)\,dx,$$

at some point in $[a, b]$, just as the Mean Value Theorem states that the derivative of a function assumes its average value over an interval at some point in the interval.

The Mean Value Theorem for Integrals is also a special case of the following more general result.

Theorem A.5.5 (Weighted Mean Value Theorem for Integrals) *If f is continuous on $[a, b]$, and g is a function that is integrable on $[a, b]$ and does not change sign on $[a, b]$, then*

$$\int_a^b f(x)g(x)\,dx = f(c)\int_a^b g(x)\,dx$$

for some c in (a, b).

In the case where $g(x)$ is a function that is easy to antidifferentiate and $f(x)$ is not, this theorem can be used to obtain an estimate of the integral of $f(x)g(x)$ over an interval.

Example A.5.6 *Let $f(x)$ be continuous on the interval $[a, b]$. Then, for any $x \in [a, b]$, by the Weighted Mean Value Theorem for Integrals, we have*

$$\int_a^x f(s)(s-a)\,ds = f(c)\int_a^x (s-a)\,ds = f(c)\left.\frac{(s-a)^2}{2}\right|_a^x = f(c)\frac{(x-a)^2}{2},$$

where $a < c < x$. It is important to note that we can apply the Weighted Mean Value Theorem because the function $g(x) = (x - a)$ does not change sign on $[a, b]$. □

A.6 Taylor's Theorem

In many cases, it is useful to approximate a given function $f(x)$ by a polynomial, because one can work much more easily with polynomials than with other types of functions. As such, it is necessary to have some insight into the accuracy of such an approximation. The following theorem, which is a consequence of the Weighted Mean Value Theorem for Integrals, provides this insight.

Theorem A.6.1 (Taylor's Theorem) *Let f be n times continuously differentiable on an interval $[a, b]$, and suppose that $f^{(n+1)}$ exists on $[a, b]$. Let $x_0 \in [a, b]$. Then, for any point $x \in [a, b]$,*

$$f(x) = P_n(x) + R_n(x),$$

where

$$\begin{aligned} P_n(x) &= \sum_{j=0}^{n} \frac{f^{(j)}(x_0)}{j!}(x-x_0)^j \\ &= f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2}f''(x_0)(x-x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n \end{aligned}$$

and

$$R_n(x) = \int_{x_0}^x \frac{f^{(n+1)}(s)}{n!}(x-s)^n\,ds = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)^{n+1},$$

where $\xi(x)$ is between x_0 and x.

The polynomial $P_n(x)$ is the nth *Taylor polynomial* of f with *center* x_0, and the expression $R_n(x)$ is called the *Taylor remainder* of $P_n(x)$. When the center x_0 is zero, the nth Taylor polynomial is also known as the nth *Maclaurin polynomial*.

Exercise A.6.1 *Plot the graph of* $f(x) = \cos x$ *on the interval* $[0, \pi]$*, and then use the* `hold` *command to include the graphs of* $P_0(x)$*,* $P_2(x)$*, and* $P_4(x)$*, the Maclaurin polynomials of degree 0, 2 and 4, respectively, on the same plot but with different colors and line styles. To what extent do these Taylor polynomials agree with* $f(x)$*?*

The final form of the remainder is obtained by applying the Mean Value Theorem for Integrals to the integral form. As $P_n(x)$ can be used to approximate $f(x)$, the remainder $R_n(x)$ is also referred to as the *truncation error* of $P_n(x)$. The accuracy of the approximation on an interval can be analyzed by using techniques for finding the extreme values of functions to bound the $(n+1)$-st derivative on the interval.

Because approximation of functions by polynomials is employed in the development and analysis of many techniques in numerical analysis, the usefulness of Taylor's Theorem cannot be overstated. In fact, it can be said that Taylor's Theorem is the Fundamental Theorem of Numerical Analysis, just as the theorem describing inverse relationship between derivatives and integrals is called the Fundamental Theorem of Calculus.

The following examples illustrate how the nth-degree Taylor polynomial $P_n(x)$ and the remainder $R_n(x)$ can be computed for a given function $f(x)$.

Example A.6.2 *If we set* $n = 1$ *in Taylor's Theorem, then we have*

$$f(x) = P_1(x) + R_1(x)$$

where

$$P_1(x) = f(x_0) + f'(x_0)(x - x_0).$$

This polynomial is a linear function that describes the tangent line to the graph of f *at the point* $(x_0, f(x_0))$*.*

If we set $n = 0$ *in the theorem, then we obtain*

$$f(x) = P_0(x) + R_0(x),$$

where

$$P_0(x) = f(x_0)$$

and

$$R_0(x) = f'(\xi(x))(x - x_0),$$

where $\xi(x)$ *lies between* x_0 *and* x*. If we use the integral form of the remainder,*

$$R_n(x) = \int_{x_0}^{x} \frac{f^{(n+1)}(s)}{n!}(x - s)^n \, ds,$$

then we have

$$f(x) = f(x_0) + \int_{x_0}^{x} f'(s) \, ds,$$

which is equivalent to the Total Change Theorem and part of the Fundamental Theorem of Calculus. Using the Mean Value Theorem for integrals, we can see how the first form of the remainder can be obtained from the integral form. □

Example A.6.3 *Let* $f(x) = \sin x$. *Then*

$$f(x) = P_3(x) + R_3(x),$$

where

$$P_3(x) = x - \frac{x^3}{3!} = x - \frac{x^3}{6},$$

and

$$R_3(x) = \frac{1}{4!}x^4 \sin \xi(x) = \frac{1}{24}x^4 \sin \xi(x),$$

where $\xi(x)$ *is between 0 and* x. *The polynomial* $P_3(x)$ *is the 3rd Maclaurin polynomial of* $\sin x$, *or the 3rd Taylor polynomial with center* $x_0 = 0$.

If $x \in [-1, 1]$, *then*

$$|R_n(x)| = \left|\frac{1}{24}x^4 \sin \xi(x)\right| = \left|\frac{1}{24}\right| |x^4| |\sin \xi(x)| \leq \frac{1}{24},$$

since $|\sin x| \leq 1$ *for all* x. *This bound on* $|R_n(x)|$ *serves as an upper bound for the error in the approximation of* $\sin x$ *by* $P_3(x)$ *for* $x \in [-1, 1]$. □

Exercise A.6.2 *On the interval* $[-1, 1]$, *plot* $f(x) = \sin x$ *and its Taylor polynomial of degree 3 centered at* $x_0 = 0$, $P_3(x)$. *How do they compare? In a separate figure window, plot the error* $R_3(x) = f(x) - P_3(x)$ *on* $[-1, 1]$, *and also plot the lines* $y = \pm\frac{1}{24}$, *corresponding to the upper bound on* $|R_3(x)|$ *from the preceding example. Confirm that the error actually does satisfy this bound.*

Example A.6.4 *Let* $f(x) = e^x$. *Then*

$$f(x) = P_2(x) + R_2(x),$$

where

$$P_2(x) = 1 + x + \frac{x^2}{2},$$

and

$$R_2(x) = \frac{x^3}{6}e^{\xi(x)},$$

where $\xi(x)$ *is between 0 and* x. *The polynomial* $P_2(x)$ *is the 2nd Maclaurin polynomial of* e^x, *or the 2nd Taylor polynomial with center* $x_0 = 0$.

If $x > 0$, *then* $R_2(x)$ *can become quite large, whereas its magnitude is much smaller if* $x < 0$. *Therefore, one method of computing* e^x *using a Maclaurin polynomial is to use the nth Maclaurin polynomial* $P_n(x)$ *of* e^x *when* $x < 0$, *where n is chosen sufficiently large so that* $R_n(x)$ *is small for the given value of* x. *If* $x > 0$, *then we instead compute* e^{-x} *using the nth Maclaurin polynomial for* e^{-x}, *which is given by*

$$P_n(x) = 1 - x + \frac{x^2}{2} - \frac{x^3}{6} + \cdots + \frac{(-1)^n x^n}{n!},$$

and then obtaining an approximation to e^x *by taking the reciprocal of our computed value of* e^{-x}.
□

Example A.6.5 *Let* $f(x) = x^2$. *Then, for any real number* x_0,

$$f(x) = P_1(x) + R_1(x),$$

where

$$P_1(x) = x_0^2 + 2x_0(x - x_0) = 2x_0x - x_0^2,$$

and

$$R_1(x) = (x - x_0)^2.$$

Note that the remainder does not include a "mystery point" $\xi(x)$ *since the 2nd derivative of* x^2 *is only a constant. The linear function* $P_1(x)$ *describes the tangent line to the graph of* $f(x)$ *at the point* $(x_0, f(x_0))$. *If* $x_0 = 1$, *then we have*

$$P_1(x) = 2x - 1,$$

and

$$R_1(x) = (x - 1)^2.$$

We can see that near $x = 1$, $P_1(x)$ *is a reasonable approximation to* x^2, *since the error in this approximation, given by* $R_1(x)$, *would be small in this case.* □

Taylor's theorem can be generalized to functions of several variables, using partial derivatives. Here, we consider the case of two independent variables.

Theorem A.6.6 (Taylor's Theorem in Two Variables) *Let* $f(t, y)$ *be* $(n + 1)$ *times continuously differentiable on a convex set* D, *and let* $(t_0, y_0) \in D$. *Then, for every* $(t, y) \in D$, *there exists* ξ *between* t_0 *and* t, *and* μ *between* y_0 *and* y, *such that*

$$f(t, y) = P_n(t, y) + R_n(t, y),$$

where $P_n(t, y)$ *is the* nth Taylor polynomial *of* f *about* (t_0, y_0),

$$\begin{aligned} P_n(t,y) &= f(t_0,y_0) + \left[(t-t_0)\frac{\partial f}{\partial t}(t_0,y_0) + (y-y_0)\frac{\partial f}{\partial y}(t_0,y_0)\right] + \\ &\quad \left[\frac{(t-t_0)^2}{2}\frac{\partial^2 f}{\partial t^2}(t_0,y_0) + (t-t_0)(y-y_0)\frac{\partial^2 f}{\partial t \partial y}(t_0,y_0) + \frac{(y-y_0)^2}{2}\frac{\partial^2 f}{\partial y^2}(t_0,y_0)\right] + \\ &\quad \cdots + \left[\frac{1}{n!}\sum_{j=0}^{n}\begin{pmatrix} n \\ j \end{pmatrix}(t-t_0)^{n-j}(y-y_0)^j\frac{\partial^n f}{\partial t^{n-j}\partial y^j}(t_0,y_0)\right], \end{aligned}$$

and $R_n(t, y)$ *is the* remainder term *associated with* $P_n(t, y)$,

$$R_n(t,y) = \frac{1}{(n+1)!}\sum_{j=0}^{n+1}\begin{pmatrix} n+1 \\ j \end{pmatrix}(t-t_0)^{n+1-j}(y-y_0)^j\frac{\partial^{n+1} f}{\partial t^{n+1-j}\partial y^j}(\xi,\mu).$$

Appendix B

Review of Linear Algebra

B.1 Matrices

Writing a system of equations can be quite tedious. Therefore, we instead represent a system of linear equations using a *matrix*, which is an array of elements, or entries. We say that a matrix A is $m \times n$ if it has m rows and n columns, and we denote the element in row i and column j by a_{ij}. We also denote the matrix A by $[a_{ij}]$.

With this notation, a general system of m equations with n unknowns can be represented using a matrix A that contains the coefficients of the equations, a vector $\mathbf{x}$ that contains the unknowns, and a vector $\mathbf{b}$ that contains the quantities on the right-hand sides of the equations. Specifically,

$$A = \left[\begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \vdots & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{array}\right], \quad \mathbf{x} = \left[\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array}\right], \quad \mathbf{b} = \left[\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_m \end{array}\right].$$

Note that the vectors $\mathbf{x}$ and $\mathbf{b}$ are represented by *column* vectors.

Example The coefficients in the linear system

$$\begin{array}{rcl} 3x_1 + 2x_2 & = & 4, \\ -x_1 + 5x_2 & = & -3 \end{array}$$

can be represented by the matrix

$$A = \left[\begin{array}{rr} 3 & 2 \\ -1 & 5 \end{array}\right].$$

The coefficient of x_2 in the first equation is represented by the entry in the first row and second column of A, which is $a_{12} = 2$. □

B.2 Vector Spaces

Matrices are much more than notational conveniences for writing systems of linear equations. A matrix A can also be used to represent a linear function f_A whose domain and range are both sets of vectors called *vector spaces*. A vector space over a *field* (such as the field of real or complex numbers) is a set of vectors, together with two operations: addition of vectors, and multiplication of a vector by a scalar from the field.

Specifically, if $\mathbf{u}$ and $\mathbf{v}$ are vectors belonging to a vector space V over a field F, then the *sum* of $\mathbf{u}$ and $\mathbf{v}$, denoted by $\mathbf{u}+\mathbf{v}$, is a vector in V, and the *scalar product* of $\mathbf{u}$ with a scalar α in F, denoted by $\alpha\mathbf{u}$, is also a vector in V. These operations have the following properties:

- *Commutativity*: For any vectors $\mathbf{u}$ and $\mathbf{v}$ in V,
$$\mathbf{u}+\mathbf{v}=\mathbf{v}+\mathbf{u}$$
- *Associativity*: For any vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ in V,
$$(\mathbf{u}+\mathbf{v})+\mathbf{w}=\mathbf{u}+(\mathbf{v}+\mathbf{w})$$
- *Identity element for vector addition*: There is a vector $\mathbf{0}$, known as the *zero vector*, such that for any vector $\mathbf{u}$ in V,
$$\mathbf{u}+\mathbf{0}=\mathbf{0}+\mathbf{u}=\mathbf{u}$$
- *Additive inverse*: For any vector $\mathbf{u}$ in V, there is a unique vector $-\mathbf{u}$ in V such that
$$\mathbf{u}+(-\mathbf{u})=-\mathbf{u}+\mathbf{u}=\mathbf{0}$$
- *Distributivity over vector addition*: For any vectors $\mathbf{u}$ and $\mathbf{v}$ in V, and scalar α in F,
$$\alpha(\mathbf{u}+\mathbf{v})=\alpha\mathbf{u}+\alpha\mathbf{v}$$
- *Distributivity over scalar multiplication*: For any vector $\mathbf{u}$ in V, and scalars α and β in F,
$$(\alpha+\beta)\mathbf{u}=\alpha\mathbf{u}+\beta\mathbf{u}$$
- *Associativity of scalar multiplication*: For any vector $\mathbf{u}$ in V and any scalars α and β in F,
$$\alpha(\beta\mathbf{u})=(\alpha\beta)\mathbf{u}$$
- *Identity element for scalar multiplication*: For any vector $\mathbf{u}$ in V,
$$1\mathbf{u}=\mathbf{u}$$

Example Let V be the vector space $\mathbb{R}^3$. The vector addition operation on V consists of adding corresponding components of vectors, as in

$$\begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}+\begin{bmatrix} -2 \\ 4 \\ 5 \end{bmatrix}=\begin{bmatrix} 1 \\ 4 \\ 6 \end{bmatrix}.$$

The scalar multiplication operation consists of scaling each component of a vector by a scalar:

$$\frac{1}{2}\begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}=\begin{bmatrix} \frac{3}{2} \\ 0 \\ \frac{1}{2} \end{bmatrix}.$$

□

B.3 Subspaces

Before we can explain how matrices can be used to easily describe linear transformations, we must introduce some important concepts related to vector spaces.

A *subspace* of a vector space V is a subset of V that is, itself, a vector space. In particular, a subset S of V is also a subsapce if it is *closed* under the operations of vector addition and scalar multiplication. That is, if $\mathbf{u}$ and $\mathbf{v}$ are vectors in S, then the vectors $\mathbf{u}+\mathbf{v}$ and $\alpha\mathbf{u}$, where α is any scalar, must also be in S. In particular, S cannot be a subspace unless it includes the zero vector.

Example The set S of all vectors in $\mathbb{R}^3$ of the form

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ 0 \end{bmatrix},$$

where $x_1, x_2 \in \mathbb{R}$, is a subspace of $\mathbb{R}^3$, as the sum of any two vectors in S, or a scalar multiple of any vector in S, must have a third component of zero, and therefore is also in S.

On the other hand, the set $\tilde{S}$ consisting of all vectors in $\mathbb{R}^3$ that have a third component of 1 is *not* a subspace of $\mathbb{R}^3$, as the sum of vectors in $\tilde{S}$ will not have a third component of 1, and therefore will not be in $\tilde{S}$. That is, $\tilde{S}$ is not *closed* under addition. $\Box$

B.4 Linear Independence and Bases

Often a vector space or subspace can be characterized as the set of all vectors that can be obtained by adding and/or scaling members of a given set of specific vectors. For example, $\mathbb{R}^2$ can be described as the set of all vectors that can be obtained by adding and/or scaling the vectors

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

These vectors comprise what is known as the *standard basis* of $\mathbb{R}^2$.

More generally, given a set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$ from a vector space V, a vector $\mathbf{v} \in V$ is called a *linear combination* of $\mathbf{v}_1$, $\mathbf{v}_2$, …, $\mathbf{v}_k$ if there exist constants c_1, c_2, …, c_k such that

$$\mathbf{v} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_k\mathbf{v}_k = \sum_{i=1}^{k} c_i\mathbf{v}_i.$$

We then define the *span* of $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$, denoted by $\operatorname{span}\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$, to be the set of *all* linear combinations of $\mathbf{v}_1$, $\mathbf{v}_2$, …, $\mathbf{v}_k$. From the definition of a linear combination, it follows that this set is a subspace of V.

Example Let

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix}.$$

Then the vector

$$\mathbf{v} = \begin{bmatrix} 6 \\ 10 \\ 2 \end{bmatrix}$$

is a linear combination of $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$, as

$$\mathbf{v} = \mathbf{v}_1 + 2\mathbf{v}_2 + \mathbf{v}_3.$$

□

When a subspace is defined as the span of a set of vectors, it is helpful to know whether the set includes any vectors that are, in some sense, redundant, for if this is the case, the description of the subspace can be simplified. To that end, we say that a set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$ is *linearly independent* if the equation

$$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_k\mathbf{v}_k = \mathbf{0}$$

holds if and only if $c_1 = c_2 = \cdots = c_k = 0$. Otherwise, we say that the set is *linearly dependent.* If the set is linearly independent, then any vector $\mathbf{v}$ in the span of the set is a *unique* linear combination of members of the set; that is, there is only one way to choose the coefficients of a linear combination that is used to obtain $\mathbf{v}$.

Example The subspace S of $\mathbb{R}^3$ defined by

$$S = \left\{ \left[\begin{array}{c} x_1 \\ x_2 \\ 0 \end{array}\right] \middle| \; x_1, x_2 \in \mathbb{R} \right\}$$

can be described as

$$S = \operatorname{span}\left\{ \left[\begin{array}{c} 1 \\ 0 \\ 0 \end{array}\right], \left[\begin{array}{c} 0 \\ 1 \\ 0 \end{array}\right] \right\}$$

or

$$S = \operatorname{span}\left\{ \left[\begin{array}{c} 1 \\ 1 \\ 0 \end{array}\right], \left[\begin{array}{r} 1 \\ -1 \\ 0 \end{array}\right] \right\},$$

but

$$S \neq \operatorname{span}\left\{ \left[\begin{array}{c} 1 \\ 1 \\ 0 \end{array}\right], \left[\begin{array}{r} -1 \\ -1 \\ 0 \end{array}\right] \right\},$$

as these vectors only span the subspace of vectors whose first two components are equal, and whose third component is zero, which does not account for every vector in S. It should be noted that the two vectors in the third set are linearly dependent, while the pairs of vectors in the previous two sets are linearly independent. □

Example The vectors

$$\mathbf{v}_1 = \left[\begin{array}{c} 1 \\ 0 \\ 1 \end{array}\right], \quad \mathbf{v}_2 = \left[\begin{array}{c} 1 \\ 1 \\ 0 \end{array}\right]$$

are linearly independent. It follows that the only way in which the vector

$$\mathbf{v} = \left[\begin{array}{c} 3 \\ 1 \\ 2 \end{array}\right]$$

can be expressed as a linear combination of $\mathbf{v}_1$ and $\mathbf{v}_2$ is

$$\mathbf{v} = 2\mathbf{v}_1 + \mathbf{v}_2.$$

On the other hand, if

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix},$$

then, because $\mathbf{v}_1$ and $\mathbf{v}_2$ are linearly dependent, any linear combination of the form $c_1\mathbf{v}_1 + c_2\mathbf{v}_2$, such that $c_1 + 2c_2 = 3$, will equal $\mathbf{v}$. □

Given a vector space V, if there exists a set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$ such that V is the span of $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$, *and* $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$ is linearly independent, then we say that $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$ is a *basis* of V. Any basis of V must have the same number of elements, k. We call this number the *dimension* of V, which is denoted by $\dim(V)$.

Example The standard basis of $\mathbb{R}^3$ is

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

The set

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

is also a basis for $\mathbb{R}^3$, as it consists of three linearly independent vectors, and the dimension of $\mathbb{R}^3$ is three. □

B.5 Linear Transformations

A function $f_A : V \to W$, whose domain V and range W are vector spaces over a field F, is a *linear transformation* if it has the properties

$$f_A(\mathbf{x} + \mathbf{y}) = f_A(\mathbf{x}) + f_A(\mathbf{y}), \quad f_A(\alpha\mathbf{x}) = \alpha f_A(\mathbf{x}),$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors in V and α is a scalar from F. If V and W are the same vector space, then we say that f_A is a *linear operator* on V.

B.5.1 The Matrix of a Linear Transformation

If V is a vector space of dimension n over a field, such as $\mathbb{R}^n$ or $\mathbb{C}^n$, and W is a vector space of dimension m, then a linear transformation f_A with domain V and range W can be represented by an $m \times n$ matrix A whose entries belong to the field.

Suppose that the set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ is a *basis* for V, and the set $\{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m\}$ is a basis for W. Then, a_{ij} is the scalar by which $\mathbf{w}_i$ is multiplied when applying the function f_A to the vector $\mathbf{v}_j$. That is,

$$f_A(\mathbf{v}_j) = a_{1j}\mathbf{w}_1 + a_{2j}\mathbf{w}_2 + \cdots + a_{mj}\mathbf{w}_m = \sum_{i=1}^{m} a_{ij}\mathbf{w}_i.$$

In other words, the jth column of A describes the *image* under f_A of the vector $\mathbf{v}_j$, in terms of the coefficients of $f_A(\mathbf{v}_j)$ in the basis $\{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m\}$.

If V and W are spaces of real or complex vectors, then, by convention, the bases $\{\mathbf{v}_j\}_{j=1}^{n}$ and $\{\mathbf{w}_i\}_{i=1}^{m}$ are each chosen to be the *standard basis* for $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. The jth vector in the standard basis is a vector whose components are all zero, except for the jth component, which is equal to one. These vectors are called the *standard basis vectors* of an n-dimensional space of real or complex vectors, and are denoted by $\mathbf{e}_j$. From this point on, we will generally assume that V is $\mathbb{R}^n$, and that the field is $\mathbb{R}$, for simplicity.

Example The standard basis for $\mathbb{R}^3$ consists of the vectors

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

□

B.5.2 Matrix-Vector Multiplication

To describe the action of A on a general vector $\mathbf{x}$ from V, we can write

$$\mathbf{x} = x_1\mathbf{e}_1 + x_2\mathbf{e}_2 + \cdots + x_n\mathbf{e}_n = \sum_{j=1}^{n} x_j\mathbf{e}_j.$$

Then, because A represents a linear function,

$$f_A(\mathbf{x}) = \sum_{j=1}^{n} x_j f_A(\mathbf{e}_j) = \sum_{j=1}^{n} x_j \mathbf{a}_j,$$

where $\mathbf{a}_j$ is the jth column of A.

We define the vector $\mathbf{y} = f_A(\mathbf{x})$ above to be the *matrix-vector product* of A and $\mathbf{x}$, which we denote by $A\mathbf{x}$. Each element of the vector $\mathbf{y} = A\mathbf{x}$ is given by

$$y_i = [A\mathbf{x}]_i = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = \sum_{j=1}^{n} a_{ij}x_j.$$

From this definition, we see that the jth column of A is equal to the matrix-vector product $A\mathbf{e}_j$.

Example Let

$$A = \begin{bmatrix} 3 & 0 & -1 \\ 1 & -4 & 2 \\ 5 & 1 & -3 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix}.$$

Then

$$A\mathbf{x} = 10\begin{bmatrix} 3 \\ 1 \\ 5 \end{bmatrix} + 11\begin{bmatrix} 0 \\ -4 \\ 1 \end{bmatrix} + 12\begin{bmatrix} -1 \\ 2 \\ -3 \end{bmatrix} = \begin{bmatrix} 18 \\ -10 \\ 25 \end{bmatrix}.$$

We see that $A\mathbf{x}$ is a linear combination of the columns of A, with the coefficients of the linear combination obtained from the components of $\mathbf{x}$. □

Example Let

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

Then the matrix-vector product of A and $\mathbf{x}$ is

$$\mathbf{y} = A\mathbf{x} = \begin{bmatrix} 3(1) + 1(-1) \\ 1(1) + 0(-1) \\ 2(1) + 4(-1) \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ -2 \end{bmatrix}.$$

□

B.5.3 Special Subspaces

Let A be an $m \times n$ matrix. Then the *range* of A, denoted by $\text{ran}(A)$, is the set of all vectors of the form $\mathbf{y} = A\mathbf{x}$, where $\mathbf{x} \in \mathbb{R}^n$. It follows that $\text{ran}(A)$ is the span of the columns of A, which is also called the *column space* of A.

The dimension of $\text{ran}(A)$ is called the *column rank* of A. Similarly, the dimension of the *row space* of A is called the *rank, row* of A. It can be shown that the row rank and column rank are equal; this common value is simply called the *rank* of A, and is denoted by $\text{rank}(A)$. We say that A is *rank-deficient* if $\text{rank}(A) < \min\{m, n\}$; otherwise, we say that A has *full rank.* It is interesting to note that any outer product of vectors has rank one.

The *null space* of A, denoted by $\text{null}(A)$, is the set of all vectors $\mathbf{x} \in \mathbb{R}^n$ such that $A\mathbf{x} = \mathbf{0}$. Its dimension is called the *nullity* of A. It can be shown that for an $m \times n$ matrix A,

$$\dim(\text{null}(A)) + \text{rank}(A) = n.$$

B.6 Matrix-Matrix Multiplication

It follows from this definition that a general system of m linear equations in n unknowns can be described in matrix-vector form by the equation

$$A\mathbf{x} = \mathbf{b},$$

where $A\mathbf{x}$ is a matrix-vector product of the $m \times n$ coefficient matrix A and the vector of unknowns $\mathbf{x}$, and $\mathbf{b}$ is the vector of right-hand side values.

Of course, if $m = n = 1$, the system of equations $A\mathbf{x} = \mathbf{b}$ reduces to the scalar linear equation $ax = b$, which has the solution $x = a^{-1}b$, provided that $a \neq 0$. As a^{-1} is the unique number such that $a^{-1}a = aa^{-1} = 1$, it is desirable to generalize the concepts of multiplication and identity element to square matrices, for which $m = n$.

The matrix-vector product can be used to define the *composition* of linear functions represented by matrices. Let A be an $m \times n$ matrix, and let B be an $n \times p$ matrix. Then, if $\mathbf{x}$ is a vector of length p, and $\mathbf{y} = B\mathbf{x}$, then we have

$$A\mathbf{y} = A(B\mathbf{x}) = (AB)\mathbf{x} = C\mathbf{x},$$

where C is an $m \times p$ matrix with entries

$$C_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}.$$

We define the *matrix product* of A and B to be the matrix $C = AB$ with entries defined in this manner. It should be noted that the product BA is not defined, unless $m = p$. Even if this is the case, in general, $AB \neq BA$. That is, matrix multiplication is not *commutative*. However, matrix multiplication is *associative*, meaning that if A is $m \times n$, B is $n \times p$, and C is $p \times k$, then $A(BC) = (AB)C$.

Example Consider the 2×2 matrices

$$A = \left[\begin{array}{cc} 1 & -2 \\ -3 & 4 \end{array}\right], \quad B = \left[\begin{array}{cc} -5 & 6 \\ 7 & -8 \end{array}\right].$$

Then

$$\begin{aligned} AB &= \left[\begin{array}{cc} 1 & -2 \\ -3 & 4 \end{array}\right]\left[\begin{array}{cc} -5 & 6 \\ 7 & -8 \end{array}\right] \\ &= \left[\begin{array}{cc} 1(-5) - 2(7) & 1(6) - 2(-8) \\ -3(-5) + 4(7) & -3(6) + 4(-8) \end{array}\right] \\ &= \left[\begin{array}{cc} -19 & 22 \\ 43 & -50 \end{array}\right], \end{aligned}$$

whereas

$$\begin{aligned} BA &= \left[\begin{array}{cc} -5 & 6 \\ 7 & -8 \end{array}\right]\left[\begin{array}{cc} 1 & -2 \\ -3 & 4 \end{array}\right] \\ &= \left[\begin{array}{cc} -5(1) + 6(-3) & -5(-2) + 6(4) \\ 7(1) - 8(-3) & 7(-2) - 8(4) \end{array}\right] \\ &= \left[\begin{array}{cc} -23 & 34 \\ 31 & -46 \end{array}\right]. \end{aligned}$$

We see that $AB \neq BA$. □

Example If

$$A = \left[\begin{array}{cc} 3 & 1 \\ 1 & 0 \\ 2 & 4 \end{array}\right], \quad B = \left[\begin{array}{cc} 1 & 5 \\ 4 & -1 \end{array}\right],$$

then the matrix-matrix product of A and B is

$$C = AB = \left[\begin{array}{cc} 3(1) + 1(4) & 3(5) + 1(-1) \\ 1(1) + 0(4) & 1(5) + 0(-1) \\ 2(1) + 4(4) & 2(5) + 4(-1) \end{array}\right] = \left[\begin{array}{cc} 7 & 14 \\ 1 & 5 \\ 18 & 6 \end{array}\right].$$

It does not make sense to compute BA, because the dimensions are incompatible. □

B.7 Other Fundamental Matrix Operations

B.7.1 Vector Space Operations

The set of all matrices of size $m \times n$, for fixed m and n, is itself a vector space of dimension mn. The operations of vector addition and scalar multiplication for matrices are defined as follows: If A and B are $m \times n$ matrices, then the sum of A and B, denoted by $A + B$, is the $m \times n$ matrix C with entries

$$c_{ij} = a_{ij} + b_{ij}.$$

If α is a scalar, then the product of α and an $m \times n$ matrix A, denoted by αA, is the $m \times n$ matrix B with entries

$$b_{ij} = \alpha a_{ij}.$$

It is natural to identify $m \times n$ matrices with vectors of length mn, in the context of these operations.

Matrix addition and scalar multiplication have properties analogous to those of vector addition and scalar multiplication. In addition, matrix multiplication has the following properties related to these operations. We assume that A is an $m \times n$ matrix, B and D are $n \times k$ matrices, and α is a scalar.

- Distributivity: $A(B + D) = AB + AD$
- Commutativity of scalar multiplication: $\alpha(AB) = (\alpha A)B = A(\alpha B)$

B.7.2 The Transpose of a Matrix

An $n \times n$ matrix A is said to be *symmetric* if $a_{ij} = a_{ji}$ for $i, j = 1, 2, \ldots, n$. The $n \times n$ matrix B whose entries are defined by $b_{ij} = a_{ji}$ is called the *transpose* of A, which we denote by A^T. Therefore, A is symmetric if $A = A^T$. More generally, if A is an $m \times n$ matrix, then A^T is the $n \times n$ matrix B whose entries are defined by $b_{ij} = a_{ji}$. The transpose has the following properties:

- $(A^T)^T = A$
- $(A + B)^T = A^T + B^T$
- $(AB)^T = B^T A^T$

Example If

$$A = \begin{bmatrix} 3 & 1 \\ 1 & 0 \\ 2 & 4 \end{bmatrix},$$

then

$$A^T = \begin{bmatrix} 3 & 1 & 2 \\ 1 & 0 & 4 \end{bmatrix}.$$

□

Example Let A be the matrix from a previous example,

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

Then

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

It follows that

$$A + A^T = \begin{bmatrix} 3+3 & 0+1 & -1+5 \\ 1+0 & -4-4 & 2+1 \\ 5-1 & 1+2 & -3-3 \end{bmatrix} = \begin{bmatrix} 6 & 1 & 4 \\ 1 & -8 & 3 \\ 4 & 3 & -6 \end{bmatrix}.$$

This matrix is symmetric. This can also be seen by the properties of the transpose, since

$$(A + A^T)^T = A^T + (A^T)^T = A^T + A = A + A^T.$$

□

Example The matrix

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

is symmetric, while

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

is *skew-symmetric*, meaning that $A^T = -A$. □

B.7.3 Inner and Outer Products

We now define several other operations on matrices and vectors that will be useful in our study of numerical linear algebra. For simplicity, we work with real vectors and matrices.

Given two vectors $\mathbf{x}$ and $\mathbf{y}$ in $\mathbb{R}^n$, the *dot product*, or *inner product*, of $\mathbf{x}$ and $\mathbf{y}$ is the scalar

$$\mathbf{x}^T\mathbf{y} = x_1y_1 + x_2y_2 + \cdots + x_ny_n = \sum_{i=1}^{n} x_iy_i,$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Note that $\mathbf{x}$ and $\mathbf{y}$ must both be defined to be *column* vectors, and they must have the same length. If $\mathbf{x}^T\mathbf{y} = 0$, then we say that $\mathbf{x}$ and $\mathbf{y}$ are *orthogonal*.

Let $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$, where m and n are not necessarily equal. The term “inner product” suggests the existence of another operation called the *outer product*, which is defined by

$$\mathbf{x}\mathbf{y}^T = \begin{bmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_n \\ x_2y_1 & x_2y_2 & \cdots & x_2y_n \\ \vdots & & \vdots & \\ x_my_1 & x_my_2 & \cdots & x_my_n \end{bmatrix}.$$

Note that whereas the inner product is a scalar, the outer product is an $m \times n$ matrix.

Example Let

$$\mathbf{x} = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 4 \\ -1 \\ 3 \end{bmatrix}.$$

Then the inner (dot) product of $\mathbf{x}$ and $\mathbf{y}$ is

$$\mathbf{x}^T\mathbf{y} = 1(4) + 0(-1) + 2(3) = 10,$$

while the outer product of $\mathbf{x}$ and $\mathbf{y}$ is

$$\mathbf{x}\mathbf{y}^T = \begin{bmatrix} 1(4) & 1(-1) & 1(3) \\ 0(4) & 0(-1) & 0(3) \\ 2(4) & 2(-1) & 2(3) \end{bmatrix} = \begin{bmatrix} 4 & -1 & 3 \\ 0 & 0 & 0 \\ 8 & -2 & 6 \end{bmatrix}.$$

□

Example Let

$$A = \begin{bmatrix} 1 & -3 & 7 \\ 2 & 5 & -8 \\ 4 & -6 & -9 \end{bmatrix}.$$

To change a_{11} from 1 to 10, we can perform the outer product update $B = A + (10-1)\mathbf{e}_1\mathbf{e}_1^T$. Similary, the outer product update $C = B + 5\mathbf{e}_2\mathbf{e}_1^T$ adds 5 to b_{21}, resulting in the matrix

$$C = \begin{bmatrix} 10 & -3 & 7 \\ 7 & 5 & -8 \\ 4 & -6 & -9 \end{bmatrix}.$$

Note that

$$\mathbf{e}_2\mathbf{e}_1^T = \begin{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} 1 & \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} 0 & \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

□

B.7.4 Hadamard Product

If $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, the *Hadamard product*, or *componentwise product*, of $\mathbf{x}$ and $\mathbf{y}$, denoted by $\mathbf{x} \circ \mathbf{y}$ or $\mathbf{x}.*\mathbf{y}$, is the vector $\mathbf{z}$ obtained by multiplying corresponding components of $\mathbf{x}$ and $\mathbf{y}$. That is, if $\mathbf{z} = \mathbf{x}.*\mathbf{y}$, then $z_i = x_i y_i$, for $i = 1, 2, \ldots, n$.

Example If $\mathbf{x} = \begin{bmatrix} 1 & -2 \end{bmatrix}^T$ and $\mathbf{y} = \begin{bmatrix} -3 & 4 \end{bmatrix}^T$, then

$$\mathbf{x}^T\mathbf{y} = 1(-3) + (-2)4 = -11, \quad \mathbf{x}\mathbf{y}^T = \begin{bmatrix} 1(-3) & 1(4) \\ -2(-3) & -2(4) \end{bmatrix} = \begin{bmatrix} -3 & 4 \\ 6 & -8 \end{bmatrix},$$

and

$$\mathbf{x}.*\mathbf{y} = \begin{bmatrix} 1(-3) \\ -2(4) \end{bmatrix} = \begin{bmatrix} -3 \\ -8 \end{bmatrix}.$$

□

B.7.5 Kronecker Product

B.7.6 Partitioning

It is useful to describe matrices as collections of row or column vectors. Specifically, a *row partition* of an $m \times n$ matrix A is a description of A as a "stack" of row vectors $\mathbf{r}_1^T$, $\mathbf{r}_2^T$, …, $\mathbf{r}_m^T$. That is,

$$A = \left[\begin{array}{c} \mathbf{r}_1^T \\ \mathbf{r}_2^T \\ \vdots \\ \mathbf{r}_m^T \end{array} \right].$$

On the other hand, we can view A is a "concatenation" of column vectors $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n$:

$$A = \left[\begin{array}{cccc} \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \end{array} \right].$$

This description of A is called a *column partition*.

Example If

$$A = \left[\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array} \right],$$

then a column partitioning of A is

$$A = \left[\begin{array}{cc} \mathbf{c}_1 & \mathbf{c}_2 \end{array} \right], \quad \mathbf{c}_1 = \left[\begin{array}{c} 1 \\ 3 \end{array} \right], \quad \mathbf{c}_2 = \left[\begin{array}{c} 2 \\ 4 \end{array} \right].$$

A row partitioning of A is

$$A = \left[\begin{array}{c} \mathbf{r}_1^T \\ \mathbf{r}_2^T \end{array} \right], \quad \mathbf{r}_1 = \left[\begin{array}{c} 1 \\ 2 \end{array} \right], \quad \mathbf{r}_2 = \left[\begin{array}{c} 3 \\ 4 \end{array} \right].$$

□

B.8 Understanding Matrix-Matrix Multiplication

The fundamental operation of matrix-matrix multiplication can be understood in three different ways, based on other operations that can be performed on matrices and vectors. Let A be an $m \times n$ matrix, and B be an $n \times p$ matrix, in which case $C = AB$ is an $m \times p$ matrix. We can then view the computation of C in the following ways:

- Dot product: each entry c_{ij} is the dot product of the ith row of A and the jth column of B.
- Matrix-vector multiplication: the jth column of C is a linear combination of the columns of A, where the coefficients are obtained from the jth column of B. That is, if

  $$C = \left[\begin{array}{cccc} \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_p \end{array} \right], \quad B = \left[\begin{array}{cccc} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_p \end{array} \right]$$

  are column partitions of C and B, then $\mathbf{c}_j = A\mathbf{b}_j$, for $j = 1, 2, \ldots, p$.

- Outer product: given the partitions

$$A = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \end{bmatrix}, \quad B = \begin{bmatrix} \mathbf{b}_1^T \\ \mathbf{b}_2^T \\ \cdots \\ \mathbf{b}_n^T \end{bmatrix},$$

  we can write

$$C = \mathbf{a}_1\mathbf{b}_1^T + \mathbf{a}_2\mathbf{b}_2^T + \cdots + \mathbf{a}_n\mathbf{b}_n^T = \sum_{i=1}^n \mathbf{a}_i\mathbf{b}_i^T.$$

  That is, C is a sum of *outer product updates.*

B.8.1 The Identity Matrix

When $n = 1$, the identity element of 1×1 matrices, the number 1, is the unique number such that $a(1) = 1(a) = a$ for any number a. To determine the identity element for $n \times n$ matrices, we seek a matrix I such that $AI = IA = A$ for any $n \times n$ matrix A. That is, we must have

$$\sum_{k=1}^n a_{ik}I_{kj} = a_{ij}, \quad i, j = 1, \ldots, n.$$

This can only be the case for *any* matrix A if $I_{jj} = 1$ for $j = 1, 2, \ldots, n$, and $I_{ij} = 0$ when $i \neq j$. We call this matrix the *identity matrix*

$$I = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & 0 & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & 0 & 1 & 0 \\ 0 & \cdots & \cdots & 0 & 1 \end{bmatrix}.$$

Note that the jth column of I is the standard basis vector $\mathbf{e}_j$.

B.8.2 The Inverse of a Matrix

Given an $n \times n$ matrix A, it is now natural to ask whether it is possible to find an $n \times n$ matrix B such that $AB = BA = I$. Such a matrix, if it exists, would then serve as the *inverse* of A, in the sense of matrix multiplication. We denote this matrix by A^{-1}, just as we denote the multiplicative inverse of a nonzero number a by a^{-1}. If the inverse of A exists, we say that A is *invertible* or *nonsingular*; otherwise, we say that A is *singular*. If A^{-1} exists, then we can use it to describe the solution of the system of linear equations $A\mathbf{x} = \mathbf{b}$, for

$$A^{-1}A\mathbf{x} = (A^{-1}A)\mathbf{x} = I\mathbf{x} = \mathbf{x} = A^{-1}\mathbf{b},$$

which generalizes the solution $x = a^{-1}b$ of a single linear equation in one unknown.

However, just as we can use the inverse to describe the solution to a system of linear equations, we can use systems of linear equations to characterize the inverse. Because A^{-1} satisfies $AA^{-1} = I$, it follows from multiplication of both sides of this equation by the jth standard basis vector $\mathbf{e}_j$ that

$$A\mathbf{b}_j = \mathbf{e}_j, \quad j = 1, 2, \ldots, n,$$

where $\mathbf{b}_j = A^{-1}\mathbf{e}_j$ is the jth column of $B = A^{-1}$. That is, we can compute A^{-1} by solving n systems of linear equations of the form $A\mathbf{b}_j = \mathbf{e}_j$, using a method such as Gaussian elimination and back substitution. If Gaussian elimination fails due to the inability to obtain a nonzero pivot element for each column, then A^{-1} does not exist, and we conclude that A is singular.

The inverse of a nonsingular matrix A has the following properties:

- A^{-1} is unique.
- A^{-1} is nonsingular, and $(A^{-1})^{-1} = A$.
- If B is also a nonsingular $n \times n$ matrix, then $(AB)^{-1} = B^{-1}A^{-1}$.
- $(A^{-1})^T = (A^T)^{-1}$. It is common practice to denote the transpose of A^{-1} by A^{-T}.

Because the set of all $n \times n$ matrices has an identity element, matrix multiplication is associative, and each nonsingular $n \times n$ matrix has a unique inverse with respect to matrix multiplication that is also an $n \times n$ nonsingular matrix, this set forms a *group*, which is denoted by $GL(n)$, the *general linear group*.

B.9 Triangular and Diagonal Matrices

There are certain types of matrices for which the fundamental problems of numerical linear algebra, solving systems of linear equations or computing eigenvalues and eigenvectors, are relatively easy to solve. We now discuss a few such types.

Let A be an $m \times n$ matrix. We define the *main diagonal* of A to be the entries $a_{11}, a_{22}, \ldots, a_{pp}$, where $p = \min\{m, n\}$. That is, the main diagonal consists of all entries for which the row index and column index are equal. We then say that A is a *diagonal matrix* if the only nonzero entries of A lie on the main diagonal. That is, A is a diagonal matrix if $a_{ij} = 0$ whenever $i \neq j$.

We say that A is *upper triangular* if $a_{ij} = 0$ whenever $i > j$. That is, all nonzero entries of A are confined to the "upper triangle" of A, which consists of all entries on or "above" the main diagonal. Similarly, we say that A is *lower triangular* if $a_{ij} = 0$ whenever $i < j$. Such a matrix has all of its nonzero entries on or below the main diagonal.

We will see that a system of linear equations of the form $A\mathbf{x} = \mathbf{b}$ is easily solved if A is an upper or lower triangular matrix, and that the eigenvalues of a square matrix A are easy to obtain if A is triangular. As such, certain methods for solving both kinds of problems proceed by reducing A to triangular form.

Example The matrices

$$U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

are upper triangular, lower triangular, and diagonal, respectively. □

B.10 Determinants

We previously learned that a 2×2 matrix A is invertible if and only if the quantity $a_{11}a_{22} - a_{12}a_{21}$ is nonzero. This generalizes the fact that a 1×1 matrix a is invertible if and only if its single entry, $a_{11} = a$, is nonzero. We now discuss the generalization of this determination of invertibility to general square matrices.

The *determinant* of an $n \times n$ matrix A, denoted by $\det(A)$ or $|A|$, is defined as follows:

- If $n = 1$, then $\det(A) = a_{11}$.
- If $n > 1$, then $\det(A)$ is recursively defined by

  $$\det(A) = \sum_{j=1}^{n} a_{ij}(-1)^{i+j} \det(M_{ij}), \quad 1 \leq i \leq n,$$

  where M_{ij}, called a *minor* of A, is the matrix obtained by removing row i and column j of A.
- Alternatively,

  $$\det(A) = \sum_{i=1}^{n} a_{ij}(-1)^{i+j} \det(M_{ij}), \quad 1 \leq j \leq n.$$

The matrix $A_{ij} = (-1)^{i+j} M_{ij}$ is called a *cofactor* of A.

This definition of the determinant, however, does not lead directly to a practical algorithm for its computation, because it requires $O(n!)$ floating-point operations, whereas typical algorithms for matrix computations run in polynomial time. However, the computational effort can be reduced by choosing from the multiple formulas for $\det(A)$ above. By consistently choosing the row or column with the most zeros, the number of operations can be minimized.

However, more practical methods for computing the determinant can be obtained by using its properties:

- If any row or column of A has only zero entries, then $\det(A) = 0$.
- If any two rows or columns of A are the same, then $\det(A) = 0$.
- If $\tilde{A}$ is obtained from A by adding a multiple of a row of A to another row, then $\det(\tilde{A}) = \det(A)$.
- If $\tilde{A}$ is obtained from A by interchanging two rows of A, then $\det(\tilde{A}) = -\det(A)$.
- If $\tilde{A}$ is obtained from A by scaling a row by a scalar λ, then $\det(\tilde{A}) = \lambda \det(A)$.
- If B is an $n \times n$ matrix, then $\det(AB) = \det(A)\det(B)$.
- $\det(A^T) = \det(A)$
- If A is nonsingular, then $\det(A^{-1}) = (\det(A))^{-1}$
- If A is a triangular matrix (either upper or lower), then $\det(A) = \prod_{i=1}^{n} a_{ii}$.

The best-known application of the determinant is the fact that it indicates whether a matrix A is nonsingular, or invertible. The following statements are all equivalent.

- $\det(A) \neq 0$.
- A is nonsingular.
- A^{-1} exists.
- The system $A\mathbf{x} = \mathbf{b}$ has a unique solution for any n-vector $\mathbf{b}$.
- The system $A\mathbf{x} = \mathbf{0}$ has only the *trivial solution* $\mathbf{x} = \mathbf{0}$.

The determinant has other interesting applications. The determinant of a 3 × 3 matrix is equal to the volume of a parallelepiped defined by the vectors that are the rows (or columns) of the matrix.

Example Because the matrices

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 4 & 0 \\ -3 & -5 & -6 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & -1 & 0 & 5 \\ 0 & 2 & 3 & -6 \\ 0 & 0 & -4 & 7 \\ 0 & 0 & 0 & 8 \end{bmatrix}$$

are lower and upper triangular, respectively, their determinants are the products of their diagonal entries. That is,

$$\det(L) = 1(4)(-6) = -24, \quad \det(U) = 1(2)(-4)(8) = -64.$$

□

B.11 Vector and Matrix Norms

B.11.1 Vector Norms

Given vectors $\mathbf{x}$ and $\mathbf{y}$ of length one, which are simply scalars x and y, the most natural notion of distance between x and y is obtained from the absolute value; we define the distance to be $|x - y|$. We therefore define a distance function for vectors that has similar properties.

A function $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ is called a *vector norm* if it has the following properties:

1. $\|\mathbf{x}\| \geq 0$ for any vector $\mathbf{x} \in \mathbb{R}^n$, and $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = \mathbf{0}$
2. $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$ for any vector $\mathbf{x} \in \mathbb{R}^n$ and any scalar $\alpha \in \mathbb{R}$
3. $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ for any vectors $\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^n$.

The last property is called the *triangle inequality*. It should be noted that when $n = 1$, the absolute value function is a vector norm.

The most commonly used vector norms belong to the family of *p-norms*, or *ℓ_p-norms*, which are defined by

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

It can be shown that for any $p > 0$, $\|\cdot\|_p$ defines a vector norm. The following p-norms are of particular interest:

- $p = 1$: The ℓ_1-*norm*
$$\|\mathbf{x}\|_1 = |x_1| + |x_2| + \cdots + |x_n|$$
- $p = 2$: The ℓ_2-*norm* or *Euclidean norm*
$$\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} = \sqrt{\mathbf{x}^T\mathbf{x}}$$
- $p = \infty$: The ℓ_∞-*norm*
$$\|\mathbf{x}\|_\infty = \max_{1\leq i\leq n} |x_i|$$

Example Given the vector
$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix},$$
we have
$$\begin{aligned} \|\mathbf{x}\|_1 &= |1| + |2| + |-3| = 6, \\ \|\mathbf{x}\|_2 &= \sqrt{1^2 + 2^2 + (-3)^2} = \sqrt{14}, \\ \|\mathbf{x}\|_\infty &= \max\{|1|, |2|, |-3|\} = 3. \end{aligned}$$
□

It can be shown that the ℓ_2-norm satisfies the *Cauchy-Bunyakovsky-Schwarz inequality*, also known as simply the *Cauchy-Schwarz inequality*,
$$|\mathbf{x}^T\mathbf{y}| \leq \|\mathbf{x}\|_2\|\mathbf{y}\|_2$$
for any vectors $\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^n$. This inequality is useful for showing that the ℓ_2-norm satisfies the triangle inequality. It is a special case of the *Hölder inequality*
$$|\mathbf{x}^T\mathbf{y}| \leq \|\mathbf{x}\|_p\|\mathbf{y}\|_q, \quad \frac{1}{p} + \frac{1}{q} = 1.$$

We will prove the Cauchy-Schwarz inequality for vectors in $\mathbb{R}^n$; the proof can be generalized to a complex vector space. For $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and $c \in \mathbb{R}$, with $\mathbf{y} \neq \mathbf{0}$, we have
$$(\mathbf{x} - c\mathbf{y})^T(\mathbf{x} - c\mathbf{y}) = \|\mathbf{x} - c\mathbf{y}\|_2^2 \geq 0.$$
It follows from the properties of the inner product that
$$\begin{aligned} 0 &\leq (\mathbf{x} - c\mathbf{y})^T(\mathbf{x} - c\mathbf{y}) \\ &\leq \mathbf{x}^T\mathbf{x} - \mathbf{x}^T(c\mathbf{y}) - (c\mathbf{y})^T\mathbf{x} + (c\mathbf{y})^T(c\mathbf{y}) \\ &\leq \|\mathbf{x}\|_2^2 - 2c\mathbf{x}^T\mathbf{y} + c^2\|\mathbf{y}\|_2^2. \end{aligned}$$

We now try to find the value of c that minimizes this expression. Differentiating with respect to c and equating to zero yields the equation

$$-2\mathbf{x}^T\mathbf{y} + 2c\|\mathbf{y}\|_2^2 = 0,$$

and therefore the minimum occurs when $c = \mathbf{x}^T\mathbf{y}/\|\mathbf{y}\|_2^2$. It follows that

$$\begin{aligned}
0 &\leq \|\mathbf{x}\|_2^2 - 2c\mathbf{x}^T\mathbf{y} + c^2\|\mathbf{y}\|^2 \\
&\leq \|\mathbf{x}\|_2^2 - 2\frac{\mathbf{x}^T\mathbf{y}}{\|\mathbf{y}\|_2^2}\mathbf{x}^T\mathbf{y} + \frac{(\mathbf{x}^T\mathbf{y})^2}{\|\mathbf{y}\|_2^4}\|\mathbf{y}\|_2^2 \\
&\leq \|\mathbf{x}\|_2^2 - 2\frac{(\mathbf{x}^T\mathbf{y})^2}{\|\mathbf{y}\|_2^2} + \frac{(\mathbf{x}^T\mathbf{y})^2}{\|g\|^2} \\
&\leq \|\mathbf{x}\|_2^2 - \frac{(\mathbf{x}^T\mathbf{y})^2}{\|\mathbf{y}\|_2^2}.
\end{aligned}$$

It follows that

$$(\mathbf{x}^T\mathbf{y})^2 \leq \|\mathbf{x}\|_2^2\|\mathbf{y}\|_2^2.$$

Taking the square root of both sides yields the Cauchy-Schwarz inequality.

Now that we have defined various notions of the size, or magnitude, of a vector, we can discuss distance and convergence. Given a vector norm $\|\cdot\|$, and vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, we define the *distance* between $\mathbf{x}$ and $\mathbf{y}$, with respect to this norm, by $\|\mathbf{x} - \mathbf{y}\|$. Then, we say that a sequence of n-vectors $\{\mathbf{x}^{(k)}\}_{k=0}^{\infty}$ *converges* to a vector $\mathbf{x}$ if

$$\lim_{k\to\infty} \|\mathbf{x}^{(k)} - \mathbf{x}\| = 0.$$

That is, the distance between $\mathbf{x}^{(k)}$ and $\mathbf{x}$ must approach zero. It can be shown that regardless of the choice of norm, $\mathbf{x}^{(k)} \to \mathbf{x}$ if and only if

$$\mathbf{x}_i^{(k)} \to x_i, \quad i = 1, 2, \ldots, n.$$

That is, each component of $\mathbf{x}^{(k)}$ must converge to the corresponding component of x. This is due to the fact that for any vector norm $\|\cdot\|$, $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x}$ is the zero vector.

Because we have defined convergence with respect to an arbitrary norm, it is important to know whether a sequence can converge to a limit with respect to one norm, while converging to a different limit in another norm, or perhaps not converging at all. Fortunately, for p-norms, this is never the case. We say that two vector norms $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ are *equivalent* if there exists constants C_1 and C_2, that are independent of $\mathbf{x}$, such that for any vector $\mathbf{x} \in \mathbb{R}^n$,

$$C_1\|\mathbf{x}\|_\alpha \leq \|\mathbf{x}\|_\beta \leq C_2\|\mathbf{x}\|_\alpha.$$

It follows that if two norms are equivalent, then a sequence of vectors that converges to a limit with respect to one norm will converge to the same limit in the other. It can be shown that all ℓ_p-norms are equivalent. In particular, if $\mathbf{x} \in \mathbb{R}^n$, then

$$\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{n}\|\mathbf{x}\|_2,$$

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_\infty,$$

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_1 \leq n\|\mathbf{x}\|_\infty.$$

We will now prove the equivalence of $\|\cdot\|_1$ and $\|\cdot\|_2$. Let $\mathbf{x} \in \mathbb{R}^n$. First, we have

$$\|\mathbf{x}\|_2^2 = \sum_{i=1}^n |x_i|^2 \leq \sum_{i,j=1}^n |x_i||x_j| \leq \|\mathbf{x}\|_1^2,$$

and therefore $\|\mathbf{x}\|_2 \leq \|\mathbf{x}_1\|_1$. Then, we define the vector $\mathbf{y}$ by

$$y_i = \begin{cases} 1 & x_i \geq 0 \\ -1 & x_i < 0 \end{cases}.$$

It follows that $\|\mathbf{x}_1\| = \mathbf{y}^T\mathbf{x}$. By the Cauchy-Schwarz inequality,

$$\|\mathbf{x}\|_1 = \mathbf{y}^T\mathbf{x} \leq \|\mathbf{y}\|_2\|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_2$$

and the equivalence of the norms has been established.

B.11.2 Matrix Norms

It is also very useful to be able to measure the magnitude of a matrix, or the distance between matrices. However, it is not sufficient to simply define the norm of an $m \times n$ matrix A as the norm of an mn-vector $\mathbf{x}$ whose components are the entries of A. We instead define a *matrix norm* to be a function $\|\cdot\| : \mathbb{R}^{m\times n} \to \mathbb{R}$ that has the following properties:

- $\|A\| \geq 0$ for any $A \in \mathbb{R}^{m\times n}$, and $\|A\| = 0$ if and only if $A = 0$
- $\|\alpha A\| = |\alpha|\|A\|$ for any $m \times n$ matrix A and scalar α
- $\|A + B\| \leq \|A\| + \|B\|$ for any $m \times n$ matrices A and B

Another property that is often, but not always, included in the definition of a matrix norm is the *submultiplicative property*: if A is $m \times n$ and B is $n \times p$, we require that

$$\|AB\| \leq \|A\|\|B\|.$$

This is particularly useful when A and B are square matrices.

Any vector norm *induces* a matrix norm. It can be shown that given a vector norm, defined appropriately for m-vectors and n-vectors, the function $\|\cdot\| : \mathbb{R}^{m\times n} \to \mathbb{R}$ defined by

$$\|A\| = \sup_{\mathbf{x}\neq\mathbf{0}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} = \max_{\|\mathbf{x}\|=1} \|A\mathbf{x}\|$$

is a matrix norm. It is called the *natural*, or *induced*, matrix norm. Furthermore, if the vector norm is a ℓ_p-norm, then the induced matrix norm satisfies the submultiplicative property.

The following matrix norms are of particular interest:

- The ℓ_1-*norm*:

  $$\|A\|_1 = \max_{\|\mathbf{x}\|_1=1} \|A\mathbf{x}\|_1 = \max_{1\le j\le n} \sum_{i=1}^m |a_{ij}|.$$

  That is, the ℓ_1-norm of a matrix is its maximum column sum of $|A|$. To see this, let $\mathbf{x} \in \mathbb{R}^n$ satisfy $\|\mathbf{x}\|_1 = 1$. Then

  $$\begin{aligned} \|A\mathbf{x}\|_1 &= \sum_{i=1}^m |(A\mathbf{x})_i| \\ &= \sum_{i=1}^m \left| \sum_{j=1}^n a_{ij}x_j \right| \\ &\le \sum_{j=1}^n |x_j| \left(\sum_{i=1}^m |a_{ij}| \right) \\ &\le \sum_{j=1}^n |x_j| \max_{1\le j\le n} \left(\sum_{i=1}^m |a_{ij}| \right) \\ &\le \max_{1\le j\le n} \left(\sum_{i=1}^m |a_{ij}| \right). \end{aligned}$$

  Equality is achieved if $\mathbf{x} = \mathbf{e}_J$, where the index J satisfies

  $$\max_{1\le j\le n} \left(\sum_{i=1}^m |a_{ij}| \right) = \sum_{i=1}^m |a_{iJ}|.$$

  It follows that the maximum column sum of $|A|$ is equal to the maximum of $\|A\mathbf{x}\|_1$ taken over all the set of all unit 1-norm vectors.

- The ℓ_∞-*norm*:

  $$\|A\|_\infty = \max_{\|\mathbf{x}\|_\infty=1} \|A\mathbf{x}\|_\infty = \max_{1\le i\le m} \sum_{j=1}^n |a_{ij}|.$$

  That is, the ℓ_∞-norm of a matrix is its maximum *row* sum. This formula can be obtained in a similar manner as the one for the matrix 1-norm.

- The ℓ_2-*norm*:

  $$\|A\|_2 = \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2.$$

  To obtain a formula for this norm, we note that the function

  $$g(\mathbf{x}) = \frac{\|A\mathbf{x}\|_2^2}{\|\mathbf{x}\|_2^2}$$

  has a local maximum or minimum whenever $\mathbf{x}$ is a *unit* ℓ_2-norm vector (that is, $\|\mathbf{x}\|_2 = 1$) that satisfies

  $$A^T A\mathbf{x} = \|A\mathbf{x}\|_2^2\mathbf{x},$$

as can be shown by differentiation of $g(\mathbf{x})$. That is, $\mathbf{x}$ is an *eigenvector* of A^TA, with corresponding *eigenvalue* $\|A\mathbf{x}\|_2^2 = g(\mathbf{x})$. We conclude that

$$\|A\|_2 = \max_{1\leq i\leq n} \sqrt{\lambda_i(A^TA)}.$$

That is, the ℓ_2-norm of a matrix is the square root of the largest eigenvalue of A^TA, which is guaranteed to be nonnegative, as can be shown using the vector 2-norm. We see that unlike the vector ℓ_2-norm, the matrix ℓ_2-norm is much more difficult to compute than the matrix ℓ_1-norm or ℓ_∞-norm.

- The *Frobenius norm*:

  $$\|A\|_F = \left(\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2\right)^{1/2}.$$

  It should be noted that the Frobenius norm is *not* induced by any vector ℓ_p-norm, but it is equivalent to the vector ℓ_2-norm in the sense that $\|A\|_F = \|\mathbf{x}\|_2$ where $\mathbf{x}$ is obtained by reshaping A into a vector.

Like vector norms, matrix norms are equivalent. For example, if A is an $m \times n$ matrix, we have

$$\|A\|_2 \leq \|A\|_F \leq \sqrt{n}\|A\|_2,$$

$$\frac{1}{\sqrt{n}}\|A\|_\infty \leq \|A\|_2 \leq \sqrt{m}\|A\|_\infty,$$

$$\frac{1}{\sqrt{m}}\|A\|_1 \leq \|A\|_2 \leq \sqrt{n}\|A\|_1.$$

Example Let

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

Then

$$\|A\|_1 = \max\{|1| + |0| + |-1|, |2| + |1| + |0|, |3| + |0| + |4|\} = 7,$$

and

$$\|A\|_\infty = \max\{|1| + |2| + |3|, |0| + |1| + |0|, |-1| + |0| + |4|\} = 6.$$

□

B.12 Function Spaces and Norms

We now define norms on more general vector spaces. Let $\mathcal{V}$ be a vector space over the field of real numbers $\mathbb{R}$. A *norm* on $\mathcal{V}$ is a function $\|\cdot\| : \mathcal{V} \to \mathbb{R}$ that has the following properties:

1. $\|f\| \geq 0$ for all $f \in \mathcal{V}$, and $\|f\| = 0$ if and only if f is the zero vector of $\mathcal{V}$.
2. $\|cf\| = |c|\|f\|$ for any vector $f \in \mathcal{V}$ and any scalar $c \in \mathbb{R}$.

3. $\|f+g\| \leq \|f\| + \|g\|$ for all $f, g \in \mathcal{V}$.

The last property is known as the *triangle inequality.* A vector space $\mathcal{V}$, together with a norm $\|\cdot\|$, is called a *normed vector space* or *normed linear space.* In particular, we are interested in working with *function spaces*, which are vector spaces in which the vectors are functions.

Example The space $C[a,b]$ of functions that are continuous on the interval $[a,b]$ is a normed vector space with the norm

$$\|f\|_\infty = \max_{a\leq x\leq b} |f(x)|,$$

known as the ∞-*norm* or *maximum norm.* □

Example The space $C[a,b]$ can be equipped with a different norm, such as

$$\|f\|_2 = \left(\int_a^b |f(x)|^2 w(x)\,dx\right)^{1/2},$$

where the *weight function* $w(x)$ is positive and integrable on (a,b). It is allowed to be singular at the endpoints, as will be seen in certain examples. This norm is called the *2-norm* or *weighted 2-norm.* □

The 2-norm and ∞-norm are related as follows:

$$\|f\|_2 \leq W\|f\|_\infty, \quad W = \|1\|_2.$$

However, unlike the ∞-norm and 2-norm defined for the vector space $\mathbb{R}^n$, these norms are not *equivalent* in the sense that a function that has a small 2-norm necessarily has a small ∞-norm. In fact, given any $\epsilon > 0$, no matter how small, and any $M > 0$, no matter how large, there exists a function $f \in C[a,b]$ such that

$$\|f\|_2 < \epsilon, \quad \|f\|_\infty > M.$$

We say that a function f is *absolutely continouous* on $[a,b]$ if its derivative is finite almost everywhere in $[a,b]$ (meaning that it is not finite on at most a subset of $[a,b]$ that has *measure zero*), is integrable on $[a,b]$, and satisfies

$$\int_a^x f'(s)\,dx = f(x) - f(a), \quad a \leq x \leq b.$$

Any continuously differentiable function is absolutely continuous, but the converse is not necessarily true.

Example B.12.1 *For example, $f(x) = |x|$ is absolutely continuous on any interval of the form $[-a,a]$, but it is not continuously differentiable on such an interval.* □

Next, we define the *Sobolev spaces* $H^k(a,b)$ as follows. The space $H^1(a,b)$ is the set of all absolutely continuous functions on $[a,b]$ whose derivatives belong to $L^2(a,b)$. Then, for $k > 1$, $H^k(a,b)$ is the subset of $H^{k-1}(a,b)$ consisting of functions whose $(k-1)$st derivatives are absolutely continuous, and whose kth derivatives belong to $L^2(a,b)$. If we denote by $C^k[a,b]$ the set of all functions defined on $[a,b]$ that are k times continuously differentiable, then $C^k[a,b]$ is a proper subset of $H^k(a,b)$. For example, any piecewise linear belongs to $H^1(a,b)$, but does not generally belong to $C^1[a,b]$.

Example B.12.2 *The function* $f(x) = x^{3/4}$ *belongs to* $H^1(0,1)$ *because* $f'(x) = \frac{3}{4}x^{-1/4}$ *is integrable on* $[0,1]$*, and also* square-integrable *on* $[0,1]$*, since*

$$\int_0^1 |f'(x)|^2\,dx = \int_0^1 \frac{9}{16}x^{-1/2} = \frac{9}{8}x^{1/2}\Big|_0^1 = \frac{9}{8}.$$

However, $f \notin C^1[a,b]$*, because* $f'(x)$ *is singular at* $x = 0$. □

B.13 Inner Product Spaces

Recall that two m-vectors $\mathbf{u} = \langle u_1, u_2, \ldots, u_m\rangle$ and $\mathbf{v} = \langle v_1, v_2, \ldots, v_m\rangle$ are *orthogonal* if

$$\mathbf{u}\cdot\mathbf{v} = \sum_{i=1}^m u_i v_i = 0,$$

where $\mathbf{u}\cdot\mathbf{v}$ is the *dot product*, or *inner product*, of $\mathbf{u}$ and $\mathbf{v}$.

By viewing functions defined on an interval $[a,b]$ as infinitely long vectors, we can generalize the inner product, and the concept of orthogonality, to functions. To that end, we define the *inner product* of two real-valued functions $f(x)$ and $g(x)$ defined on the interval $[a,b]$ by

$$\langle f, g\rangle = \int_a^b f(x)g(x)\,dx.$$

Then, we say f and g are *orthogonal* with respect to this inner product if $\langle f, g\rangle = 0$.

In general, an inner product on a vector space $\mathcal{V}$ over $\mathbb{R}$, be it continuous or discrete, has the following properties:

1. $\langle f+g, h\rangle = \langle f, h\rangle + \langle g, h\rangle$ for all $f, g, h \in \mathcal{V}$
2. $\langle cf, g\rangle = c\langle f, g\rangle$ for all $c \in \mathbb{R}$ and all $f \in \mathcal{V}$
3. $\langle f, g\rangle = \langle g, f\rangle$ for all $f, g \in \mathcal{V}$
4. $\langle f, f\rangle \geq 0$ for all $f \in \mathcal{V}$, and $\langle f, f\rangle = 0$ if and only if $f = \mathbf{0}$.

This inner product can be used to define the *norm* of a function, which generalizes the concept of the magnitude of a vector to functions, and therefore provides a measure of the "magnitude" of a function. Recall that the magnitude of a vector $\mathbf{v}$, denoted by $\|\mathbf{v}\|$, can be defined by

$$\|\mathbf{v}\| = (\mathbf{v}\cdot\mathbf{v})^{1/2}.$$

Along similar lines, we define the *2-norm* of a function $f(x)$ defined on $[a,b]$ by

$$\|f\|_2 = (\langle f, f\rangle)^{1/2} = \left(\int_a^b [f(x)]^2\,dx\right)^{1/2}.$$

It can be verified that this function does in fact satisfy the properties required of a norm.

One very important property that $\|\cdot\|_2$ has is that it satisfies the *Cauchy-Schwarz inequality*

$$|\langle f, g\rangle| \leq \|f\|_2\|g\|_2, \quad f, g \in \mathcal{V}.$$

This can be proven by noting that for any scalar $c \in \mathbb{R}$,

$$c^2\|f\|_2^2 + 2c\langle f, g\rangle + \|g\|_2^2 = \|cf + g\|_2^2 \geq 0.$$

The left side is a quadratic polynomial in c. In order for this polynomial to not have any negative values, it must either have complex roots or a double real root. This is the case if the discrimant satisfies

$$4\langle f, g\rangle^2 - 4\|f\|_2^2\|g\|_2^2 \leq 0,$$

from which the Cauchy-Schwarz inequality immediately follows. By setting $c = 1$ and applying this inequality, we immediately obtain the triangle-inequality property of norms.

B.14 Eigenvalues

We have learned what it means for a sequence of vectors to converge to a limit. However, using the definition alone, it may still be difficult to determine, conclusively, whether a given sequence of vectors converges. For example, suppose a sequence of vectors is defined as follows: we choose the initial vector $\mathbf{x}^{(0)}$ arbitrarily, and then define the rest of the sequence by

$$\mathbf{x}^{(k+1)} = A\mathbf{x}^{(k)}, \quad k = 0, 1, 2, \ldots$$

for some matrix A. Such a sequence actually arises in the study of the convergence of various iterative methods for solving systems of linear equations.

An important question is whether a sequence of this form converges to the zero vector. This will be the case if

$$\lim_{k\to\infty} \|\mathbf{x}^{(k)}\| = 0$$

in some vector norm. From the definition of $\mathbf{x}^{(k)}$, we must have

$$\lim_{k\to\infty} \|A^k\mathbf{x}^{(0)}\| = 0.$$

From the submultiplicative property of matrix norms,

$$\|A^k\mathbf{x}^{(0)}\| \leq \|A\|^k\|\mathbf{x}^{(0)}\|,$$

from which it follows that the sequence will converge to the zero vector if $\|A\| < 1$. However, this is only a *sufficient* condition; it is not *necessary.*

To obtain a sufficient *and* necessary condition, it is necessary to achieve a better understanding of the effect of matrix-vector multiplication on the magnitude of a vector. However, because matrix-vector multiplication is a complicated operation, this understanding can be difficult to acquire. Therefore, it is helpful to identify circumstances under which this operation can be simply described.

To that end, we say that a nonzero vector $\mathbf{x}$ is an *eigenvector* of an $n \times n$ matrix A if there exists a scalar λ such that

$$A\mathbf{x} = \lambda\mathbf{x}.$$

The scalar λ is called an *eigenvalue* of A corresponding to $\mathbf{x}$. Note that although $\mathbf{x}$ is required to be nonzero, it is possible that λ can be zero. It can also be complex, even if A is a real matrix.

If we rearrange the above equation, we have

$$(A - \lambda I)\mathbf{x} = \mathbf{0}.$$

That is, if λ is an eigenvalue of A, then $A - \lambda I$ is a singular matrix, and therefore $\det(A - \lambda I) = 0$. This equation is actually a polynomial in λ, which is called the *characteristic polynomial* of A. If A is an $n \times n$ matrix, then the characteristic polynomial is of degree n, which means that A has n eigenvalues, which may repeat.

The following properties of eigenvalues and eigenvectors are helpful to know:

- If λ is an eigenvalue of A, then there is at least one eigenvector of A corresponding to λ
- If there exists an invertible matrix P such that $B = PAP^{-1}$, then A and B have the same eigenvalues. We say that A and B are *similar*, and the transformation PAP^{-1} is called a *similarity transformation*.
- If A is a symmetric matrix, then its eigenvalues are real.
- If A is a *skew-symmetric* matrix, meaning that $A^T = -A$, then its eigenvalues are either equal to zero, or are purely imaginary.
- If A is a real matrix, and $\lambda = u + iv$ is a complex eigenvalue of A, then $\bar{\lambda} = u - iv$ is also an eigenvalue of A.
- If A is a triangular matrix, then its diagonal entries are the eigenvalues of A.
- $\det(A)$ is equal to the product of the eigenvalues of A.
- $\mathrm{tr}(A)$, the sum of the diagonal entries of A, is also equal to the sum of the eigenvalues of A.

It follows that any method for computing the roots of a polynomial can be used to obtain the eigenvalues of a matrix A. However, in practice, eigenvalues are normally computed using iterative methods that employ orthogonal similarity transformations to reduce A to upper triangular form, thus revealing the eigenvalues of A. In practice, such methods for computing eigenvalues are used to compute roots of polynomials, rather than using polynomial root-finding methods to compute eigenvalues, because they are much more robust with respect to roundoff error.

It can be shown that if each eigenvalue λ of a matrix A satisfies $|\lambda| < 1$, then, for any vector $\mathbf{x}$,

$$\lim_{k\to\infty} A^k\mathbf{x} = \mathbf{0}.$$

Furthermore, the converse of this statement is also true: if there exists a vector $\mathbf{x}$ such that $A^k\mathbf{x}$ does not approach $\mathbf{0}$ as $k \to \infty$, then at least one eigenvalue λ of A must satisfy $|\lambda| \geq 1$.

Therefore, it is through the eigenvalues of A that we can describe a necessary and sufficient condition for a sequence of vectors of the form $\mathbf{x}^{(k)} = A^k\mathbf{x}^{(0)}$ to converge to the zero vector. Specifically, we need only check if the magnitude of the largest eigenvalue is less than 1. For convenience, we define the *spectral radius* of A, denoted by $\rho(A)$, to be $\max|\lambda|$, where λ is an eigenvalue of A. We can then conclude that the sequence $\mathbf{x}^{(k)} = A^k\mathbf{x}^{(0)}$ converges to the zero vector if and only if $\rho(A) < 1$.

Example Let

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

Then

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

Because $A - 2I$ has a column of all zeros, it is singular. Therefore, 2 is an eigenvalue of A. In fact, $A\mathbf{e}_1 = 2\mathbf{e}_1$, so $\mathbf{e}_1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$ is an eigenvector.

Because A is upper triangular, its eigenvalues are the diagonal elements, 2, 4 and 1. Because the largest eigenvalue in magnitude is 4, the spectral radius of A is $\rho(A) = 4$. □

The spectral radius is closely related to natural (induced) matrix norms. Let λ be the largest eigenvalue of A, with $\mathbf{x}$ being a corresponding eigenvector. Then, for any natural matrix norm $\|\cdot\|$, we have

$$\rho(A)\|\mathbf{x}\| = |\lambda|\|\mathbf{x}\| = \|\lambda\mathbf{x}\| = \|A\mathbf{x}\| \leq \|A\|\|\mathbf{x}\|.$$

Therefore, we have $\rho(A) \leq \|A\|$. When A is symmetric, we also have

$$\|A\|_2 = \rho(A).$$

For a general matrix A, we have

$$\|A\|_2 = [\rho(A^TA)]^{1/2},$$

which can be seen to reduce to $\rho(A)$ when $A^T = A$, since, in general, $\rho(A^k) = \rho(A)^k$.

Because the condition $\rho(A) < 1$ is necessary and sufficient to ensure that $\lim_{k\to\infty} A^k\mathbf{x} = \mathbf{0}$, it is possible that such convergence may occur even if $\|A\| \geq 1$ for some natural norm $\|\cdot\|$. However, if $\rho(A) < 1$, we can conclude that

$$\lim_{k\to\infty} \|A^k\| = 0,$$

even though $\lim_{k\to\infty} \|A\|^k$ may not even exist.

In view of the definition of a matrix norm, that $\|A\| = 0$ if and only if $A = 0$, we can conclude that if $\rho(A) < 1$, then A^k converges to the zero matrix as $k \to \infty$. In summary, the following statements are all equivalent:

1. $\rho(A) < 1$
2. $\lim_{k\to\infty} \|A^k\| = 0$, for any natural norm $\|\cdot\|$
3. $\lim_{k\to\infty} (A^k)_{ij} = 0$, $i, j = 1, 2, \ldots, n$
4. $\lim_{k\to\infty} A^k\mathbf{x} = \mathbf{0}$

These results are very useful for analyzing the convergence behavior of various iterative methods for solving systems of linear equations.

B.15 Differentiation of Matrices

Suppose that $A(t)$ is an $m \times n$ matrix in which each entry is a function of a parameter t. Then the matrix $A'(t)$, or dA/dt, is the $m \times n$ matrix obtained by differentiating each entry with respect to t. That is,

$$\left[\frac{dA(t)}{dt}\right]_{ij} = \frac{d}{dt}[a_{ij}(t)].$$

Matrices obey differentiation rules that are analogous to differentiation rules for functions, but the rules of matrix-matrix multiplication must be taken into account. For example, if $A(t)$ is an $m \times n$ matrix and B is an $n \times p$ matrix, then

$$\frac{d}{dt}[A(t)B(t)] = \frac{d}{dt}[A(t)]B(t) + A(t)\frac{d}{dt}[B(t)],$$

and if $A(t)$ is a nonsingular $n \times n$ matrix, then

$$\frac{d}{dt}[A(t)^{-1}] = -A(t)^{-1}\frac{d}{dt}[A(t)]A(t)^{-1}.$$

It is also useful to know how to differentiate functions of vectors with respect to their components. Let A be an $n \times n$ matrix and $\mathbf{x} \in \mathbb{R}^n$. Then, we have

$$\nabla\left(\mathbf{x}^T A\mathbf{x}\right) = (A + A^T)\mathbf{x}, \quad \nabla(\mathbf{x}^T\mathbf{b}) = \mathbf{b}.$$

These formulas are useful in problems involving minimization of functions of $\mathbf{x}$, such as the *least-squares problem*, which entails approximately solving a system of m equations in n unknowns, where typically $m > n$.

Bibliography

[1] Abramowitz, M. and Stegun, I. A., Eds.: *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, 9th printing. New York: Dover (1972).

[2] Aitken, A. C.: “On interpolation by iteration of proportional parts, without the use of differences”. *Proc. Edinburgh Math. Soc.* **3**(2) (1932), p. 56-76.

[3] Banach, S.: “Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales.” *Fund. Math.* **3** (1922), p. 133-181.

[4] Bashforth, F. and Adams, J. C.: *An Attempt to test the Theories of Capillary Action by comparing the theoretical and measured forms of drops of fluid. With an explanation of the method of integration employed in constructing the tables which give the theoretical forms of such drops*, Cambridge University Press (1883).

[5] Berrut, J.-P. and Trefethen, L. N.: “Barycentric Lagrange Interpolation”. *SIAM Review* **46**(3) (2004), p. 501-517.

[6] Birkhoff, G. and De Boor, C.: “Error bounds for spline interpolation”, *Journal of Mathematics and Mechanics* **13** (1964), p. 827-836.

[7] Birkhoff, G. and Rota, G.: *Ordinary differential equations*, (Fourth edition), John Wiley & Sons, New York, 1989.

[8] Bogacki, P. and Shampine, L. F.: “A 3(2) pair of Runge?Kutta formulas”, *Applied Mathematics Letters* **2**(4) (1989), p. 321-325.

[9] Brouwer, L. E. J.: “Über Abbildungen von Mannigfaltigkeiten”, *Mathematische Annalen* **71** (1911), p. 97-115.

[10] Burden, R. L. and Faires, J. D.: *Numerical Analysis*, 9th Edition. Brooks Cole (2004).

[11] Dahlquist, G.: “A special stability problem for linear multistep methods”, *BIT* **3** (1963), p. 27-43.

[12] Demmel, J. W.: *Applied Numerical Linear Algebra*, SIAM (1997).

[13] Dormand, J. R. and Prince, P. J.: “A family of embedded Runge-Kutta formulae”, *Journal of Computational and Applied Mathematics* **6** (1) (1980), p. 19-26.

[14] Fehlberg, E.: "Klassische Runge-Kutta Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf Wrmeleitungsprobleme", *Computing* **6** (1970), p. 61-71.

[15] Gautschi, W. *Numerical Analysis: an Introduction*, Birkhäuser, Boston, MA, 1997.

[16] Golub, G. H. and van Loan, C. F.: *Matrix Computations*, 4th Edition, Johns Hopkins University Press (2012).

[17] Gustafsson, B., Kreiss, H.-O. and Oliger, J. E.: *Time-Dependent Problems and Difference Methods*, John WIley & Sons, New York (1995).

[18] Heath, M. T.: *Scientific Computing: An Introductory Survey*, 2nd Edition, McGraw-Hill (2002).

[19] Issacson, E. and Keller, H. B.: *Analysis of numerical methods*, John Wiley & Sons, New York (1966).

[20] Keller, H. B.: *Numerical Methods for Two-Point Boundary Value Problems*, Blaisdell, Waltham, MA (1968).

[21] Lambers, J. V. and Rice, J. R., "Numerical Quadrature for General Two-Dimensional Domains", *Computer Science Technical Reports*, Paper 906 (1991).

[22] Laurie, D.: "Calculation of Gauss-Kronrod quadrature rules", *Mathematics of Computation of the American Mathematical Society* **66**(219) (1997), p. 1133-1145.

[23] Le Gendre, M.: "Recherches sur l'attraction des sphéroïdes homogènes", *Mémoires de Mathématiques et de Physique, présentés à l'Académie Royale des Sciences, par divers savans, et lus dans ses Assemblées*, Tome X (1785), p. 411-435.

[24] Moler, C. B.:"Demonstration of a matrix laboratory". *Lecture notes in mathematics* (J. P. Hennart, ed.), Springer-Verlag, Berlin (1982), p. 84-98.

[25] Moulton, F. R.: *New methods in exterior ballistics*, University of Chicago Press (1926).

[26] Neville, E. H.: "Iterative Interpolation", *J. Indian Math Soc.* **20** (1934), p. 87-120.

[27] Padé, H.: "Sur la répresentation approchée d'une fonction par des fractions rationelles", Thesis, *Ann. École Nor.* (3), **9** (1892), p. 1-93.

[28] Quarteroni, A. and Saleri, F.: *Scientific Computing with MATLAB*, Texts in computational science and engineering **2**, Springer (2003), p. 66.

[29] Ralston, A. and Rabinowitz, P.: *A first course in numerical analysis*, (2nd edition), McGraw-Hill, New York, 1978.

[30] Rice, J. R.: "A Metalgorithm for Adaptive Quadrature", *Journal of the ACM* **22**(1) (1975), p. 61-82.

[31] Richardson, L. F.: "The approximate arithmetical solution by finite differences of physical problems including differential equations, with an application to the stresses in a masonry dam", *Philosophical Transactions of the Royal Society A.* **210**(459-470) (1911), p. 307-357.

[32] Romberg, W.: "Vereinfachte numerische Integration", *Det Kongelige Norske Videnskabers Selskab Forhandlinger*, Trondheim **28**(7) (1955), p. 30-36.

[33] Runge, C.: "Über empirische Funktionen und die Interpolation zwischen quidistanten Ordinaten", *Zeitschrift für Mathematik und Physik* **46** (1901), p. 224-243.

[34] Schultz, M. H.: *Spline analysis*, Prentice-Hall, Englewood Cliffs, NJ (1973).

[35] Shampine, L. F. and Reichelt, M. W.: "The MATLAB ODE Suite", *SIAM J. Sci. Comput.* **18**(1) (1997), p. 1-22.

[36] Süli, E. and Mayers, D.: *An Introduction to Numerical Analysis*, Cambridge University Press (2003).

[37] Wilbraham, H.: "On a certain periodic function", *The Cambridge and Dublin Mathematical Journal* **3** (1848), p. 198-201.

Index

p-norm, 400
2-norm, 406, 407

A-stability, 332
absolute continuity, 406
absolute stability, region of, 332
Adams methods, 319
Adams-Bashforth method, 319
Adams-Moulton method, 320
Adams-Moulton predictor-corrector method, 321
adaptive quadrature, 257
adaptive time-stepping, 335
aitken's Δ^2 method, 298
algorithm, stable, 29
aliasing, 221
antiderivative, 232
arithmetic, floating-point, 38
associativity, 386

backward difference operator, 176
backward differentiation formula, 322
backward error, 28
backward Euler's method, 318
base, 35
basis, 389
Bernoulli numbers, 252
binomial coefficient, extended, 174
bisection, 274, 374
Bogacki-Shampine method, 337
boundary condition, 343
boundary value problem, two-point, 343
Broyden's Method, 307

Cartesian product rule, 262
catastrophic cancellation, 39
Cauchy-Schwarz inequality, 401, 407
ceiling function, 372
characteristic, 35
characteristic equation, 328
characteristic polynomial, 409
Chebyshev polynomials, 178, 209, 247
chopping, 37
circulant matrix, 231
closure, 387
cofactor, 399
collocation, 354
collocation points, 354
colon operator, 13
column space, 391
commutativity, 386
complete orthogonal decomposition, 113
condition number, absolute, 28
condition number, relative, 28
consistency, 323
consistency, one-step method, 324
continued fraction, 214
continuity, 371, 372
contraction, 282
convergence, 323
convergence, cubic, 272
convergence, linear, 271
convergence, local, 284
convergence, quadratic, 272
convergence, sequence of vectors, 402
convergence, superlinear, 271
cubature, 263
cubature rule, 263

Dahlquist's Barrier Theorem, first, 334
Dahlquist's Barrier Theorem, second, 334
Dahlquist's Equivalence Theorem, 333
data fitting, 159
degenerate function, 272
degree of accuracy, 233, 239
determinant, 399

diagonal, main, 398
difference equation, 312
difference operator, forward, 174, 298
difference, backward, 226
difference, centered, 226
difference, forward, 225
differentiation matrix, 230
dimension, 389
direct construction, 241
discrete fourier transform, 220
distance, vectors, 402
divided difference, 166
divided-difference table, 167
Dormand-Prince method, 337
dot product, 394

eigenvalue, 405, 408
eigenvector, 405, 408
embedded pair, 337
equilibration, 69
equivalence of norms, 402
error analysis, backward, 28
error, absolute, 26
error, computational, 25
error, data, 25
error, relative, 26
error, roundoff, 37
error, truncation, 25
Euler's method, 312
Euler's method, modified, 316
Euler-Maclaurin Expansion, 252
explicit method, 318
exponent, 35

field, 386
finite difference, 226, 346
finite element method, 361
fixed point, 279, 301
fixed point, stable, 284
fixed point, unstable, 284
fixed-point iteration, 279, 300
forward error, 27
Fourier series, 217
FSAL method, 337
function, component, 301, 371
function, coordinate, 301, 371

Galerkin method, 359
Gauss-Lobatto quadrature, 249
Gauss-Radau quadrature, 248
Gibbs' phenomenon, 223
Givens rotation, 104
global error, 329, 335
Gram-Schmidt orthogonalization, 95, 204
    modified, 97
group, 398
group, general linear, 398

Hölder inequality, 401
Hadamard product, 395
Hadamard's conditions, 311
Hermite polynomial, 180
Hilbert matrix, 202
horner's method, 170
Householder reflection, 98

identity matrix, 397
IEEE floating-point standard, 38
ill-conditioned matrix, 202
ill-posed problem, 311
image, 390
implicit method, 318
initial value problem, 343
inner product, 394, 407
inner product space, 201
integral, definite, 378
integral, inner, 263
integral, iterated, 262
integral, outer, 262
integrand, 378
integration, 378
intermediate value theorem, 272
interpolant, Fourier, 219
interpolating polynomial, 159
interpolation, 159
interpolation point, 159
interpolation, barycentric, 163
interpolation, Newton, 166
inverse function theorem, 272
inverse of matrix, 397

Jacobian matrix, 302

Kronecker delta, 181

Lagrange interpolation, 161
Lagrange multipliers, 122
Lagrange polynomial, 161
least squares problem, 194
    linear constraints, 121
    quadratic constraints, 123
    total, 125
least-squares problem, 411
least-squares problem, continuous, 201
Legendre polynomials, 206
limit, 369
limit of integration, 378
linear combination, 387
linear dependence, 388
linear independence, 201, 388
linear operator, 389
linear transformation, 389
Lipschitz condition, 282, 311
Lipschitz constant, 282, 311
local extrapolation, 337
local truncation error, 324

M-file, 14
machine number, 36
Maclaurin polynomial, 380
mantissa, 35
matrix, 385
matrix product, 392
matrix, diagonal, 398
matrix, invertible, 397
matrix, lower triangular, 398
matrix, nonsingular, 397
matrix, singular, 397
matrix, upper triangular, 398
matrix-vector product, 390
maximum norm, 406
midpoint method, explicit, 316
Midpoint Rule, 234
Midpoint Rule, Composite, 238
minor, 399
moment matching, 241
monomial basis, 160
Monte Carlo method, 267
multistep method, 319

nested form, 171
nested multiplication, 170
Neville's Method, 165
Newton backward-difference formula, 176
newton form, 166
Newton forward-difference formula, 174
Newton's method, 18, 289, 304
Newton-Cotes quadrature, 234
node, quadrature, 232
norm, 217, 405, 407
norm, ℓ_1, matrix, 404
norm, ℓ_1, vector, 401
norm, ℓ_2, matrix, 404
norm, ℓ_2, vector, 401
norm, ℓ_∞, matrix, 404
norm, ℓ_∞, vector, 401
norm, equivalent, 406
norm, Euclidean, 401
norm, Frobenius, 405
norm, induced, 403
norm, matrix, 403
norm, natural, 403
norm, vector, 400
normal equations, 93, 195, 197, 201
normalization, 36
normed space, 201
normed vector space, 406
null space, 391
nullity, 391
number system, floating-point, 35
numerical analysis, 3

one-step method, 312
optimization problem, 376
order of accuracy, 225, 239, 330
orthogonal set, 203
orthogonal vectors, 394
orthogonality, 407
orthonormal set, 204
osculatory interpolation, 180
outer product, 306, 394
outer product update, 397
overdetermined system, 91
overflow, 36

Padé approximant, 212

Parseval's identity, 218
partition, column, 396
partition, row, 396
periodic boundary conditions, 217
piecewise continuous, 223
polynomial, piecewise, 183
polynomial, trigonometric, 209
power form, 170
precision, 34, 35
precision, double, 38
precision, machine, 37
precision, single, 38
projection, 115
    symmetric, 97
Property A, 68
pseudo-inverse, 113, 115, 120

QR factorization, 95
quadrature rule, 232
quadrature rule, closed, 232
quadrature rule, composite, 238
quadrature rule, Gaussian, 241
quadrature rule, interpolatory, 233
quadrature rule, open, 232
quadrature, adaptive, 256
quadrature, automatic, 256
quadrature, Gauss-Kronrod, 246

range, 391
rank, 391
rank, column, 391
rank, full, 391
rank-deficient, 391
rank-one update, 306
recursion coefficients, 205
relaxation, 287
Richardson extrapolation, 250
Riemann sum, 377
Romberg integration, 253
root condition, 328
root, double, 272
root, simple, 328
rounding, 37
rounding to nearest, 37
rounding toward zero, 37
row rank, 391
row space, 391
Runge's Example, 238
Runge's example, 177
Runge-Kutta method, 316
Runge-Kutta method, fourth-order, 316
Runge-Kutta-Fehlberg method, 337

scalar product, 386
secant line, 295
secant method, 295
Sherman-Morrison Formula, 307
shooting method, 343
significant digits, 26
similar matrices, 409
similarity transformation, 409
Simpson's method, 328
Simpson's Rule, 235
Simpson's Rule, Composite, 238
skew-symmetric matrix, 394, 409
Sobolev space, 406
span, 387
spectral radius, 409
spline, 186
spline, cubic, 186
spline, Hermite cubic, 191
spline, linear, 184
square-integrable, 218
square-integrable function, 407
stability, 323, 327
stability polynomial, 331
stability, strong, 328
stability, weak, 328
stage, Runge-Kutta method, 317
standard basis, 387, 390
stationary point, 301
steffenson's method, 298
stiff differential equation, 330
stiffness matrix, 361
submultiplicative property, 403
subspace, 387
symmetric matrix, 393

Taylor polynomial, 380
Taylor remainder, 380
test equation, 330
test function, 359

three-term recurrence relation, 205
time step, 312
time-marching, 312
time-stepping, 312
transpose, 393
transpose, Hermitian, 14
trapezoidal method, 318
trapezoidal method, explicit, 316
Trapezoidal Rule, 234
Trapezoidal Rule, Composite, 238
trial function, 359
triangle inequality, 400, 406
trivial solution, 400

underflow, 36
underflow, gradual, 36
unit roundoff, 37

Vandermonde matrix, 160, 198
vector space, 386

weak form, 359
weight function, 208, 247, 406
weight, quadrature, 232
well-posed problem, 311
well-posedness, 25

zero-stability, 325