

# Contents

---

|   |             |
|---|-------------|
| <b>Preface to the Third Edition (2007)</b>                            | <b>xi</b>   |
| <b>Preface to the Second Edition (1992)</b>                           | <b>xiv</b>  |
| <b>Preface to the First Edition (1985)</b>                            | <b>xvii</b> |
| <b>License and Legal Information</b>                                  | <b>xix</b>  |
| <b>1 Preliminaries</b>  | <b>1</b>    |
| 1.0 Introduction . . . . .  | 1           |
| 1.1 Error, Accuracy, and Stability . . . . .                          | 8           |
| 1.2 C Family Syntax . . . . .   | 12          |
| 1.3 Objects, Classes, and Inheritance . . . . .                       | 17          |
| 1.4 Vector and Matrix Objects . . . . .                               | 24          |
| 1.5 Some Further Conventions and Capabilities . . . . .               | 30          |
| <b>2 Solution of Linear Algebraic Equations</b>                       | <b>37</b>   |
| 2.0 Introduction . . . . .  | 37          |
| 2.1 Gauss-Jordan Elimination . . . . .                                | 41          |
| 2.2 Gaussian Elimination with Backsubstitution . . . . .              | 46          |
| 2.3 LU Decomposition and Its Applications . . . . .                   | 48          |
| 2.4 Tridiagonal and Band-Diagonal Systems of Equations . . . . .      | 56          |
| 2.5 Iterative Improvement of a Solution to Linear Equations . . . . . | 61          |
| 2.6 Singular Value Decomposition . . . . .                            | 65          |
| 2.7 Sparse Linear Systems . . . . .                                   | 75          |
| 2.8 Vandermonde Matrices and Toeplitz Matrices . . . . .              | 93          |
| 2.9 Cholesky Decomposition . . . . .                                  | 100         |
| 2.10 QR Decomposition . . . . .                                       | 102         |
| 2.11 Is Matrix Inversion an $N^3$ Process? . . . . .                  | 106         |
| <b>3 Interpolation and Extrapolation</b>                              | <b>110</b>  |
| 3.0 Introduction . . . . .  | 110         |
| 3.1 Preliminaries: Searching an Ordered Table . . . . .               | 114         |
| 3.2 Polynomial Interpolation and Extrapolation . . . . .              | 118         |
| 3.3 Cubic Spline Interpolation . . . . .                              | 120         |
| 3.4 Rational Function Interpolation and Extrapolation . . . . .       | 124         |

|          |  |            |
|----------|--|------------|
| 3.5      | Coefficients of the Interpolating Polynomial . . . . .                                     | 129        |
| 3.6      | Interpolation on a Grid in Multidimensions . . . . .                                       | 132        |
| 3.7      | Interpolation on Scattered Data in Multidimensions . . . . .                               | 139        |
| 3.8      | Laplace Interpolation . . . . .  | 150        |
| <b>4</b> | <b>Integration of Functions</b>  | <b>155</b> |
| 4.0      | Introduction . . . . .   | 155        |
| 4.1      | Classical Formulas for Equally Spaced Abscissas . . . . .                                  | 156        |
| 4.2      | Elementary Algorithms . . . . .  | 162        |
| 4.3      | Romberg Integration . . . . .  | 166        |
| 4.4      | Improper Integrals . . . . .   | 167        |
| 4.5      | Quadrature by Variable Transformation . . . . .  | 172        |
| 4.6      | Gaussian Quadratures and Orthogonal Polynomials . . . . .                                  | 179        |
| 4.7      | Adaptive Quadrature . . . . .  | 194        |
| 4.8      | Multidimensional Integrals . . . . .   | 196        |
| <b>5</b> | <b>Evaluation of Functions</b>   | <b>201</b> |
| 5.0      | Introduction . . . . .   | 201        |
| 5.1      | Polynomials and Rational Functions . . . . .   | 201        |
| 5.2      | Evaluation of Continued Fractions . . . . .  | 206        |
| 5.3      | Series and Their Convergence . . . . .   | 209        |
| 5.4      | Recurrence Relations and Clenshaw's Recurrence Formula . . . . .                           | 219        |
| 5.5      | Complex Arithmetic . . . . .   | 225        |
| 5.6      | Quadratic and Cubic Equations . . . . .  | 227        |
| 5.7      | Numerical Derivatives . . . . .  | 229        |
| 5.8      | Chebyshev Approximation . . . . .  | 233        |
| 5.9      | Derivatives or Integrals of a Chebyshev-Approximated Function .                            | 240        |
| 5.10     | Polynomial Approximation from Chebyshev Coefficients . . . . .                             | 241        |
| 5.11     | Economization of Power Series . . . . .  | 243        |
| 5.12     | Padé Approximants . . . . .  | 245        |
| 5.13     | Rational Chebyshev Approximation . . . . .   | 247        |
| 5.14     | Evaluation of Functions by Path Integration . . . . .                                      | 251        |
| <b>6</b> | <b>Special Functions</b>   | <b>255</b> |
| 6.0      | Introduction . . . . .   | 255        |
| 6.1      | Gamma Function, Beta Function, Factorials, Binomial Coefficients                           | 256        |
| 6.2      | Incomplete Gamma Function and Error Function . . . . .                                     | 259        |
| 6.3      | Exponential Integrals . . . . .  | 266        |
| 6.4      | Incomplete Beta Function . . . . .   | 270        |
| 6.5      | Bessel Functions of Integer Order . . . . .  | 274        |
| 6.6      | Bessel Functions of Fractional Order, Airy Functions, Spherical Bessel Functions . . . . . | 283        |
| 6.7      | Spherical Harmonics . . . . .  | 292        |
| 6.8      | Fresnel Integrals, Cosine and Sine Integrals . . . . .                                     | 297        |
| 6.9      | Dawson's Integral . . . . .  | 302        |
| 6.10     | Generalized Fermi-Dirac Integrals . . . . .  | 304        |
| 6.11     | Inverse of the Function $x \log(x)$ . . . . .  | 307        |
| 6.12     | Elliptic Integrals and Jacobian Elliptic Functions . . . . .                               | 309        |

---

|           |   |            |
|-----------|---|------------|
| 6.13      | Hypergeometric Functions . . . . .                                    | 318        |
| 6.14      | Statistical Functions . . . . .                                       | 320        |
| <b>7</b>  | <b>Random Numbers</b>   | <b>340</b> |
| 7.0       | Introduction . . . . .  | 340        |
| 7.1       | Uniform Deviates . . . . .  | 341        |
| 7.2       | Completely Hashing a Large Array . . . . .                            | 358        |
| 7.3       | Deviates from Other Distributions . . . . .                           | 361        |
| 7.4       | Multivariate Normal Deviates . . . . .                                | 378        |
| 7.5       | Linear Feedback Shift Registers . . . . .                             | 380        |
| 7.6       | Hash Tables and Hash Memories . . . . .                               | 386        |
| 7.7       | Simple Monte Carlo Integration . . . . .                              | 397        |
| 7.8       | Quasi- (that is, Sub-) Random Sequences . . . . .                     | 403        |
| 7.9       | Adaptive and Recursive Monte Carlo Methods . . . . .                  | 410        |
| <b>8</b>  | <b>Sorting and Selection</b>  | <b>419</b> |
| 8.0       | Introduction . . . . .  | 419        |
| 8.1       | Straight Insertion and Shell's Method . . . . .                       | 420        |
| 8.2       | Quicksort . . . . .   | 423        |
| 8.3       | Heapsort . . . . .  | 426        |
| 8.4       | Indexing and Ranking . . . . .  | 428        |
| 8.5       | Selecting the $M$ th Largest . . . . .                                | 431        |
| 8.6       | Determination of Equivalence Classes . . . . .                        | 439        |
| <b>9</b>  | <b>Root Finding and Nonlinear Sets of Equations</b>                   | <b>442</b> |
| 9.0       | Introduction . . . . .  | 442        |
| 9.1       | Bracketing and Bisection . . . . .                                    | 445        |
| 9.2       | Secant Method, False Position Method, and Ridders' Method . . . . .   | 449        |
| 9.3       | Van Wijngaarden-Dekker-Brent Method . . . . .                         | 454        |
| 9.4       | Newton-Raphson Method Using Derivative . . . . .                      | 456        |
| 9.5       | Roots of Polynomials . . . . .  | 463        |
| 9.6       | Newton-Raphson Method for Nonlinear Systems of Equations . . . . .    | 473        |
| 9.7       | Globally Convergent Methods for Nonlinear Systems of Equations        | 477        |
| <b>10</b> | <b>Minimization or Maximization of Functions</b>                      | <b>487</b> |
| 10.0      | Introduction . . . . .  | 487        |
| 10.1      | Initially Bracketing a Minimum . . . . .                              | 490        |
| 10.2      | Golden Section Search in One Dimension . . . . .                      | 492        |
| 10.3      | Parabolic Interpolation and Brent's Method in One Dimension . . . . . | 496        |
| 10.4      | One-Dimensional Search with First Derivatives . . . . .               | 499        |
| 10.5      | Downhill Simplex Method in Multidimensions . . . . .                  | 502        |
| 10.6      | Line Methods in Multidimensions . . . . .                             | 507        |
| 10.7      | Direction Set (Powell's) Methods in Multidimensions . . . . .         | 509        |
| 10.8      | Conjugate Gradient Methods in Multidimensions . . . . .               | 515        |
| 10.9      | Quasi-Newton or Variable Metric Methods in Multidimensions . . . . .  | 521        |
| 10.10     | Linear Programming: The Simplex Method . . . . .                      | 526        |
| 10.11     | Linear Programming: Interior-Point Methods . . . . .                  | 537        |
| 10.12     | Simulated Annealing Methods . . . . .                                 | 549        |
| 10.13     | Dynamic Programming . . . . .   | 555        |

|   |            |
|---|------------|
| <b>11 Eigensystems</b>  | <b>563</b> |
| 11.0 Introduction . . . . .   | 563        |
| 11.1 Jacobi Transformations of a Symmetric Matrix . . . . .   | 570        |
| 11.2 Real Symmetric Matrices . . . . .  | 576        |
| 11.3 Reduction of a Symmetric Matrix to Tridiagonal Form: Givens and Householder Reductions . . . . . | 578        |
| 11.4 Eigenvalues and Eigenvectors of a Tridiagonal Matrix . . . . .                                   | 583        |
| 11.5 Hermitian Matrices . . . . .   | 590        |
| 11.6 Real Nonsymmetric Matrices . . . . .   | 590        |
| 11.7 The <i>QR</i> Algorithm for Real Hessenberg Matrices . . . . .                                   | 596        |
| 11.8 Improving Eigenvalues and/or Finding Eigenvectors by Inverse Iteration . . . . .                 | 597        |
| <b>12 Fast Fourier Transform</b>  | <b>600</b> |
| 12.0 Introduction . . . . .   | 600        |
| 12.1 Fourier Transform of Discretely Sampled Data . . . . .   | 605        |
| 12.2 Fast Fourier Transform (FFT) . . . . .   | 608        |
| 12.3 FFT of Real Functions . . . . .  | 617        |
| 12.4 Fast Sine and Cosine Transforms . . . . .  | 620        |
| 12.5 FFT in Two or More Dimensions . . . . .  | 627        |
| 12.6 Fourier Transforms of Real Data in Two and Three Dimensions . .                                  | 631        |
| 12.7 External Storage or Memory-Local FFTs . . . . .  | 637        |
| <b>13 Fourier and Spectral Applications</b>   | <b>640</b> |
| 13.0 Introduction . . . . .   | 640        |
| 13.1 Convolution and Deconvolution Using the FFT . . . . .  | 641        |
| 13.2 Correlation and Autocorrelation Using the FFT . . . . .  | 648        |
| 13.3 Optimal (Wiener) Filtering with the FFT . . . . .  | 649        |
| 13.4 Power Spectrum Estimation Using the FFT . . . . .  | 652        |
| 13.5 Digital Filtering in the Time Domain . . . . .   | 667        |
| 13.6 Linear Prediction and Linear Predictive Coding . . . . .   | 673        |
| 13.7 Power Spectrum Estimation by the Maximum Entropy (All-Poles) Method . . . . .                    | 681        |
| 13.8 Spectral Analysis of Unevenly Sampled Data . . . . .   | 685        |
| 13.9 Computing Fourier Integrals Using the FFT . . . . .  | 692        |
| 13.10 Wavelet Transforms . . . . .  | 699        |
| 13.11 Numerical Use of the Sampling Theorem . . . . .   | 717        |
| <b>14 Statistical Description of Data</b>   | <b>720</b> |
| 14.0 Introduction . . . . .   | 720        |
| 14.1 Moments of a Distribution: Mean, Variance, Skewness, and So Forth                                | 721        |
| 14.2 Do Two Distributions Have the Same Means or Variances? . . . .                                   | 726        |
| 14.3 Are Two Distributions Different? . . . . .   | 730        |
| 14.4 Contingency Table Analysis of Two Distributions . . . . .  | 741        |
| 14.5 Linear Correlation . . . . .   | 745        |
| 14.6 Nonparametric or Rank Correlation . . . . .  | 748        |
| 14.7 Information-Theoretic Properties of Distributions . . . . .                                      | 754        |
| 14.8 Do Two-Dimensional Distributions Differ? . . . . .   | 762        |

---

|   |            |
|---|------------|
| 14.9 Savitzky-Golay Smoothing Filters . . . . .                         | 766        |
| <b>15 Modeling of Data</b>  | <b>773</b> |
| 15.0 Introduction . . . . .   | 773        |
| 15.1 Least Squares as a Maximum Likelihood Estimator . . . . .          | 776        |
| 15.2 Fitting Data to a Straight Line . . . . .                          | 780        |
| 15.3 Straight-Line Data with Errors in Both Coordinates . . . . .       | 785        |
| 15.4 General Linear Least Squares . . . . .                             | 788        |
| 15.5 Nonlinear Models . . . . .   | 799        |
| 15.6 Confidence Limits on Estimated Model Parameters . . . . .          | 807        |
| 15.7 Robust Estimation . . . . .  | 818        |
| 15.8 Markov Chain Monte Carlo . . . . .                                 | 824        |
| 15.9 Gaussian Process Regression . . . . .                              | 836        |
| <b>16 Classification and Inference</b>                                  | <b>840</b> |
| 16.0 Introduction . . . . .   | 840        |
| 16.1 Gaussian Mixture Models and k-Means Clustering . . . . .           | 842        |
| 16.2 Viterbi Decoding . . . . .   | 850        |
| 16.3 Markov Models and Hidden Markov Modeling . . . . .                 | 856        |
| 16.4 Hierarchical Clustering by Phylogenetic Trees . . . . .            | 868        |
| 16.5 Support Vector Machines . . . . .                                  | 883        |
| <b>17 Integration of Ordinary Differential Equations</b>                | <b>899</b> |
| 17.0 Introduction . . . . .   | 899        |
| 17.1 Runge-Kutta Method . . . . .                                       | 907        |
| 17.2 Adaptive Stepsize Control for Runge-Kutta . . . . .                | 910        |
| 17.3 Richardson Extrapolation and the Bulirsch-Stoer Method . . . . .   | 921        |
| 17.4 Second-Order Conservative Equations . . . . .                      | 928        |
| 17.5 Stiff Sets of Equations . . . . .                                  | 931        |
| 17.6 Multistep, Multivalue, and Predictor-Corrector Methods . . . . .   | 942        |
| 17.7 Stochastic Simulation of Chemical Reaction Networks . . . . .      | 946        |
| <b>18 Two-Point Boundary Value Problems</b>                             | <b>955</b> |
| 18.0 Introduction . . . . .   | 955        |
| 18.1 The Shooting Method . . . . .                                      | 959        |
| 18.2 Shooting to a Fitting Point . . . . .                              | 962        |
| 18.3 Relaxation Methods . . . . .                                       | 964        |
| 18.4 A Worked Example: Spheroidal Harmonics . . . . .                   | 971        |
| 18.5 Automated Allocation of Mesh Points . . . . .                      | 981        |
| 18.6 Handling Internal Boundary Conditions or Singular Points . . . . . | 983        |
| <b>19 Integral Equations and Inverse Theory</b>                         | <b>986</b> |
| 19.0 Introduction . . . . .   | 986        |
| 19.1 Fredholm Equations of the Second Kind . . . . .                    | 989        |
| 19.2 Volterra Equations . . . . .                                       | 992        |
| 19.3 Integral Equations with Singular Kernels . . . . .                 | 995        |
| 19.4 Inverse Problems and the Use of A Priori Information . . . . .     | 1001       |
| 19.5 Linear Regularization Methods . . . . .                            | 1006       |
| 19.6 Backus-Gilbert Method . . . . .                                    | 1014       |

---

|   |             |
|---|-------------|
| 19.7 Maximum Entropy Image Restoration . . . . .                                | 1016        |
| <b>20 Partial Differential Equations</b>  | <b>1024</b> |
| 20.0 Introduction . . . . .   | 1024        |
| 20.1 Flux-Conservative Initial Value Problems . . . . .                         | 1031        |
| 20.2 Diffusive Initial Value Problems . . . . .                                 | 1043        |
| 20.3 Initial Value Problems in Multidimensions . . . . .                        | 1049        |
| 20.4 Fourier and Cyclic Reduction Methods for Boundary Value Problems . . . . . | 1053        |
| 20.5 Relaxation Methods for Boundary Value Problems . . . . .                   | 1059        |
| 20.6 Multigrid Methods for Boundary Value Problems . . . . .                    | 1066        |
| 20.7 Spectral Methods . . . . .   | 1083        |
| <b>21 Computational Geometry</b>  | <b>1097</b> |
| 21.0 Introduction . . . . .   | 1097        |
| 21.1 Points and Boxes . . . . .   | 1099        |
| 21.2 KD Trees and Nearest-Neighbor Finding . . . . .                            | 1101        |
| 21.3 Triangles in Two and Three Dimensions . . . . .                            | 1111        |
| 21.4 Lines, Line Segments, and Polygons . . . . .                               | 1117        |
| 21.5 Spheres and Rotations . . . . .  | 1128        |
| 21.6 Triangulation and Delaunay Triangulation . . . . .                         | 1131        |
| 21.7 Applications of Delaunay Triangulation . . . . .                           | 1141        |
| 21.8 Quadtrees and Octrees: Storing Geometrical Objects . . . . .               | 1149        |
| <b>22 Less-Numerical Algorithms</b>   | <b>1160</b> |
| 22.0 Introduction . . . . .   | 1160        |
| 22.1 Plotting Simple Graphs . . . . .   | 1160        |
| 22.2 Diagnosing Machine Parameters . . . . .                                    | 1163        |
| 22.3 Gray Codes . . . . .   | 1166        |
| 22.4 Cyclic Redundancy and Other Checksums . . . . .                            | 1168        |
| 22.5 Huffman Coding and Compression of Data . . . . .                           | 1175        |
| 22.6 Arithmetic Coding . . . . .  | 1181        |
| 22.7 Arithmetic at Arbitrary Precision . . . . .                                | 1185        |
| <b>Index</b>  | <b>1195</b> |

## Preface to the Third Edition (2007)

“I was just going to say, when I was interrupted...” begins Oliver Wendell Holmes in the second series of his famous essays, *The Autocrat of the Breakfast Table*. The interruption referred to was a gap of 25 years. In our case, as the autocrats of *Numerical Recipes*, the gap between our second and third editions has been “only” 15 years. Scientific computing has changed enormously in that time.

The first edition of *Numerical Recipes* was roughly coincident with the first commercial success of the personal computer. The second edition came at about the time that the Internet, as we know it today, was created. Now, as we launch the third edition, the practice of science and engineering, and thus scientific computing, has been profoundly altered by the mature Internet and Web. It is no longer difficult to find *somebody’s* algorithm, and usually free code, for almost any conceivable scientific application. The critical questions have instead become, “How does it work?” and “Is it any good?” Correspondingly, the second edition of *Numerical Recipes* has come to be valued more and more for its text explanations, concise mathematical derivations, critical judgments, and advice, and less for its code implementations *per se*.

Recognizing the change, we have expanded and improved the text in many places in this edition and added many completely new sections. We seriously considered leaving the code out entirely, or making it available only on the Web. However, in the end, we decided that without code, it wouldn’t be *Numerical Recipes*. That is, without code you, the reader, could never know whether our advice was in fact honest, implementable, and practical. Many discussions of algorithms in the literature and on the Web omit crucial details that can only be uncovered by actually coding (our job) or reading compilable code (your job). Also, we needed actual code to teach and illustrate the large number of lessons about object-oriented programming that are implicit and explicit in this edition.

Our wholehearted embrace of a style of object-oriented computing for scientific applications should be evident throughout this book. We say “*a style*,” because, contrary to the claims of various self-appointed experts, there can be no one rigid style of programming that serves all purposes, not even all scientific purposes. Our style is ecumenical. If a simple, global, C-style function will fill the need, then we use it. On the other hand, you will find us building some fairly complicated structures for something as complicated as, e.g., integrating ordinary differential equations. For more on the approach taken in this book, see §1.3 – §1.5.

In bringing the text up to date, we have luckily not had to bridge a full 15-year gap. Significant modernizations were incorporated into the second edition versions in Fortran 90 (1996)\* and C++ (2002), in which, notably, the last vestiges of unit-based arrays were expunged in favor of C-style zero-based indexing. Only with this third edition, however, have we incorporated a substantial amount (several hundred pages!) of completely new material. Highlights include:

- a new chapter on classification and inference, including such topics as Gaussian mixture models, hidden Markov modeling, hierarchical clustering (phylogenetic trees), and support vector machines

---

\*“Alas, poor Fortran 90! We knew him, Horatio: a programming language of infinite jest, of most excellent fancy: he hath borne us on his back a thousand times.”

- a new chapter on computational geometry, including topics like KD trees, quad- and octrees, Delaunay triangulation and applications, and many useful algorithms for lines, polygons, triangles, spheres, etc.
- many new statistical distributions, with pdfs, cdfs, and inverse cdfs
- an expanded treatment of ODEs, emphasizing recent advances, and with completely new routines
- much expanded sections on uniform random deviates and on deviates from many other statistical distributions
- an introduction to spectral and pseudospectral methods for PDEs
- interior point methods for linear programming
- more on sparse matrices
- interpolation on scattered data in multidimensions
- curve interpolation in multidimensions
- quadrature by variable transformation and adaptive quadrature
- more on Gaussian quadratures and orthogonal polynomials
- more on accelerating the convergence of series
- improved incomplete gamma and beta functions and new inverse functions
- improved spherical harmonics and fast spherical harmonic transforms
- generalized Fermi-Dirac integrals
- multivariate Gaussian deviates
- algorithms and implementations for hash memory functions
- incremental quantile estimation
- chi-square with small numbers of counts
- dynamic programming
- hard and soft error correction and Viterbi decoding
- eigensystem routines for real, nonsymmetric matrices
- multitaper methods for power spectral estimation
- wavelets on the interval
- information-theoretic properties of distributions
- Markov chain Monte Carlo
- Gaussian process regression and kriging
- stochastic simulation of chemical reaction networks
- code for plotting simple graphs from within programs

The *Numerical Recipes* Web site, [www.nr.com](http://www.nr.com), is one of the oldest active sites on the Internet, as evidenced by its two-letter domain name. We will continue to make the Web site useful to readers of this edition. Go there to find the latest bug reports, to purchase the machine-readable source code, or to participate in our readers' forum. With this third edition, we also plan to offer, by subscription, a completely electronic version of *Numerical Recipes* — accessible via the Web, downloadable, printable, and, unlike any paper version, always up to date with the latest corrections. Since the electronic version does not share the page limits of the print version, it will grow over time by the addition of completely new sections, available only electronically. This, we think, is the future of *Numerical Recipes* and perhaps of technical reference books generally. If it sounds interesting to you, look at <http://www.nr.com/electronic>.

This edition also incorporates some “user-friendly” typographical and stylistic improvements: Color is used for headings and to highlight executable code. For code, a label in the margin gives the name of the source file in the machine-readable distribution. Instead of printing repetitive `#include` statements, we provide a con-

venient Web tool at <http://www.nr.com/dependencies> that will generate exactly the statements you need for any combination of routines. Subsections are now numbered and referred to by number. References to journal articles now include, in most cases, the article title, as an aid to easy Web searching. Many references have been updated; but we have kept references to the grand old literature of classical numerical analysis when we think that books and articles deserve to be remembered.

## Acknowledgments

Regrettably, over 15 years, we were not able to maintain a systematic record of the many dozens of colleagues and readers who have made important suggestions, pointed us to new material, corrected errors, and otherwise improved the *Numerical Recipes* enterprise. It is a tired cliché to say that “you know who you are.” Actually, in most cases, we know who you are, and we are grateful. But a list of names would be incomplete, and therefore offensive to those whose contributions are no less important than those listed. We apologize to both groups, those we might have listed and those we might have missed.

We prepared this book for publication on Windows and Linux machines, generally with Intel Pentium processors, using LaTeX in the TeTeX and MiKTeX implementations. Packages used include amsmath, amsfonts, txfonts, and graphicx, among others. Our principal development environments were Microsoft Visual Studio / Microsoft Visual C++ and GNU C++. We used the SourceJammer cross-platform source control system. Many tasks were automated with Perl scripts. We could not live without GNU Emacs. To all the developers: “You know who you are,” and we thank you.

Research by the authors on computational methods was supported in part by the U.S. National Science Foundation and the U.S. Department of Energy.

## Preface to the Second Edition (1992)

Our aim in writing the original edition of *Numerical Recipes* was to provide a book that combined general discussion, analytical mathematics, algorithmics, and actual working programs. The success of the first edition puts us now in a difficult, though hardly unenviable, position. We wanted, then and now, to write a book that is informal, fearlessly editorial, unesoteric, and above all useful. There is a danger that, if we are not careful, we might produce a second edition that is weighty, balanced, scholarly, and boring.

It is a mixed blessing that we know more now than we did six years ago. Then, we were making educated guesses, based on existing literature and our own research, about which numerical techniques were the most important and robust. Now, we have the benefit of direct feedback from a large reader community. Letters to our alter-ego enterprise, Numerical Recipes Software, are in the thousands per year. (Please, *don't telephone* us.) Our post office box has become a magnet for letters pointing out that we have omitted some particular technique, well known to be important in a particular field of science or engineering. We value such letters and digest them carefully, especially when they point us to specific references in the literature.

The inevitable result of this input is that this second edition of *Numerical Recipes* is substantially larger than its predecessor, in fact about 50% larger in both words and number of included programs (the latter now numbering well over 300). “Don’t let the book grow in size,” is the advice that we received from several wise colleagues. We have tried to follow the intended spirit of that advice, even as we violate the letter of it. We have not lengthened, or increased in difficulty, the book’s principal discussions of mainstream topics. Many new topics are presented at this same accessible level. Some topics, both from the earlier edition and new to this one, are now set in smaller type that labels them as being “advanced.” The reader who ignores such advanced sections completely will not, we think, find any lack of continuity in the shorter volume that results.

Here are some highlights of the new material in this second edition:

- a new chapter on integral equations and inverse methods
- a detailed treatment of multigrid methods for solving elliptic partial differential equations
- routines for band-diagonal linear systems
- improved routines for linear algebra on sparse matrices
- Cholesky and  $QR$  decomposition
- orthogonal polynomials and Gaussian quadratures for arbitrary weight functions
- methods for calculating numerical derivatives
- Padé approximants and rational Chebyshev approximation
- Bessel functions, and modified Bessel functions, of fractional order and several other new special functions
- improved random number routines
- quasi-random sequences
- routines for adaptive and recursive Monte Carlo integration in high-dimensional spaces
- globally convergent methods for sets of nonlinear equations
- simulated annealing minimization for continuous control spaces

- fast Fourier transform (FFT) for real data in two and three dimensions
- fast Fourier transform using external storage
- improved fast cosine transform routines
- wavelet transforms
- Fourier integrals with upper and lower limits
- spectral analysis on unevenly sampled data
- Savitzky-Golay smoothing filters
- fitting straight line data with errors in both coordinates
- a two-dimensional Kolmogorov-Smirnoff test
- the statistical bootstrap method
- embedded Runge-Kutta-Fehlberg methods for differential equations
- high-order methods for stiff differential equations
- a new chapter on “less-numerical” algorithms, including Huffman and arithmetic coding, arbitrary precision arithmetic, and several other topics

Consult the Preface to the first edition, following, or the Contents, for a list of the more “basic” subjects treated.

## Acknowledgments

It is not possible for us to list by name here all the readers who have made useful suggestions; we are grateful for these. In the text, we attempt to give specific attribution for ideas that appear to be original and are not known in the literature. We apologize in advance for any omissions.

Some readers and colleagues have been particularly generous in providing us with ideas, comments, suggestions, and programs for this second edition. We especially want to thank George Rybicki, Philip Pinto, Peter Lepage, Robert Lupton, Douglas Eardley, Ramesh Narayan, David Spergel, Alan Oppenheim, Sallie Baliunas, Scott Tremaine, Glennys Farrar, Steven Block, John Peacock, Thomas Loredo, Matthew Choptuik, Gregory Cook, L. Samuel Finn, P. Deufhard, Harold Lewis, Peter Weinberger, David Syer, Richard Ferch, Steven Ebstein, Bradley Keister, and William Gould. We have been helped by Nancy Lee Snyder’s mastery of a complicated  $\text{\TeX}$  manuscript. We express appreciation to our editors Lauren Cowles and Alan Harvey at Cambridge University Press, and to our production editor Russell Hahn. We remain, of course, grateful to the individuals acknowledged in the Preface to the first edition.

Special acknowledgment is due to programming consultant Seth Finkelstein, who wrote, rewrote, or influenced many of the routines in this book, as well as in its Fortran-language twin and the companion Example books. Our project has benefited enormously from Seth’s talent for detecting, and following the trail of, even very slight anomalies (often compiler bugs, but occasionally our errors), and from his good programming sense. To the extent that this edition of *Numerical Recipes in C* has a more graceful and “C-like” programming style than its predecessor, most of the credit goes to Seth. (Of course, we accept the blame for the Fortranish lapses that still remain.)

We prepared this book for publication on DEC and Sun workstations running the UNIX operating system and on a 486/33 PC compatible running MS-DOS 5.0 / Windows 3.0. We enthusiastically recommend the principal software used: GNU Emacs,  $\text{\TeX}$ , Perl, Adobe Illustrator, and PostScript. Also used were a variety of C

compilers — too numerous (and sometimes too buggy) for individual acknowledgement. It is a sobering fact that our standard test suite (exercising all the routines in this book) has uncovered compiler bugs in many of the compilers tried. When possible, we work with developers to see that such bugs get fixed; we encourage interested compiler developers to contact us about such arrangements.

WHP and SAT acknowledge the continued support of the U.S. National Science Foundation for their research on computational methods. DARPA support is acknowledged for §13.10 on wavelets.

## Preface to the First Edition (1985)

We call this book *Numerical Recipes* for several reasons. In one sense, this book is indeed a “cookbook” on numerical computation. However, there is an important distinction between a cookbook and a restaurant menu. The latter presents choices among complete dishes in each of which the individual flavors are blended and disguised. The former — and this book — reveals the individual ingredients and explains how they are prepared and combined.

Another purpose of the title is to connote an eclectic mixture of presentational techniques. This book is unique, we think, in offering, for each topic considered, a certain amount of general discussion, a certain amount of analytical mathematics, a certain amount of discussion of algorithmics, and (most important) actual implementations of these ideas in the form of working computer routines. Our task has been to find the right balance among these ingredients for each topic. You will find that for some topics we have tilted quite far to the analytic side; this where we have felt there to be gaps in the “standard” mathematical training. For other topics, where the mathematical prerequisites are universally held, we have tilted toward more in-depth discussion of the nature of the computational algorithms, or toward practical questions of implementation.

We admit, therefore, to some unevenness in the “level” of this book. About half of it is suitable for an advanced undergraduate course on numerical computation for science or engineering majors. The other half ranges from the level of a graduate course to that of a professional reference. Most cookbooks have, after all, recipes at varying levels of complexity. An attractive feature of this approach, we think, is that the reader can use the book at increasing levels of sophistication as his/her experience grows. Even inexperienced readers should be able to use our most advanced routines as black boxes. Having done so, we hope that these readers will subsequently go back and learn what secrets are inside.

If there is a single dominant theme in this book, it is that practical methods of numerical computation can be simultaneously efficient, clever, and — important — clear. The alternative viewpoint, that efficient computational methods must necessarily be so arcane and complex as to be useful only in “black box” form, we firmly reject.

Our purpose in this book is thus to open up a large number of computational black boxes to your scrutiny. We want to teach you to take apart these black boxes and to put them back together again, modifying them to suit your specific needs. We assume that you are mathematically literate, i.e., that you have the normal mathematical preparation associated with an undergraduate degree in a physical science, or engineering, or economics, or a quantitative social science. We assume that you know how to program a computer. We do not assume that you have any prior formal knowledge of numerical analysis or numerical methods.

The scope of *Numerical Recipes* is supposed to be “everything up to, but not including, partial differential equations.” We honor this in the breach: First, we *do* have one introductory chapter on methods for partial differential equations. Second, we obviously cannot include *everything* else. All the so-called “standard” topics of a numerical analysis course have been included in this book: linear equations, interpolation and extrapolation, integration, nonlinear root finding, eigensystems, and ordinary differential equations. Most of these topics have been taken beyond their

standard treatments into some advanced material that we have felt to be particularly important or useful.

Some other subjects that we cover in detail are not usually found in the standard numerical analysis texts. These include the evaluation of functions and of particular special functions of higher mathematics; random numbers and Monte Carlo methods; sorting; optimization, including multidimensional methods; Fourier transform methods, including FFT methods and other spectral methods; two chapters on the statistical description and modeling of data; and two-point boundary value problems, both shooting and relaxation methods.

## Acknowledgments

Many colleagues have been generous in giving us the benefit of their numerical and computational experience, in providing us with programs, in commenting on the manuscript, or with general encouragement. We particularly wish to thank George Rybicki, Douglas Eardley, Philip Marcus, Stuart Shapiro, Paul Horowitz, Bruce Musicus, Irwin Shapiro, Stephen Wolfram, Henry Abarbanel, Larry Smarr, Richard Muller, John Bahcall, and A.G.W. Cameron.

We also wish to acknowledge two individuals whom we have never met: Forman Acton, whose 1970 textbook *Numerical Methods That Work* (New York: Harper and Row) has surely left its stylistic mark on us; and Donald Knuth, both for his series of books on *The Art of Computer Programming* (Reading, MA: Addison-Wesley), and for TeX, the computer typesetting language that immensely aided production of this book.

Research by the authors on computational methods was supported in part by the U.S. National Science Foundation.

## **License and Legal Information**

You must read this section if you intend to use the code in this book on a computer. You'll need to read the following Disclaimer of Warranty, acquire a Numerical Recipes software license, and get the code onto your computer. Without the license, which can be the limited, free "immediate license" under terms described below, this book is intended as a text and reference book, for reading and study purposes only.

For purposes of licensing, the electronic version of the *Numerical Recipes* book is equivalent to the paper version. It is not equivalent to a Numerical Recipes software license, which must still be acquired separately or as part of a combined electronic product. For information on Numerical Recipes electronic products, go to <http://www.nr.com/electronic>.

### ***Disclaimer of Warranty***

**We make no warranties, express or implied, that the programs contained in this volume are free of error, or are consistent with any particular standard of merchantability, or that they will meet your requirements for any particular application. They should not be relied on for solving a problem whose incorrect solution could result in injury to a person or loss of property. If you do use the programs in such a manner, it is at your own risk. The authors and publisher disclaim all liability for direct or consequential damages resulting from your use of the programs.**

### ***The Restricted, Limited Free License***

We recognize that readers may have an immediate, urgent wish to copy a small amount of code from this book for use in their own applications. If you personally keyboard no more than 10 routines from this book into your computer, then we authorize you (and only you) to use those routines (and only those routines) on that single computer. You are not authorized to transfer or distribute the routines to any other person or computer, nor to have any other person keyboard the programs into a computer on your behalf. We do not want to hear bug reports from you, because experience has shown that *virtually all* reported bugs in such cases are typing errors! This free license is not a GNU General Public License.

### ***Regular Licenses***

When you purchase a code subscription or one-time code download from the Numerical Recipes Web site (<http://www.nr.com>), or when you buy physical Numerical Recipes media published by Cambridge University Press, you automatically get a *Numerical Recipes Personal Single-User License*. This license lets you personally use Numerical Recipes code on any one computer at a time, but not to allow anyone else access to the code. You may also, under this license, transfer precompiled, executable programs incorporating the code to other, unlicensed, users or computers, providing that (i) your application is noncommercial (i.e., does not involve the selling of your program for a fee); (ii) the programs were first developed, compiled, and successfully run by you; and (iii) our routines are bound into the programs in such a manner that they cannot be accessed as individual routines and cannot practicably be

unbound and used in other programs. That is, under this license, your program user must not be able to use our programs as part of a program library or “mix-and-match” workbench. See the Numerical Recipes Web site for further details.

Businesses and organizations that purchase code subscriptions, downloads, or media, and that thus acquire one or more Numerical Recipes Personal Single-User Licenses, may permanently assign those licenses, in the number acquired, to individual employees. In most cases, however, businesses and organizations will instead want to purchase Numerical Recipes licenses “by the seat,” allowing them to be used by a pool of individuals rather than being individually permanently assigned. See <http://www.nr.com/licenses> for information on such licenses.

Instructors at accredited educational institutions who have adopted this book for a course may purchase on behalf of their students one-semester subscriptions to both the electronic version of the *Numerical Recipes* book and to the Numerical Recipes code. During the subscription term, students may download, view, save, and print all of the book and code. See <http://www.nr.com/licenses> for further information.

Other types of corporate licenses are also available. Please see the Numerical Recipes Web site.

## About Copyrights on Computer Programs

Like artistic or literary compositions, computer programs are protected by copyright. Generally it is an infringement for you to copy into your computer a program from a copyrighted source. (It is also not a friendly thing to do, since it deprives the program’s author of compensation for his or her creative effort.) Under copyright law, all “derivative works” (modified versions, or translations into another computer language) also come under the same copyright as the original work.

Copyright does not protect ideas, but only the expression of those ideas in a particular form. In the case of a computer program, the ideas consist of the program’s methodology and algorithm, including the necessary sequence of steps adopted by the programmer. The expression of those ideas is the program source code (particularly any arbitrary or stylistic choices embodied in it), its derived object code, and any other derivative works.

If you analyze the ideas contained in a program, and then express those ideas in your own completely different implementation, then that new program implementation belongs to you. That is what we have done for those programs in this book that are not entirely of our own devising. When programs in this book are said to be “based” on programs published in copyright sources, we mean that the ideas are the same. The expression of these ideas as source code is our own. We believe that no material in this book infringes on an existing copyright.

## Trademarks

Several registered trademarks appear within the text of this book. Words that are known to be trademarks are shown with an initial capital letter. However, the capitalization of any word is not an expression of the authors’ or publisher’s opinion as to whether or not it is subject to proprietary rights, nor is it to be regarded as affecting the validity of any trademark.

Numerical Recipes, NR, and nr.com (when identifying our products) are trademarks of Numerical Recipes Software.

## Attributions

The fact that ideas are legally “free as air” in no way supersedes the ethical requirement that ideas be credited to their known originators. When programs in this book are based on known sources, whether copyrighted or in the public domain, published or “handed-down,” we have attempted to give proper attribution. Unfortunately, the lineage of many programs in common circulation is often unclear. We would be grateful to readers for new or corrected information regarding attributions, which we will attempt to incorporate in subsequent printings.

## Routines by Chapter and Section

Previous editions included a table of all the routines in the book, along with a short description, arranged by chapter and section. This information is now available as an interactive Web page at <http://www.nr.com/routines>. The following illustration gives the idea.

| sort by name   | sort by section | sort by file |      |            |
|----------------|-----------------|--------------|------|------------|
| Internal Name  | Chapter         | Section      | Page | File       |
| flimoon        | 1               | 1.0          | 2    | calendar.h |
| julday         | 1               | 1.0          | 6    | calendar.h |
| caldat         | 1               | 1.0          | 6    | calendar.h |
| gaussj         | 2               | 2.1          | 43   | gaussj.h   |
| ludcmp         | 2               | 2.3          | 51   | ludcmp.h   |
| ludcmp_ludcmp  | 2               | 2.3          | 51   | ludcmp.h   |
| ludcmp_solve   | 2               | 2.3          | 52   | ludcmp.h   |
| ludcmp_inverse | 2               | 2.3          | 53   | ludcmp.h   |



# Preliminaries

## 1.0 Introduction

This book is supposed to teach you methods of numerical computing that are practical, efficient, and (insofar as possible) elegant. We presume throughout this book that you, the reader, have particular tasks that you want to get done. We view our job as educating you on how to proceed. Occasionally we may try to reroute you briefly onto a particularly beautiful side road; but by and large, we will guide you along main highways that lead to practical destinations.

Throughout this book, you will find us fearlessly editorializing, telling you what you should and shouldn't do. This prescriptive tone results from a conscious decision on our part, and we hope that you will not find it irritating. We do not claim that our advice is infallible! Rather, we are reacting against a tendency, in the textbook literature of computation, to discuss every possible method that has ever been invented, without ever offering a practical judgment on relative merit. We do, therefore, offer you our practical judgments whenever we can. As you gain experience, you will form your own opinion of how reliable our advice is. Be assured that it is not perfect!

We presume that you are able to read computer programs in C++. The question, “Why C++?”, is a complicated one. For now, suffice it to say that we wanted a language with a C-like syntax in the small (because that is most universally readable by our audience), which had a rich set of facilities for object-oriented programming (because that is an emphasis of this third edition), and which was highly backward-compatible with some old, but established and well-tested, tricks in numerical programming. That pretty much led us to C++, although Java (and the closely related C#) were close contenders.

Honesty compels us to point out that in the 20-year history of *Numerical Recipes*, we have never been correct in our predictions about the future of programming languages for scientific programming, *not once!* At various times we convinced ourselves that the wave of the scientific future would be ...Fortran ...Pascal ...C ...Fortran 90 (or 95 or 2000) ...Mathematica ...Matlab ...C++ or Java .... Indeed, several of these enjoy continuing success and have significant followings (not including Pascal!). None, however, currently command a majority, or even a large plurality, of scientific users.

With this edition, we are no longer trying to predict the future of programming languages. Rather, we want a serviceable way of communicating ideas about scientific programming. We hope that these ideas transcend the language, C++, in which we are expressing them.

When we include programs in the text, they look like this:

calendar.h

```
void flmoon(const Int n, const Int nph, Int &jd, Doub &frac) {
```

Our routines begin with an introductory comment summarizing their purpose and explaining their calling sequence. This routine calculates the phases of the moon. Given an integer *n* and a code *nph* for the phase desired (*nph* = 0 for new moon, 1 for first quarter, 2 for full, 3 for last quarter), the routine returns the Julian Day Number *jd*, and the fractional part of a day *frac* to be added to it, of the *n<sup>th</sup>* such phase since January, 1900. Greenwich Mean Time is assumed.

```
    const Doub RAD=3.141592653589793238/180.0;
    Int i;
    Doub am,as,c,t,t2,xtra;
    c=n+nph/4.0;                                This is how we comment an individual line.
    t=c/1236.85;
    t2=t*t;
    as=359.2242+29.105356*c;
    am=306.0253+385.816918*c+0.010730*t2;      You aren't really intended to understand
    jd=2415020+28*n+7*nph;                      this algorithm, but it does work!
    xtra=0.75933+1.53058868*c+((1.178e-4)-(1.55e-7)*t)*t2;
    if (nph == 0 || nph == 2)
        xtra += (0.1734-3.93e-4*t)*sin(RAD*as)-0.4068*sin(RAD*am);
    else if (nph == 1 || nph == 3)
        xtra += (0.1721-4.0e-4*t)*sin(RAD*as)-0.6280*sin(RAD*am);
    else throw("nph is unknown in flmoon");        This indicates an error condition.
    i=Int(xtra >= 0.0 ? floor(xtra) : ceil(xtra-1.0));
    jd += i;
    frac=xtra-i;
}
```

Note our convention of handling all errors and exceptional cases with a statement like `throw("some error message");`. Since C++ has no built-in exception class for type `char*`, executing this statement results in a fairly rude program abort. However we will explain in §1.5.1 how to get a more elegant result without having to modify the source code.

## 1.0.1 What Numerical Recipes Is Not

We want to use the platform of this introductory section to emphasize what *Numerical Recipes* is not:

1. *Numerical Recipes* is not a textbook on programming, or on best programming practices, or on C++, or on software engineering. We are not opposed to good programming. We try to communicate good programming practices whenever we can — but only incidentally to our main purpose, which is to teach how practical numerical methods actually work. The unity of style and subordination of function to standardization that is necessary in a good programming (or software engineering) textbook is just not what we have in mind for this book. Each section in this book has as its focus a particular computational method. Our goal is to explain and illustrate that method as clearly as possible. No single programming style is best for all such methods, and, accordingly, our style varies from section to section.

2. *Numerical Recipes* is not a program library. That may surprise you if you are one of the many scientists and engineers who use our source code regularly. What

makes our code *not* a program library is that it demands a greater intellectual commitment from the user than a program library ought to do. If you haven't read a routine's accompanying section and gone through the routine line by line to understand how it works, then you use it at great peril! We consider this a feature, not a bug, because our primary purpose is to teach methods, not provide packaged solutions. This book does not include formal exercises, in part because we consider each section's code to be the exercise: If you can understand each line of the code, then you have probably mastered the section.

There are some fine commercial program libraries [1,2] and integrated numerical environments [3-5] available. Comparable free resources are available, both program libraries [6,7] and integrated environments [8-10]. When you want a packaged solution, we recommend that you use one of these. *Numerical Recipes* is intended as a cookbook for cooks, not a restaurant menu for diners.

## 1.0.2 Frequently Asked Questions

This section is for people who want to jump right in.

### 1. How do I use NR routines with my own program?

The easiest way is to put a bunch of #include's at the top of your program. Always start with `nr3.h`, since that defines some necessary utility classes and functions (see §1.4 for a lot more about this). For example, here's how you compute the mean and variance of the Julian Day numbers of the first 20 full moons after January 1900. (Now *there's* a useful pair of quantities!)

```
#include "nr3.h"
#include "calendar.h"
#include "moment.h"

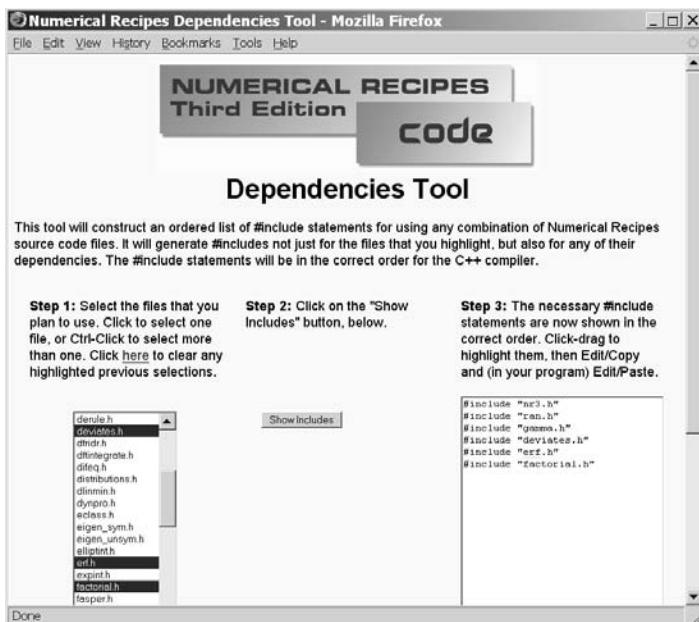
Int main(void) {
    const Int NTOT=20;
    Int i,jd,nph=2;
    Doub frac,ave,vrnce;
    VecDoub data(NTOT);
    for (i=0;i<NTOT;i++) {
        fmloon(i,nph,jd,frac);
        data[i]=jd;
    }
    avevar(data,ave,vrnce);
    cout << "Average = " << setw(12) << ave;
    cout << " Variance = " << setw(13) << vrnce << endl;
    return 0;
}
```

Be sure that the NR source code files are in a place that your compiler can find them to #include. Compile and run the above file. (We can't tell you how to do this part.) Output should be something like this:

```
Average = 2.41532e+06 Variance = 30480.7
```

### 2. Yes, but where do I actually get the NR source code as computer files?

You can buy a code subscription, or a one-time code download, at the Web site <http://www.nr.com>, or you can get the code on media published by Cambridge



**Figure 1.0.1.** The interactive page located at <http://www.nr.com/dependencies> sorts out the dependencies for any combination of *Numerical Recipes* routines, giving an ordered list of the necessary #include files.

University Press (e.g., from Amazon.com or your favorite online or physical bookstore). The code comes with a personal, single-user license (see License and Legal Information on p. xix). The reason that the book (or its electronic version) and the code license are sold separately is to help keep down the price of each. Also, making these products separate meets the needs of more users: Your company or educational institution may have a site license — ask them.

### 3. How do I know which files to #include? It's hard to sort out the dependencies among all the routines.

In the margin next to each code listing is the name of the source code file that it is in. Make a list of the source code files that you are using. Then go to <http://www.nr.com/dependencies> and click on the name of each source code file. The interactive Web page will return a list of the necessary #includes, in the correct order, to satisfy all dependencies. Figure 1.0.1 will give you an idea of how this works.

### 4. What is all this Doub, Int, VecDoub, etc., stuff?

We always use defined types, not built-in types, so that they can be redefined if necessary. The definitions are in `nr3.h`. Generally, as you can guess, `Doub` means `double`, `Int` means `int`, and so forth. Our convention is to begin all defined types with an uppercase letter. `VecDoub` is a vector class type. Details on our types are in §1.4.

### 5. What are Numerical Recipes Webnotes?

*Numerical Recipes* Webnotes are documents, accessible on the Web, that include some code implementation listings, or other highly specialized topics, that are not included in the paper version of the book. A list of all Webnotes is at

| Tested Operating Systems and Compilers |  |
|--|--|
| O/S                                    | Compiler                                   |
| Microsoft Windows XP SP2               | Visual C++ ver. 14.00 (Visual Studio 2005) |
| Microsoft Windows XP SP2               | Visual C++ ver. 13.10 (Visual Studio 2003) |
| Microsoft Windows XP SP2               | Intel C++ Compiler ver. 9.1                |
| Novell SUSE Linux 10.1                 | GNU GCC (g++) ver. 4.1.0                   |
| Red Hat Enterprise Linux 4 (64-bit)    | GNU GCC (g++) ver. 3.4.6 and ver. 4.1.0    |
| Red Hat Linux 7.3                      | Intel C++ Compiler ver. 9.1                |
| Apple Mac OS X 10.4 (Tiger) Intel Core | GNU GCC (g++) ver. 4.0.1                   |

<http://www.nr.com/webnotes>. By moving some specialized material into Webnotes, we are able to keep down the size and price of the paper book. Webnotes are automatically included in the electronic version of the book; see next question.

6. *I am a post-paper person. I want Numerical Recipes on my laptop. Where do I get the complete, fully electronic version?*

A fully electronic version of *Numerical Recipes* is available by annual subscription. You can subscribe instead of, or in addition to, owning a paper copy of the book. A subscription is accessible via the Web, downloadable, printable, and, unlike any paper version, always up to date with the latest corrections. Since the electronic version does not share the page limits of the printed version, it will grow over time by the addition of completely new sections, available only electronically. This, we think, is the future of *Numerical Recipes* and perhaps of technical reference books generally. We anticipate various electronic formats, changing with time as technologies for display and rights management continuously improve: We place a big emphasis on user convenience and usability. See <http://www.nr.com/electronic> for further information.

7. *Are there bugs in NR?*

Of course! By now, most NR code has the benefit of long-time use by a large user community, but new bugs are sure to creep in. Look at <http://www.nr.com> for information about known bugs, or to report apparent new ones.

### 1.0.3 Computational Environment and Program Validation

The code in this book should run without modification on any compiler that implements the ANSI/ISO C++ standard, as described, for example, in Stroustrup's book [11].

As surrogates for the large number of hardware and software configurations, we have tested all the code in this book on the combinations of operating systems and compilers shown in the table above.

In validating the code, we have taken it directly from the machine-readable form of the book's manuscript, so that we have tested exactly what is printed. (This does not, of course, mean that the code is bug-free!)

## 1.0.4 About References

You will find references, and suggestions for further reading, listed at the end of most sections of this book. References are cited in the text by bracketed numbers like this [12].

We do not pretend to any degree of bibliographical completeness in this book. For topics where a substantial secondary literature exists (discussion in textbooks, reviews, etc.) we often limit our references to a few of the more useful secondary sources, especially those with good references to the primary literature. Where the existing secondary literature is insufficient, we give references to a few primary sources that are intended to serve as starting points for further reading, not as complete bibliographies for the field.

Since progress is ongoing, it is inevitable that our references for many topics are already, or will soon become, out of date. We have tried to include older references that are good for “forward” Web searching: A search for more recent papers that cite the references given should lead you to the most current work.

Web references and URLs present a problem, because there is no way for us to guarantee that they will still be there when you look for them. A date like 2007+ means “it was there in 2007.” We try to give citations that are complete enough for you to find the document by Web search, even if it has moved from the location listed.

The order in which references are listed is not necessarily significant. It reflects a compromise between listing cited references in the order cited, and listing suggestions for further reading in a roughly prioritized order, with the most useful ones first.

## 1.0.5 About “Advanced Topics”

Material set in smaller type, like this, signals an “advanced topic,” either one outside of the main argument of the chapter, or else one requiring of you more than the usual assumed mathematical background, or else (in a few cases) a discussion that is more speculative or an algorithm that is less well tested. Nothing important will be lost if you skip the advanced topics on a first reading of the book.

Here is a function for getting the Julian Day Number from a calendar date.

calendar.h

```
Int julday(const Int mm, const Int id, const Int iyyy) {
    In this routine julday returns the Julian Day Number that begins at noon of the calendar date
    specified by month mm, day id, and year iyyy, all integer variables. Positive year signifies A.D.;
    negative, B.C. Remember that the year after 1 B.C. was 1 A.D.

    const Int IGREG=15+31*(10+12*1582);           Gregorian Calendar adopted Oct. 15, 1582.
    Int ja,jul,jy=iyyy,jm;
    if (jy == 0) throw("julday: there is no year zero.");
    if (jy < 0) ++jy;
    if (mm > 2) {
        jm=mm+1;
    } else {
        --jy;
        jm=mm+13;
    }
    jul = Int(floor(365.25*jy)+floor(30.6001*jm)+id+1720995);
    if (id+31*(mm+12*iyyy) >= IGREG) {           Test whether to change to Gregorian Cal-
        ja=Int(0.01*jy);                           endar.
        jul += 2-ja+Int(0.25*ja);
    }
    return jul;
}
```

And here is its inverse.

```
void caldat(const Int julian, Int &mm, Int &id, Int &iyyy) {
Inverse of the function julday given above. Here julian is input as a Julian Day Number, and
the routine outputs mm,id, and iyyy as the month, day, and year on which the specified Julian
Day started at noon.
    const Int IREG=2299161;
    Int ja,jalpha,jb,jc,jd,je;

    if (julian >= IREG) {      Cross-over to Gregorian Calendar produces this correc-
        jalpha=Int((Doub(julian-1867216)-0.25)/36524.25);           tion.
        ja=julian+1+jalpha-Int(0.25*jalpha);
    } else if (julian < 0) {    Make day number positive by adding integer number of
        ja=julian+36525*(1-julian/36525);       Julian centuries, then subtract them off
    } else                      at the end.
        ja=julian;
    jb=ja+1524;
    jc=Int(6680.0+(Doub(jb-2439870)-122.1)/365.25);
    jd=Int(365*jc+(0.25*jc));
    je=Int((jb-jd)/30.6001);
    id=jb-jd-Int(30.6001*je);
    mm=je-1;
    if (mm > 12) mm -= 12;
    iyyy=jc-4715;
    if (mm > 2) --iyyy;
    if (iyyy <= 0) --iyyy;
    if (julian < 0) iyyy -= 100*(1-julian/36525);
}
```

calendar.h

As an exercise, you might try using these functions, along with `f1moon` in §1.0, to search for future occurrences of a full moon on Friday the 13th. (Answers, in time zone GMT minus 5: 9/13/2019 and 8/13/2049.) For additional calendrical algorithms, applicable to various historical calendars, see [13].

#### CITED REFERENCES AND FURTHER READING:

- Visual Numerics, 2007+, *IMSL Numerical Libraries*, at <http://www.vni.com>.[1]
- Numerical Algorithms Group, 2007+, *NAG Numerical Library*, at <http://www.nag.co.uk>.[2]
- Wolfram Research, Inc., 2007+, *Mathematica*, at <http://www.wolfram.com>.[3]
- The MathWorks, Inc., 2007+, *MATLAB*, at <http://www.mathworks.com>.[4]
- Maplesoft, Inc., 2007+, *Maple*, at <http://www.maplesoft.com>.[5]
- GNU Scientific Library, 2007+, at <http://www.gnu.org/software/gsl>.[6]
- Netlib Repository, 2007+, at <http://www.netlib.org>.[7]
- Scilab Scientific Software Package, 2007+, at <http://www.scilab.org>.[8]
- GNU Octave, 2007+, at <http://www.gnu.org/software/octave>.[9]
- R Software Environment for Statistical Computing and Graphics, 2007+, at  
<http://www.r-project.org>.[10]
- Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley).[11]
- Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell).[12]
- Hatcher, D.A. 1984, "Simple Formulae for Julian Day Numbers and Calendar Dates," *Quarterly Journal of the Royal Astronomical Society*, vol. 25, pp. 53–55; see also *op. cit.* 1985, vol. 26, pp. 151–155, and 1986, vol. 27, pp. 506–507.[13]

## 1.1 Error, Accuracy, and Stability

Computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed number of *bits* (binary digits) or *bytes* (groups of 8 bits). Almost all computers allow the programmer a choice among several different such *representations* or *data types*. Data types can differ in the number of bits utilized (the *wordlength*), but also in the more fundamental respect of whether the stored number is represented in *fixed-point* (like `int`) or *floating-point* (like `float` or `double`) format.

A number in integer representation is exact. Arithmetic between numbers in integer representation is also exact, with the provisos that (i) the answer is not outside the range of (usually, signed) integers that can be represented, and (ii) that division is interpreted as producing an integer result, throwing away any integer remainder.

### 1.1.1 Floating-Point Representation

In a floating-point representation, a number is represented internally by a sign bit  $S$  (interpreted as plus or minus), an exact integer exponent  $E$ , and an exactly represented binary mantissa  $M$ . Taken together these represent the number

$$S \times M \times b^{E-e} \quad (1.1.1)$$

where  $b$  is the base of the representation ( $b = 2$  almost always), and  $e$  is the *bias* of the exponent, a fixed integer constant for any given machine and representation.

|                      | $S$ | $E$    | $F$     | Value                                  |
|----------------------|-----|--------|---------|--|
| float                | any | 1–254  | any     | $(-1)^S \times 2^{E-127} \times 1.F$   |
|                      | any | 0      | nonzero | $(-1)^S \times 2^{-126} \times 0.F^*$  |
|                      | 0   | 0      | 0       | + 0.0                                  |
|                      | 1   | 0      | 0       | - 0.0                                  |
|                      | 0   | 255    | 0       | + $\infty$                             |
|                      | 1   | 255    | 0       | - $\infty$                             |
|                      | any | 255    | nonzero | NaN                                    |
| double               | any | 1–2046 | any     | $(-1)^S \times 2^{E-1023} \times 1.F$  |
|                      | any | 0      | nonzero | $(-1)^S \times 2^{-1022} \times 0.F^*$ |
|                      | 0   | 0      | 0       | + 0.0                                  |
|                      | 1   | 0      | 0       | - 0.0                                  |
|                      | 0   | 2047   | 0       | + $\infty$                             |
|                      | 1   | 2047   | 0       | - $\infty$                             |
|                      | any | 2047   | nonzero | NaN                                    |
| *unnormalized values |     |        |         |  |

Several floating-point bit patterns can in principle represent the same number. If  $b = 2$ , for example, a mantissa with leading (high-order) zero bits can be left-shifted, i.e., multiplied by a power of 2, if the exponent is decreased by a compensating amount. Bit patterns that are “as left-shifted as they can be” are termed *normalized*.

Virtually all modern processors share the same floating-point data representations, namely those specified in IEEE Standard 754-1985 [1]. (For some discussion of nonstandard processors, see §22.2.) For 32-bit float values, the exponent is represented in 8 bits (with  $e = 127$ ), the mantissa in 23; for 64-bit double values, the exponent is 11 bits (with  $e = 1023$ ), the mantissa, 52. An additional trick is used for the mantissa for most nonzero floating values: Since the high-order bit of a properly normalized mantissa is *always* one, the stored mantissa bits are viewed as being preceded by a “phantom” bit with the value 1. In other words, the mantissa  $M$  has the numerical value  $1.F$ , where  $F$  (called the *fraction*) consists of the bits (23 or 52 in number) that are actually stored. This trick gains a little “bit” of precision.

Here are some examples of IEEE 754 representations of double values:

$$\begin{aligned}
 0\ 011111111111\ 0000 \text{ (+ 48 more zeros)} &= +1 \times 2^{1023-1023} \times 1.0_2 = 1. \\
 1\ 011111111111\ 0000 \text{ (+ 48 more zeros)} &= -1 \times 2^{1023-1023} \times 1.0_2 = -1. \\
 0\ 011111111111\ 1000 \text{ (+ 48 more zeros)} &= +1 \times 2^{1023-1023} \times 1.1_2 = 1.5 \\
 0\ 100000000000\ 0000 \text{ (+ 48 more zeros)} &= +1 \times 2^{1024-1023} \times 1.0_2 = 2. \\
 0\ 100000000001\ 1010 \text{ (+ 48 more zeros)} &= +1 \times 2^{1025-1023} \times 1.1010_2 = 6.5
 \end{aligned} \tag{1.1.2}$$

You can examine the representation of any value by code like this:

```

union Udoub {
    double d;
    unsigned char c[8];
};

void main() {
    Udoub u;
    u.d = 6.5;
    for (int i=7;i>=0;i--) printf("%02x",u.c[i]);
    printf("\n");
}

```

This is C, and deprecated style, but it will work. On most processors, including Intel Pentium and successors, you’ll get the printed result 401a000000000000, which (writing out each hex digit as four binary digits) is the last line in equation (1.1.2). If you get the bytes (groups of two hex digits) in reverse order, then your processor is *big-endian* instead of *little-endian*: The IEEE 754 standard does not specify (or care) in which order the bytes in a floating-point value are stored.

The IEEE 754 standard includes representations of positive and negative infinity, positive and negative zero (treated as computationally equivalent, of course), and also NaN (“not a number”). The table on the previous page gives details of how these are represented.

The reason for representing some *unnormalized* values, as shown in the table, is to make “underflow to zero” more graceful. For a sequence of smaller and smaller values, after you pass the smallest normalizable value (with magnitude  $2^{-127}$  or  $2^{-1023}$ ; see table), you start right-shifting the leading bit of the mantissa. Although

you gradually lose precision, you don't actually underflow to zero until 23 or 52 bits later.

When a routine needs to know properties of the floating-point representation, it can reference the `numeric_limits` class, which is part of the C++ Standard Library. For example, `numeric_limits<double>::min()` returns the smallest normalized double value, usually  $2^{-1022} \approx 2.23 \times 10^{-308}$ . For more on this, see §22.2.

### 1.1.2 Roundoff Error

Arithmetic among numbers in floating-point representation is not exact, even if the operands happen to be exactly represented (i.e., have exact values in the form of equation 1.1.1). For example, two floating numbers are added by first right-shifting (dividing by two) the mantissa of the smaller (in magnitude) one and simultaneously increasing its exponent until the two operands have the same exponent. Low-order (least significant) bits of the smaller operand are lost by this shifting. If the two operands differ too greatly in magnitude, then the smaller operand is effectively replaced by zero, since it is right-shifted to oblivion.

The smallest (in magnitude) floating-point number that, when added to the floating-point number 1.0, produces a floating-point result different from 1.0 is termed the *machine accuracy*  $\epsilon_m$ . IEEE 754 standard `float` has  $\epsilon_m$  about  $1.19 \times 10^{-7}$ , while `double` has about  $2.22 \times 10^{-16}$ . Values like this are accessible as, e.g., `numeric_limits <double>::epsilon()`. (A more detailed discussion of machine characteristics is in §22.2.) Roughly speaking, the machine accuracy  $\epsilon_m$  is the fractional accuracy to which floating-point numbers are represented, corresponding to a change of one in the least significant bit of the mantissa. Pretty much any arithmetic operation among floating numbers should be thought of as introducing an additional fractional error of at least  $\epsilon_m$ . This type of error is called *roundoff error*.

It is important to understand that  $\epsilon_m$  is not the smallest floating-point number that can be represented on a machine. *That* number depends on how many bits there are in the exponent, while  $\epsilon_m$  depends on how many bits there are in the mantissa.

Roundoff errors accumulate with increasing amounts of calculation. If, in the course of obtaining a calculated value, you perform  $N$  such arithmetic operations, you *might* be so lucky as to have a total roundoff error on the order of  $\sqrt{N}\epsilon_m$ , if the roundoff errors come in randomly up or down. (The square root comes from a random-walk.) However, this estimate can be very badly off the mark for two reasons:

(1) It very frequently happens that the regularities of your calculation, or the peculiarities of your computer, cause the roundoff errors to accumulate preferentially in one direction. In this case the total will be of order  $N\epsilon_m$ .

(2) Some especially unfavorable occurrences can vastly increase the roundoff error of single operations. Generally these can be traced to the subtraction of two very nearly equal numbers, giving a result whose only significant bits are those (few) low-order ones in which the operands differed. You might think that such a “co-incidental” subtraction is unlikely to occur. Not always so. Some mathematical expressions magnify its probability of occurrence tremendously. For example, in the familiar formula for the solution of a quadratic equation,

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad (1.1.3)$$

the addition becomes delicate and roundoff-prone whenever  $b > 0$  and  $|ac| \ll b^2$ . (In §5.6 we will learn how to avoid the problem in this particular case.)

### 1.1.3 Truncation Error

Roundoff error is a characteristic of computer hardware. There is another, different, kind of error that is a characteristic of the program or algorithm used, independent of the hardware on which the program is executed. Many numerical algorithms compute “discrete” approximations to some desired “continuous” quantity. For example, an integral is evaluated numerically by computing a function at a discrete set of points, rather than at “every” point. Or, a function may be evaluated by summing a finite number of leading terms in its infinite series, rather than all infinity terms. In cases like this, there is an adjustable parameter, e.g., the number of points or of terms, such that the “true” answer is obtained only when that parameter goes to infinity. Any practical calculation is done with a finite, but sufficiently large, choice of that parameter.

The discrepancy between the true answer and the answer obtained in a practical calculation is called the *truncation error*. Truncation error would persist even on a hypothetical, “perfect” computer that had an infinitely accurate representation and no roundoff error. As a general rule there is not much that a programmer can do about roundoff error, other than to choose algorithms that do not magnify it unnecessarily (see discussion of “stability” below). Truncation error, on the other hand, is entirely under the programmer’s control. In fact, it is only a slight exaggeration to say that clever minimization of truncation error is practically the entire content of the field of numerical analysis!

Most of the time, truncation error and roundoff error do not strongly interact with one another. A calculation can be imagined as having, first, the truncation error that it would have if run on an infinite-precision computer, “plus” the roundoff error associated with the number of operations performed.

### 1.1.4 Stability

Sometimes an otherwise attractive method can be *unstable*. This means that any roundoff error that becomes “mixed into” the calculation at an early stage is successively magnified until it comes to swamp the true answer. An unstable method would be useful on a hypothetical, perfect computer; but in this imperfect world it is necessary for us to require that algorithms be stable — or if unstable that we use them with great caution.

Here is a simple, if somewhat artificial, example of an unstable algorithm: Suppose that it is desired to calculate all integer powers of the so-called “Golden Mean,” the number given by

$$\phi \equiv \frac{\sqrt{5} - 1}{2} \approx 0.61803398 \quad (1.1.4)$$

It turns out (you can easily verify) that the powers  $\phi^n$  satisfy a simple recursion relation,

$$\phi^{n+1} = \phi^{n-1} - \phi^n \quad (1.1.5)$$

Thus, knowing the first two values  $\phi^0 = 1$  and  $\phi^1 = 0.61803398$ , we can successively apply (1.1.5) performing only a single subtraction, rather than a slower

multiplication by  $\phi$ , at each stage.

Unfortunately, the recurrence (1.1.5) also has *another* solution, namely the value  $-\frac{1}{2}(\sqrt{5} + 1)$ . Since the recurrence is linear, and since this undesired solution has magnitude greater than unity, any small admixture of it introduced by roundoff errors will grow exponentially. On a typical machine, using a 32-bit float, (1.1.5) starts to give completely wrong answers by about  $n = 16$ , at which point  $\phi^n$  is down to only  $10^{-4}$ . The recurrence (1.1.5) is *unstable* and cannot be used for the purpose stated.

We will encounter the question of stability in many more sophisticated guises later in this book.

#### CITED REFERENCES AND FURTHER READING:

- IEEE, 1985, *ANSI/IEEE Std 754–1985: IEEE Standard for Binary Floating-Point Numbers* (New York: IEEE).[1]
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 1.
- Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2.
- Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §1.3.
- Wilkinson, J.H. 1964, *Rounding Errors in Algebraic Processes* (Englewood Cliffs, NJ: Prentice-Hall).

## 1.2 C Family Syntax

Not only C++, but also Java, C#, and (to varying degrees) other computer languages, share a lot of small-scale syntax with the older C language [1]. By small scale, we mean operations on built-in types, simple expressions, control structures, and the like. In this section, we review some of the basics, give some hints on good programming, and mention some of our conventions and habits.

### 1.2.1 Operators

A first piece of advice might seem superfluous if it were not so often ignored: You should learn all the C operators and their precedence and associativity rules. You might not yourself want to write

```
n << 1 | 1
```

as a synonym for  $2*n+1$  (for positive integer n), but you definitely do need to be able to see at a glance that

```
n << 1 + 1
```

is not at all the same thing! Please study the table on the next page while you brush your teeth every night. While the occasional set of unnecessary parentheses, for clarity, is hardly a sin, code that is habitually overparenthesized is annoying and hard to read.

| Operator Precedence and Associativity Rules in C and C++ |                                     |               |
|--|-------------------------------------|---------------|
| <code>::</code>  | scope resolution                    | left-to-right |
| <code>()</code>  | function call                       | left-to-right |
| <code>[]</code>  | array element (subscripting)        |               |
| <code>.</code>   | member selection                    |               |
| <code>-&gt;</code>                                       | member selection (by pointer)       |               |
| <code>++</code>  | post increment                      | right-to-left |
| <code>--</code>  | post decrement                      |               |
| <code>!</code>   | logical not                         | right-to-left |
| <code>~</code>   | bitwise complement                  |               |
| <code>-</code>   | unary minus                         |               |
| <code>++</code>  | pre increment                       |               |
| <code>--</code>  | pre decrement                       |               |
| <code>&amp;</code>                                       | address of                          |               |
| <code>*</code>   | contents of (dereference)           |               |
| <code>new</code>   | create                              |               |
| <code>delete</code>                                      | destroy                             |               |
| <code>(type)</code>                                      | cast to type                        |               |
| <code>sizeof</code>                                      | size in bytes                       |               |
| <code>*</code>   | multiply                            | left-to-right |
| <code>/</code>   | divide                              |               |
| <code>%</code>   | remainder                           |               |
| <code>+</code>   | add                                 | left-to-right |
| <code>-</code>   | subtract                            |               |
| <code>&lt;&lt;</code>                                    | bitwise left shift                  | left-to-right |
| <code>&gt;&gt;</code>                                    | bitwise right shift                 |               |
| <code>&lt;</code>  | arithmetic less than                | left-to-right |
| <code>&gt;</code>  | arithmetic greater than             |               |
| <code>&lt;=</code>                                       | arithmetic less than or equal to    |               |
| <code>&gt;=</code>                                       | arithmetic greater than or equal to |               |
| <code>==</code>  | arithmetic equal                    | left-to-right |
| <code>!=</code>  | arithmetic not equal                |               |
| <code>&amp;</code>                                       | bitwise and                         | left-to-right |
| <code>^</code>   | bitwise exclusive or                | left-to-right |
| <code> </code>   | bitwise or                          | left-to-right |
| <code>&amp;&amp;</code>                                  | logical and                         | left-to-right |
| <code>  </code>  | logical or                          | left-to-right |
| <code>? :</code>   | conditional expression              | right-to-left |
| <code>=</code>   | assignment operator                 | right-to-left |
| also <code>+= -= *= /= %=</code>                         |                                     |               |
| <code>&lt;&lt;= &gt;&gt;= &amp;= ^=  =</code>            |                                     |               |
| <code>,</code>   | sequential expression               | left-to-right |

## 1.2.2 Control Structures

These should all be familiar to you.

**Iteration.** In C family languages simple iteration is performed with a `for` loop, for example

```
for (j=2;j<=1000;j++) {
    b[j]=a[j-1];
    a[j-1]=j;
}
```

It is conventional to indent the block of code that is acted upon by the control structure, leaving the structure itself unindented. We like to put the initial curly brace on the same line as the `for` statement, instead of on the next line. This saves a full line of white space, and our publisher loves us for it.

**Conditional.** The conditional or `if` structure looks, in full generality, like this:

```
if (...) {
    ...
}
else if (...) {
    ...
}
else {
    ...
}
```

However, since compound-statement curly braces are required only when there is more than one statement in a block, the `if` construction can be somewhat less explicit than that shown above. Some care must be exercised in constructing nested `if` clauses. For example, consider the following:

```
if (b > 3)
    if (a > 3) b += 1;
else b -= 1;           /* questionable! */
```

As judged by the indentation used on successive lines, the intent of the writer of this code is the following: ‘If `b` is greater than 3 and `a` is greater than 3, then increment `b`. If `b` is not greater than 3, then decrement `b`.’ According to the rules, however, the actual meaning is ‘If `b` is greater than 3, then evaluate `a`. If `a` is greater than 3, then increment `b`, and if `a` is less than or equal to 3, decrement `b`.’ The point is that an `else` clause is associated with the most recent open `if` statement, no matter how you lay it out on the page. Such confusions in meaning are easily resolved by the inclusion of braces that clarify your intent and improve the program. The above fragment should be written as

```
if (b > 3) {
    if (a > 3) b += 1;
} else {
    b -= 1;
}
```

**While iteration.** Alternative to the `for` iteration is the `while` structure, for example,

---

```
while (n < 1000) {
    n *= 2;
    j += 1;
}
```

The control clause (in this case `n < 1000`) is evaluated before each iteration. If the clause is not true, the enclosed statements will not be executed. In particular, if this code is encountered at a time when `n` is greater than or equal to 1000, the statements will not even be executed once.

**Do-While iteration.** Companion to the `while` iteration is a related control structure that tests its control clause at the *end* of each iteration:

```
do {
    n *= 2;
    j += 1;
} while (n < 1000);
```

In this case, the enclosed statements will be executed at least once, independent of the initial value of `n`.

**Break and Continue.** You use the `break` statement when you have a loop that is to be repeated indefinitely until some condition *tested somewhere in the middle of the loop* (and possibly tested in more than one place) becomes true. At that point you wish to exit the loop and proceed with what comes after it. In C family languages the simple `break` statement terminates execution of the innermost `for`, `while`, `do`, or `switch` construction and proceeds to the next sequential instruction. A typical usage might be

```
for(;;) {
    ...
    if (...) break;
    ...
}
...
(statements before the test)
(statements after the test)
(next sequential instruction)
```

Companion to `break` is `continue`, which transfers program control to the end of the body of the smallest enclosing `for`, `while`, or `do` statement, but *just inside* that body's terminating curly brace. In general, this results in the execution of the next loop test associated with that body.

### 1.2.3 How Tricky Is Too Tricky?

Every programmer is occasionally tempted to write a line or two of code that is so elegantly tricky that all who read it will stand in awe of its author's intelligence. Poetic justice is that it is usually that same programmer who gets stumped, later on, trying to understand his or her own creation. You might momentarily be proud of yourself at writing the single line

```
k=(2-j)*(1+3*j)/2;
```

if you want to permute cyclically one of the values  $j = (0, 1, 2)$  into respectively  $k = (1, 2, 0)$ . You will regret it later, however. Better, and likely also faster, is

```
k=j+1;
if (k == 3) k=0;
```

On the other hand, it can also be a mistake, or at least suboptimal, to be too ploddingly literal, as in

```
switch (j) {
    case 0: k=1; break;
    case 1: k=2; break;
    case 2: k=0; break;
    default: {
        cerr << "unexpected value for j";
        exit(1);
    }
}
```

This (or similar) might be the house style if you are one of  $10^5$  programmers working for a megacorporation, but if you are programming for your own research, or within a small group of collaborators, this kind of style will soon cause you to lose the forest for the trees. You need to find the right personal balance between obscure trickery and boring prolixity. A good rule is that you should always write code that is *slightly less tricky than you are willing to read, but only slightly*.

There is a fine line between being tricky (bad) and being idiomatic (good). *Idioms* are short expressions that are sufficiently common, or sufficiently self-explanatory, that you can use them freely. For example, testing an integer n's even- or odd-ness by

```
if (n & 1) ...
```

is, we think, much preferable to

```
if (n % 2 == 1) ...
```

We similarly like to double a positive integer by writing

```
n <= 1;
```

or construct a mask of n bits by writing

```
(1 << n) - 1
```

and so forth.

Some idioms are worthy of consideration even when they are not so immediately obvious. S.E. Anderson [2] has collected a number of “bit-twiddling hacks,” of which we show three here:

The test

```
if ((v&(v-1))==0) {}           Is a power of 2 or zero.
```

tests whether v is a power of 2. If you care about the case v = 0, you have to write

```
if (v&&((v&(v-1))==0)) {}     Is a power of 2.
```

The idiom

```
for (c=0;v;c++) v &= v - 1;
```

gives as c the number of set (= 1) bits in a positive or unsigned integer v (destroying v in the process). The number of iterations is only as many as the number of bits set.

The idiom

```
v--;
v |= v >> 1; v |= v >> 2; v |= v >> 4; v |= v >> 8; v |= v >> 16;
v++;
```

rounds a positive (or unsigned) 32-bit integer v up to the next power of 2 that is  $\geq v$ .

When we use the bit-twiddling hacks, we'll include an explanatory comment in the code.

### 1.2.4 Utility Macros or Templated Functions

The file nr3.h includes, among other things, definitions for the functions

```
MAX(a,b)
MIN(a,b)
SWAP(a,b)
SIGN(a,b)
```

These are all self-explanatory, except possibly the last. SIGN(a,b) returns a value with the same magnitude as a and the same sign as b. These functions are all implemented as templated inline functions, so that they can be used for all argument types that make sense semantically. Implementation as macros is also possible.

#### CITED REFERENCES AND FURTHER READING:

Harbison, S.P., and Steele, G.L., Jr. 2002, *C: A Reference Manual*, 5th ed. (Englewood Cliffs, NJ: Prentice-Hall).[1]

Anderson, S.E. 2006, "Bit Twiddling Hacks," at <http://graphics.stanford.edu/~seander/bithacks.html>.[2]

## 1.3 Objects, Classes, and Inheritance

An *object* or *class* (the terms are interchangeable) is a program structure that groups together some variables, or functions, or both, in such a way that all the included variables or functions "see" each other and can interact intimately, while most of this internal structure is hidden from other program structures and units. Objects make possible *object-oriented programming* (OOP), which has become recognized as the almost unique successful paradigm for creating complex software. The key insight in OOP is that objects have *state* and *behavior*. The state of the object is described by the values stored in its member variables, while the possible behavior is determined by the member functions. We will use objects in other ways as well.

The terminology surrounding OOP can be confusing. Objects, classes, and structures pretty much refer to the same thing. Member functions in a class are often referred to as *methods* belonging to that class. In C++, objects are defined with either the keyword `class` or the keyword `struct`. These differ, however, in the details of how rigorously they hide the object's internals from public view. Specifically,

```
struct SomeName { ... }
```

is defined as being the same as

```
class SomeName {
public: ... }
```

In this book we *always* use `struct`. This is not because we deprecate the use of `public` and `private` access specifiers in OOP, but only because such access control would add little to understanding the underlying numerical methods that are the focus of this book. In fact, access specifiers could impede your understanding, because

you would be constantly moving things from private to public (and back again) as you program different test cases and want to examine different internal, normally private, variables.

Because our classes are declared by `struct`, not `class`, use of the word “class” is potentially confusing, and we will usually try to avoid it. So “object” means `struct`, which is really a class!

If you are an OOP beginner, it is important to understand the distinction between defining an object and instantiating it. You define an object by writing code like this:

```
struct Twovar {
    Doub a,b;
    Twovar(const Doub aa, const Doub bb) : a(aa), b(bb) {}
    Doub sum() {return a+b;}
    Doub diff() {return a-b;}
};
```

This code does not create a `Twovar` object. It only tells the compiler how to create one when, later in your program, you tell it to do so, for example by a declaration like,

```
Twovar mytwovar(3.,5.);
```

which invokes the `Twovar` constructor and creates an instance of (or *instantiates*) a `Twovar`. In this example, the constructor also sets the internal variables `a` and `b` to 3 and 5, respectively. You can have any number of simultaneously existing, noninteracting, instances:

```
Twovar anothertwovar(4.,6.);
Twovar athirdtwovar(7.,8.);
```

We have already promised you that this book is not a textbook in OOP, or the C++ language; so we will go no farther here. If you need more, good references are [1-4].

### 1.3.1 Simple Uses of Objects

We use objects in various ways, ranging from trivial to quite complex, depending on the needs of the specific numerical method that is being discussed. As mentioned in §1.0, this lack of consistency means that *Numerical Recipes* is not a useful exemplar of a program library (or, in an OOP context, a *class library*). It also means that, somewhere in this book, you can probably find an example of every possible way to think about objects in numerical computing! (We hope that you will find this a plus.)

**Object for Grouping Functions.** Sometimes an object just collects together a group of closely related functions, not too differently from the way that you might use a `namespace`. For example, a simplification of Chapter 6’s object `Erf` looks like:

```
struct Erf {                               No constructor needed.
    Doub erf(Doub x);
    Doub erfc(Doub x);
    Doub inverf(Doub p);
    Doub inverfc(Doub p);
    Doub erfccheb(Doub z);
};
```

As will be explained in §6.2, the first four methods are the ones intended to be called by the user, giving the error function, complementary error function, and the two

corresponding inverse functions. But these methods share some code and also use common code in the last method, `erfccheb`, which the user will normally ignore completely. It therefore makes sense to group the whole collection as an `Erf` object. About the only disadvantage of this is that you must instantiate an `Erf` object before you can use (say) the `erf` function:

```
Erf myerf;
...
Doub y = myerf.erf(3.);
```

Instantiating the object doesn't actually *do* anything here, because `Erf` contains no variables (i.e., has no stored state). It just tells the compiler what local name you are going to use in referring to its member functions. (We would normally use the name `erf` for the instance of `Erf`, but we thought that `erf.erf(3.)` would be confusing in the above example.)

**Object for Standardizing an Interface.** In §6.14 we'll discuss a number of useful standard probability distributions, for example, normal, Cauchy, binomial, Poisson, etc. Each gets its own object definition, for example,

```
struct Cauchydist {
    Doub mu, sig;
    Cauchydist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {}
    Doub p(Doub x);
    Doub cdf(Doub x);
    Doub invcdf(Doub p);
};
```

where the function `p` returns the probability density, the function `cdf` returns the cumulative distribution function (cdf), and the function `invcdf` returns the inverse of the cdf. Because the interface is consistent across all the different probability distributions, you can change which distribution a program is using by changing a single program line, for example from

```
Cauchydist mydist();
```

to

```
Normaldist mydist();
```

All subsequent references to functions like `mydist.p`, `mydist.cdf`, and so on, are thus changed automatically. This is hardly OOP at all, but it can be very convenient.

**Object for Returning Multiple Values.** It often happens that a function computes more than one useful quantity, but you don't know which one or ones the user is actually interested in on that particular function call. A convenient use of objects is to save all the potentially useful results and then let the user grab those that are of interest. For example, a simplified version of the `Fitab` structure in Chapter 15, which fits a straight line  $y = a + bx$  to a set of data points `xx` and `yy`, looks like this:

```
struct Fitab {
    Doub a, b;
    Fitab(const VecDoub &xx, const VecDoub &yy);           Constructor.
};
```

(We'll discuss `VecDoub` and related matters below, in §1.4.) The user calculates the fit by calling the constructor with the data points as arguments,

```
Fitab myfit(xx,yy);
```

Then the two “answers” *a* and *b* are separately available as `myfit.a` and `myfit.b`. We will see more elaborate examples throughout the book.

**Objects That Save Internal State for Multiple Uses.** This is classic OOP, worthy of the name. A good example is Chapter 2’s `LUDcmp` object, which (in abbreviated form) looks like this:

```
struct LUDcmp {
    Int n;
    MatDoub lu;
    LUDcmp(const MatDoub &a); Constructor.
    void solve(const VecDoub &b, VecDoub &x);
    void inverse(MatDoub &ainv);
    Doub det();
};
```

This object is used to solve linear equations and/or invert a matrix. You use it by creating an instance with your matrix *a* as the argument in the constructor. The constructor then computes and stores, in the internal matrix *lu*, a so-called *LU* decomposition of your matrix (see §2.3). Normally you won’t use the matrix *lu* directly (though you could if you wanted to). Rather, you now have available the methods `solve()`, which returns a solution vector *x* for any right-hand side *b*, `inverse()`, which returns the inverse matrix, and `det()`, which returns the determinant of your matrix.

You can call any or all of `LUDcmp`’s methods in any order; you might well want to call `solve` multiple times, with different right-hand sides. If you have more than one matrix in your problem, you create a separate instance of `LUDcmp` for each one, for example,

```
LUDcmp alu(a), aalu(aa);
```

after which `alu.solve()` and `aalu.solve()` are the methods for solving linear equations for each respective matrix, *a* and *aa*; `alu.det()` and `aalu.det()` return the two determinants; and so forth.

We are not finished listing ways to use objects: Several more are discussed in the next few sections.

### 1.3.2 Scope Rules and Object Destruction

This last example, `LUDcmp`, raises the important issue of how to manage an object’s time and memory usage within your program.

For a large matrix, the `LUDcmp` constructor does a lot of computation. You choose exactly where in your program you want this to occur in the obvious way, by putting the declaration

```
LUDcmp alu(a);
```

in just that place. The important distinction between a non-OOP language (like C) and an OOP language (like C++) is that, in the latter, declarations are not passive instructions to the compiler, but executable statements at run-time.

The `LUDcmp` constructor also, for a large matrix, grabs a lot of memory, to store the matrix *lu*. How do you take charge of this? That is, how do you communicate that it should save this state for as long as you might need it for calls to methods like `alu.solve()`, but not indefinitely?

The answer lies in C++'s strict and predictable rules about *scope*. You can start a temporary scope at any point by writing an open bracket, “{”. You end that scope by a matching close bracket, “}”. You can nest scopes in the obvious way. Any objects that are declared within a scope are destroyed (and their memory resources returned) when the end of the scope is reached. An example might look like this:

```

MatDoub a(1000,1000);           Create a big matrix,
VecDoub b(1000),x(1000);       and a couple of vectors.

...
{
    LUdcmp alu(a);
    ...
    alu.solve(b,x);
    ...
}
                                         End temporary scope. Resources in alu are freed.

...
Doub d = alu.det();             ERROR! alu is out of scope.

```

This example presumes that you have some other use for the matrix `a` later on. If not, then the declaration of `a` should itself probably be inside the temporary scope.

Be aware that *all* program blocks delineated by braces are scope units. This includes the main block associated with a function definition and also blocks associated with control structures. In code like this,

```

for (;;) {
    ...
    LUdcmp alu(a);
    ...
}

```

a new instance of `alu` is created at each iteration and then destroyed at the end of that iteration. This might sometimes be what you intend (if the matrix `a` changes on each iteration, for example); but you should be careful not to let it happen unintentionally.

### 1.3.3 Functions and Functors

Many routines in this book take functions as input. For example, the quadrature (integration) routines in Chapter 4 take as input the function  $f(x)$  to be integrated. For a simple case like  $f(x) = x^2$ , you code such a function simply as

```

Doub f(const Doub x) {
    return x*x;
}

```

and pass `f` as an argument to the routine. However, it is often useful to use a more general object to communicate the function to the routine. For example,  $f(x)$  may depend on other variables or parameters that need to be communicated from the calling program. Or the computation of  $f(x)$  may be associated with other sub-calculations or information from other parts of the program. In non-OOP programming, this communication is usually accomplished with global variables that pass the information “over the head” of the routine that receives the function argument `f`.

C++ provides a better and more elegant solution: *function objects* or *functors*. A functor is simply an object in which the operator () has been overloaded to play the role of returning a function value. (There is no relation between this use of the word functor and its different meaning in pure mathematics.) The case  $f(x) = x^2$  would now be coded as

```
struct Square {
    Doub operator()(const Doub x) {
        return x*x;
    }
};
```

To use this with a quadrature or other routine, you declare an instance of `Square`  
`Square g;`

and pass `g` to the routine. Inside the quadrature routine, an invocation of `g(x)` returns the function value in the usual way.

In the above example, there's no point in using a functor instead of a plain function. But suppose you have a parameter in the problem, for example,  $f(x) = cx^p$ , where  $c$  and  $p$  are to be communicated from somewhere else in your program. You can set the parameters via a constructor:

```
struct Contimespow {
    Doub c,p;
    Contimespow(const Doub cc, const Doub pp) : c(cc), p(pp) {}
    Doub operator()(const Doub x) {
        return c*pow(x,p);
    }
};
```

In the calling program, you might declare the instance of `Contimespow` by

`Contimespow h(4.,0.5);`      Communicate `c` and `p` to the functor.

and later pass `h` to the routine. Clearly you can make the functor much more complicated. For example, it can contain other helper functions to aid in the calculation of the function value.

So should we implement all our routines to accept only functors and not functions? Luckily, we don't have to decide. We can write the routines so they can accept *either* a function or a functor. A routine accepting only a function to be integrated from  $a$  to  $b$  might be declared as

`Doub someQuadrature(Doub func(const Doub), const Doub a, const Doub b);`

To allow it to accept either functions or functors, we instead make it a *templated* function:

```
template <class T>
Doub someQuadrature(T &func, const Doub a, const Doub b);
```

Now the compiler figures out whether you are calling `someQuadrature` with a function or a functor and generates the appropriate code. If you call the routine in one place in your program with a function and in another with a functor, the compiler will handle that too.

We will use this capability to pass functors as arguments in many different places in the book where function arguments are required. There is a tremendous gain in flexibility and ease of use.

As a convention, when we write `Ftor`, we mean a functor like `Square` or `Contimespow` above; when we write `fbare`, we mean a “bare” function like `f` above; and when we write `ftor` (all in lower case), we mean an instantiation of a functor, that is, something declared like

`Ftor ftor(...);`      Replace the dots by your parameters, if any.

Of course your names for functors and their instantiations will be different.

Slightly more complicated syntax is involved in passing a function to an *object*

that is templated to accept either a function or functor. So if the object is

```
template <class T>
struct SomeStruct {
    SomeStruct(T &func, ...); constructor
    ...
}
```

we would instantiate it with a functor like this:

```
Ftor ftor;
SomeStruct<Ftor> s(ctor, ...)
```

but with a function like this:

```
SomeStruct<Doub (const Doub)> s(fbare, ...)
```

In this example, `fbare` takes a single `const Doub` argument and returns a `Doub`. You must use the arguments and return type for your specific case, of course.

### 1.3.4 Inheritance

Objects can be defined as deriving from other, already defined, objects. In such *inheritance*, the “parent” class is called a *base class*, while the “child” class is called a *derived class*. A derived class has all the methods and stored state of its base class, plus it can add any new ones.

**“Is-a” Relationships.** The most straightforward use of inheritance is to describe so-called *is-a* relationships. OOP texts are full of examples where the base class is `ZooAnimal` and a derived class is `Lion`. In other words, `Lion` “is-a” `ZooAnimal`. The base class has methods common to all `ZooAnimals`, for example `eat()` and `sleep()`, while the derived class extends the base class with additional methods specific to `Lion`, for example `roar()` and `eat_visitor()`.

In this book we use *is-a* inheritance less often than you might expect. Except in some highly stylized situations, like optimized matrix classes (“triangular matrix is-a matrix”), we find that the diversity of tasks in scientific computing does not lend itself to strict *is-a* hierarchies. There are exceptions, however. For example, in Chapter 7, we define an object `Ran` with methods for returning uniform random deviates of various types (e.g., `Int` or `Doub`). Later in the chapter, we define objects for returning other kinds of random deviates, for example `normal` or `binomial`. These are defined as derived classes of `Ran`, for example,

```
struct Binomialdev : Ran {};
```

so that they can share the machinery already in `Ran`. This is a true *is-a* relationship, because “`binomial` deviate is-a `random` deviate.”

Another example occurs in Chapter 13, where objects `Daub4`, `Daub4i`, and `Daubs` are all derived from the `Wavelet` base class. Here `Wavelet` is an *abstract base class* or *ABC* [1,4] that has no content of its own. Rather, it merely specifies interfaces for all the methods that any `Wavelet` is required to have. The relationship is nevertheless *is-a*: “`Daub4` is-a `Wavelet`”.

**“Prerequisite” Relationships.** Not for any dogmatic reason, but simply because it is convenient, we frequently use inheritance to pass on to an object a set of methods that it needs as prerequisites. This is especially true when the same set of prerequisites is used by more than one object. In this use of inheritance, the base class has no particular `ZooAnimal` unity; it may be a grab-bag. There is not a logical *is-a* relationship between the base and derived classes.

An example in Chapter 10 is `Bracketmethod`, which is a base class for several

minimization routines, but which simply provides a common method for the initial bracketing of a minimum. In Chapter 7, the `Hashtable` object provides prerequisite methods to its derived classes `Hash` and `Mhash`, but one cannot say, “`Mhash` is-a `Hashtable`” in any meaningful way. An extreme example, in Chapter 6, is the base class `Gauleg18`, which does nothing except provide a bunch of constants for Gauss-Legendre integration to derived classes `Beta` and `Gamma`, both of which need them. Similarly, long lists of constants are provided to the routines `StepperDopr853` and `StepperRoss` in Chapter 17 by base classes to avoid cluttering the coding of the algorithms.

**Partial Abstraction.** Inheritance can be used in more complicated or situation-specific ways. For example, consider Chapter 4, where elementary quadrature rules such as `Trapzd` and `Midpnt` are used as building blocks to construct more elaborate quadrature algorithms. The key feature these simple rules share is a mechanism for adding more points to an existing approximation to an integral to get the “next” stage of refinement. This suggests deriving these objects from an abstract base class called `Quadrature`, which specifies that all objects derived from it must have a `next()` method. This is not a complete specification of a common is-a interface; it abstracts only one feature that turns out to be useful.

For example, in §4.6, the `Stiel` object invokes, in different situations, two different quadrature objects, `Trapzd` and `DErule`. These are not interchangeable. They have different constructor arguments and could not easily both be made `ZooAnimals` (as it were). `Stiel` of course knows about their differences. However, one of `Stiel`’s methods, `quad()`, doesn’t (and shouldn’t) know about these differences. It uses only the method `next()`, which exists, with different definitions, in both `Trapzd` and `DErule`.

While there are several different ways to deal with situations like this, an easy one is available once `Trapzd` and `DErule` have been given a common abstract base class `Quadrature` that contains nothing except a virtual interface to `next`. In a case like this, the base class is a minor design feature as far as the implementation of `Stiel` is concerned, almost an afterthought, rather than being the apex of a top-down design. As long as the usage is clear, there is nothing wrong with this.

Chapter 17, which discusses ordinary differential equations, has some even more complicated examples that combine inheritance and templating. We defer further discussion to there.

#### CITED REFERENCES AND FURTHER READING:

- Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley).[1]
- Lippman, S.B., Lajoie, J., and Moo, B.E. 2005, *C++ Primer*, 4th ed. (Boston: Addison-Wesley).[2]
- Keogh, J., and Giannini, M. 2004, *OOP Demystified* (Emeryville, CA: McGraw-Hill/Osborne).[3]
- Cline, M., Lomow, G., and Girou, M. 1999, *C++ FAQs*, 2nd ed. (Boston: Addison-Wesley).[4]

## 1.4 Vector and Matrix Objects

The C++ Standard Library [1] includes a perfectly good `vector<>` template class. About the only criticism that one can make of it is that it is so feature-rich

that some compiler vendors neglect to squeeze the last little bit of performance out of its most elementary operations, for example returning an element by its subscript. That performance is extremely important in scientific applications; its occasional absence in C++ compilers is a main reason that many scientists still (as we write) program in C, or even in Fortran!

Also included in the C++ Standard Library is the class `valarray<>`. At one time, this was supposed to be a vector-like class that was optimized for numerical computation, including some features associated with matrices and multidimensional arrays. However, as reported by one participant,

The `valarray` classes were not designed very well. In fact, nobody tried to determine whether the final specification worked. This happened because nobody felt “responsible” for these classes. The people who introduced `valarrays` to the C++ standard library left the committee a long time before the standard was finished. [1]

The result of this history is that C++, at least now, has a good (but not always reliably optimized) class for vectors and no dependable class at all for matrices or higher-dimensional arrays. What to do? We will adopt a strategy that emphasizes flexibility and assumes only a minimal set of properties for vectors and matrices. We will then provide our own, basic, classes for vectors and matrices. For most compilers, these are at least as efficient as `vector<>` and other vector and matrix classes in common use. But if, for you, they’re not, then it is easy to change to a different set of classes, as we will explain.

### 1.4.1 Typedefs

Flexibility is achieved by having several layers of `typedef` type-indirection, resolved at compile time so that there is no run-time performance penalty. The first level of type-indirection, not just for vectors and matrices but for virtually all variables, is that we use user-defined type names instead of C++ fundamental types. These are defined in `nr3.h`. If you ever encounter a compiler with peculiar built-in types, these definitions are the “hook” for making any necessary changes. The complete list of such definitions is

| <i>NR Type</i> | <i>Usual Definition</i>             | <i>Intent</i>                       |
|----------------|-------------------------------------|-------------------------------------|
| Char           | <code>char</code>                   | 8-bit signed integer                |
| Uchar          | <code>unsigned char</code>          | 8-bit unsigned integer              |
| Int            | <code>int</code>                    | 32-bit signed integer               |
| Uint           | <code>unsigned int</code>           | 32-bit unsigned integer             |
| Llong          | <code>long long int</code>          | 64-bit signed integer               |
| Ullong         | <code>unsigned long long int</code> | 64-bit unsigned integer             |
| Doub           | <code>double</code>                 | 64-bit floating point               |
| Ldoub          | <code>long double</code>            | [reserved for future use]           |
| Complex        | <code>complex&lt;double&gt;</code>  | $2 \times 64$ -bit floating complex |
| Bool           | <code>bool</code>                   | true or false                       |

An example of when you might need to change the `typedefs` in `nr3.h` is if your compiler’s `int` is not 32 bits, or if it doesn’t recognize the type `long long int`.

You might need to substitute vendor-specific types like (in the case of Microsoft) `_int32` and `_int64`.

The second level of type-indirection returns us to the discussion of vectors and matrices. The vector and matrix types that appear in *Numerical Recipes* source code are as follows. Vectors: `VecInt`, `VecUint`, `VecChar`, `VecUchar`, `VecCharp`, `VecLlong`, `VecULLong`, `VecDoub`, `VecDoubp`, `VecComplex`, and `VecBool`. Matrices: `MatInt`, `MatUint`, `MatChar`, `MatUchar`, `MatLlong`, `MatULLong`, `MatDoub`, `MatComplex`, and `MatBool`. These should all be understandable, semantically, as vectors and matrices whose elements are the corresponding user-defined types, above. Those ending in a “p” have elements that are pointers, e.g., `VecCharp` is a vector of pointers to `char`, that is, `char*`. If you are wondering why the above list is not combinatorially complete, it is because we don’t happen to use all possible combinations of `Vec`, `Mat`, fundamental type, and pointer in this book. You can add further analogous types as you need them.

Wait, there’s more! For every vector and matrix type above, we also define types with the same names plus one of the suffixes “`_I`”, “`_O`”, and “`_IO`”, for example `VecDoub_IO`. We use these suffixed types for specifying argument types in function definitions. The meaning, respectively, is that the argument is “input”, “output”, or “both input and output”.\* The `_I` types are automatically defined to be `const`. We discuss this further in §1.5.2 under the topic of `const` correctness.

It may seem capricious for us to define such a long list of types when a much smaller number of templated types would do. The rationale is flexibility: You have a hook into redefining each and every one of the types individually, according to your needs for program efficiency, local coding standards, `const`-correctness, or whatever. In fact, in `nr3.h`, all these types *are* `typedef`’d to one vector and one matrix class, along the following lines:

```
typedef NRvector<Int> VecInt, VecInt_O, VecInt_IO;
typedef const NRvector<Int> VecInt_I;
...
typedef NRvector<Doub> VecDoub, VecDoub_O, VecDoub_IO;
typedef const NRvector<Doub> VecDoub_I;
...
typedef NRmatrix<Int> MatInt, MatInt_O, MatInt_IO;
typedef const NRmatrix<Int> MatInt_I;
...
typedef NRmatrix<Doub> MatDoub, MatDoub_O, MatDoub_IO;
typedef const NRmatrix<Doub> MatDoub_I;
...
```

So (flexibility, again) you can change the definition of one particular type, like `VecDoub`, or else you can change the implementation of all vectors by changing the definition of `NRvector<>`. Or, you can just leave things the way we have them in `nr3.h`. That ought to work fine in 99.9% of all applications.

### 1.4.2 Required Methods for Vector and Matrix Classes

The important thing about the vector and matrix classes is not what names they are `typedef`’d to, but what methods are assumed for them (and are provided in the `NRvector` and `NRmatrix` template classes). For vectors, the assumed methods are a

---

\*This is a bit of history, and derives from Fortran 90’s very useful `INTENT` attributes.

subset of those in the C++ Standard Library `vector<T>` class. If `v` is a vector of type `NRvector<T>`, then we assume the methods:

|   |   |
|---|---|
| <code>v()</code>                                | Constructor, zero-length vector.  |
| <code>v(Int n)</code>                           | Constructor, vector of length <code>n</code> .  |
| <code>v(Int n, const T &amp;a)</code>           | Constructor, initialize all elements to the value <code>a</code> .  |
| <code>v(Int n, const T *a)</code>               | Constructor, initialize elements to values in a C-style array, <code>a[0], a[1], ...</code>                   |
| <code>v(const NRvector &amp;rhs)</code>         | Copy constructor.   |
| <code>v.size()</code>                           | Returns number of elements in <code>v</code> .  |
| <code>v.resize(Int newn)</code>                 | Resizes <code>v</code> to size <code>newn</code> . We do not assume that contents are preserved.              |
| <code>v.assign(Int newn, const T &amp;a)</code> | Resize <code>v</code> to size <code>newn</code> , and set all elements to the value <code>a</code> .          |
| <code>v[Int i]</code>                           | Element of <code>v</code> by subscript, either an l-value and an r-value.                                     |
| <code>v = rhs</code>                            | Assignment operator. Resizes <code>v</code> if necessary and makes it a copy of the vector <code>rhs</code> . |
| <code>typedef T value_type;</code>              | Makes <code>T</code> available externally (useful in templated functions or classes).                         |

As we will discuss later in more detail, you can use any vector class you like with *Numerical Recipes*, as long as it provides the above basic functionality. For example, a brute force way to use the C++ Standard Library `vector<T>` class instead of `NRvector` is by the preprocessor directive

```
#define NRvector vector
```

(In fact, there is a compiler switch, `_USESTDVECTOR_`, in the file `nr3.h` that will do just this.)

The methods for matrices are closely analogous. If `vv` is a matrix of type `NRmatrix<T>`, then we assume the methods:

|  |  |
|--|--|
| <code>vv()</code>  | Constructor, zero-length vector.   |
| <code>vv(Int n, Int m)</code>  | Constructor, $n \times m$ matrix.  |
| <code>vv(Int n, Int m, const T &amp;a)</code>                              | Constructor, initialize all elements to the value <code>a</code> .   |
| <code>vv(Int n, Int m, const T *a)</code>                                  | Constructor, initialize elements by rows to the values in a C-style array.                                     |
| <code>vv(const NRmatrix &amp;rhs)</code>                                   | Copy constructor.  |
| <code>vv.nrows()</code>  | Returns number of rows <code>n</code> .  |
| <code>vv.ncols()</code>  | Returns number of columns <code>m</code> .   |
| <code>vv.resize(Int newn, Int newm)</code>                                 | Resizes <code>vv</code> to <code>newn × newm</code> . We do not assume that contents are preserved.            |
| <code>vv.assign(Int newn, Int newm,</code><br><code>const t &amp;a)</code> | Resizes <code>vv</code> to <code>newn × newm</code> , and sets all elements to the value <code>a</code> .      |
| <code>vv[Int i]</code>   | Return a pointer to the first element in row <code>i</code> (not often used by itself).                        |
| <code>v[Int i][Int j]</code>   | Element of <code>vv</code> by subscript, either an l-value and an r-value.                                     |
| <code>vv = rhs</code>  | Assignment operator. Resizes <code>vv</code> if necessary and makes it a copy of the matrix <code>rhs</code> . |
| <code>typedef T value_type;</code>   | Makes <code>T</code> available externally.   |

For more precise specifications, see §1.4.3.

There is one additional property that we assume of the vector and matrix classes, namely that all of an object's elements are stored in sequential order. For a vector, this means that its elements can be addressed by pointer arithmetic relative to the first element. For example, if we have

```
VecDoub a(100);
Doub *b = &a[0];
```

then `a[i]` and `b[i]` reference the same element, both as an l-value and as an r-value. This capability is sometimes important for inner-loop efficiency, and it is also useful for interfacing with legacy code that can handle `Doub*` arrays, but not `VecDoub` vectors. Although the original C++ Standard Library did not guarantee this behavior, all known implementations of it do so, and the behavior is now required by an amendment to the standard [2].

For matrices, we analogously assume that storage is by rows within a single sequential block so that, for example,

```
Int n=97, m=103;
MatDoub a(n,m);
Doub *b = &a[0][0];
```

implies that `a[i][j]` and `b[m*i+j]` are equivalent.

A few of our routines need the capability of taking as an argument either a vector or else one row of a matrix. For simplicity, we usually code this using overloading, as, e.g.,

```
void someroutine(Doub *v, Int m) {           Version for a matrix row.
    ...
}
inline void someroutine(VecDoub &v) {          Version for a vector.
    someroutine(&v[0], v.size());
}
```

For a vector `v`, a call looks like `someroutine(v)`, while for row `i` of a matrix `vv` it is `someroutine(&vv[i][0], vv.ncols())`. While the simpler argument `vv[i]` would in fact work in our implementation of `NRmatrix`, it might not work in some other matrix class that guarantees sequential storage but has the return type for a single subscript different from `T*`.

### 1.4.3 Implementations in nr3.h

For reference, here is a complete declaration of `NRvector`.

|   |  |  |
|---|--|--|
| template <class T>                              |  |  |
| class NRvector {                                |  |  |
| private:  |  |  |
| int nn;   | Size of array, indices 0..nn-1.          |  |
| T *v;   | Pointer to data array.                   |  |
| public:   |  |  |
| NRvector();                                     | Default constructor.                     |  |
| explicit NRvector(int n);                       | Construct vector of size n.              |  |
| NRvector(int n, const T &a);                    | Initialize to constant value a.          |  |
| NRvector(int n, const T *a);                    | Initialize to values in C-style array a. |  |
| NRvector(const NRvector &rhs);                  | Copy constructor.                        |  |
| NRvector & operator=(const NRvector &rhs);      | Assignment operator.                     |  |
| typedef T value_type;                           | Make T available.                        |  |
| inline T & operator[](const int i);             | Return element number i.                 |  |
| inline const T & operator[](const int i) const; | const version.                           |  |
| inline int size() const;                        | Return size of vector.                   |  |
| void resize(int newn);                          | Resize, losing contents.                 |  |
| void assign(int newn, const T &a);              | Resize and assign a to every element.    |  |
| ~NRvector();                                    | Destructor.                              |  |
| };  |  |  |

The implementations are straightforward and can be found in the file `nr3.h`. The only issues requiring finesse are the consistent treatment of zero-length vectors and the avoidance of unnecessary resize operations.

A complete declaration of `NRmatrix` is

```
template <class T>
class NRmatrix {
private:
    int nn;
    int mm;
    T **v;
public:
    NRmatrix();
    NRmatrix(int n, int m);
    NRmatrix(int n, int m, const T &a);
    NRmatrix(int n, int m, const T *a);
    NRmatrix(const NRmatrix &rhs);
    NRmatrix & operator=(const NRmatrix &rhs);
    typedef T value_type;
    inline T* operator[](const int i);
    inline const T* operator[](const int i) const;
    inline int nrows() const;
    inline int ncols() const;
    void resize(int newn, int newm);
    void assign(int newn, int newm, const T &a);
    ~NRmatrix();
};

Number of rows and columns. Index
range is 0..nn-1, 0..mm-1.
Storage for data.

Default constructor.
Construct  $n \times m$  matrix.
Initialize to constant value a.
Initialize to values in C-style array a.
Copy constructor.
Assignment operator.
Make T available.
Subscripting: pointer to row i.
const version.

Return number of rows.
Return number of columns.
Resize, losing contents.
Resize and assign a to every element.
Destructor.
```

A couple of implementation details in `NRmatrix` are worth commenting on. The private variable `**v` points not to the data but rather to an array of pointers to the data rows. Memory allocation of this array is separate from the allocation of space for the actual data. The data space is allocated as a single block, not separately for each row. For matrices of zero size, we have to account for the separate possibilities that there are zero rows, or that there are a finite number of rows, but each with zero columns. So, for example, one of the constructors looks like this:

```
template <class T>
NRmatrix<T>::NRmatrix(int n, int m) : nn(n), mm(m), v(n>0 ? new T*[n] : NULL)
{
    int i, nel=m*n;
    if (v) v[0] = nel>0 ? new T[nel] : NULL;
    for (i=1;i<n;i++) v[i] = v[i-1] + m;
}
```

Finally, it matters *a lot* whether your compiler honors the `inline` directives in `NRvector` and `NRmatrix` above. If it doesn't, then you may be doing full function calls, saving and restoring context within the processor, every time you address a vector or matrix element. This is tantamount to making C++ useless for most numerical computing! Luckily, as we write, the most commonly used compilers are all "honorable" in this respect.

#### CITED REFERENCES AND FURTHER READING:

Josuttis, N.M. 1999, *The C++ Standard Library: A Tutorial and Reference* (Boston: Addison-Wesley).[1]

International Standardization Organization 2003, *Technical Corrigendum ISO 14882:2003*.[2]

## 1.5 Some Further Conventions and Capabilities

We collect in this section some further explanation of C++ language capabilities and how we use them in this book.

### 1.5.1 Error and Exception Handling

We already mentioned that we code error conditions with simple `throw` statements, like this

```
throw("error foo in routine bah");
```

If you are programming in an environment that has a defined set of error classes, and you want to use them, then you'll need to change these lines in the routines that you use. Alternatively, without any additional machinery, you can choose between a couple of different, useful behaviors just by making small changes in `nr3.h`.

By default, `nr3.h` redefines `throw()` by a preprocessor macro,

```
#define throw(message) \
    {printf("ERROR: %s\n in file %s at line %d\n",
        message, __FILE__, __LINE__); \
    exit(1);}
```

This uses standard ANSI C features, also present in C++, to print the source code file name and line number at which the error occurs. It is inelegant, but perfectly functional.

Somewhat more functional, and definitely more elegant, is to set `nr3.h`'s compiler switch `_USENRERRORCLASS_`, which instead substitutes the following code:

```
struct NRerror {
    char *message;
    char *file;
    int line;
    NRerror(char *m, char *f, int l) : message(m), file(f), line(l) {}
};

void NRcatch(NRerror err) {
    printf("ERROR: %s\n      in file %s at line %d\n",
        err.message, err.file, err.line);
    exit(1);
}

#define throw(message) throw(NRerror(message,__FILE__,__LINE__));
```

Now you have a (rudimentary) error class, `NRerror`, available. You use it by putting a `try...catch` control structure at any desired point (or points) in your code, for example (§2.9),

```
...
try {
    Cholesky achol(a);
}
catch (NRerror err) {
    NRcatch(err);           Executed if Cholesky throws an exception.
}
```

As shown, the use of the `NRcatch` function above simply mimics the behavior of the previous macro in the global context. But you don't have to use `NRcatch` at all: You

can substitute any code that you want for the body of the `catch` statement. If you want to distinguish between different kinds of exceptions that may be thrown, you can use the information returned in `err`. We'll let you figure this out yourself. And of course you are welcome to add more complicated error classes to your own copy of `nr3.h`.

### 1.5.2 Const Correctness

Few topics in discussions about C++ evoke more heat than questions about the keyword `const`. We are firm believers in using `const` wherever possible, to achieve what is called “`const` correctness.” Many coding errors are automatically trapped by the compiler if you have qualified identifiers that should not change with `const` when they are declared. Also, using `const` makes your code much more readable: When you see `const` in front of an argument to a function, you know immediately that the function will not modify the object. Conversely, if `const` is absent, you should be able to count on the object being changed somewhere.

We are such strong `const` believers that we insert `const` even where it is theoretically redundant: If an argument is passed *by value* to a function, then the function makes a copy of it. Even if this copy is modified by the function, the original value is unchanged after the function exits. While this allows you to change, with impunity, the values of arguments that have been passed by value, this usage is error-prone and hard to read. If your intention in passing something by value is that it is an input variable only, then make it clear. So we declare a function  $f(x)$  as, for example,

```
Doub f(const Doub x);
```

If in the function you want to use a local variable that is initialized to `x` but then gets changed, define a new quantity — don't use `x`. If you put `const` in the declaration, the compiler will not let you get this wrong.

Using `const` in your function arguments makes your function more general: Calling a function that expects a `const` argument with a non-`const` variable involves a “trivial” conversion. But trying to pass a `const` quantity to a non-`const` argument is an error.

A final reason for using `const` is that it allows certain user-defined conversions to be made. As discussed in [1], this can be useful if you want to use *Numerical Recipes* routines with another matrix/vector class library.

We now need to elaborate on what exactly `const` does for a nonsimple type such as a class that is an argument of a function. Basically, it guarantees that the object is not modified by the function. In other words, the object's data members are unchanged. But if a data member is a *pointer* to some data, and the data itself is not a member variable, then *the data can be changed* even though the pointer cannot be.

Let's look at the implications of this for a function `f` that takes an `NRvector<Doub>` argument `a`. To avoid unnecessary copying, we always pass vectors and matrices by reference. Consider the difference between declaring the argument of a function with and without `const`:

```
void f(NRvector<Doub> &a)      versus      void f(const NRvector<Doub> &a)
```

The `const` version promises that `f` does not modify the data members of `a`. But a statement like

```
a[i] = 4.;
```

inside the function definition is in principle perfectly OK — you are modifying the data pointed to, not the pointer itself.

“Isn’t there some way to protect the data?” you may ask. Yes, there is: You can declare the *return type* of the subscript operator, `operator[]`, to be `const`. This is why there are two versions of `operator[]` in the `NRvector` class,

```
T & operator[](const int i);
const T & operator[](const int i) const;
```

The first form returns a reference to a modifiable vector element, while the second returns a nonmodifiable vector element (because the return type has a `const` in front).

But how does the compiler know which version to invoke when you just write `a[i]`? That is specified by the *trailing* word `const` in the second version. It refers not to the returned element, nor to the argument `i`, but to the object whose `operator[]` is being invoked, in our example the vector `a`. Taken together, the two versions say this to the compiler: “If the vector `a` is `const`, then transfer that `const`’ness to the returned element `a[i]`. If it isn’t, then don’t.”

The remaining question is thus how the compiler determines whether `a` is `const`. In our example, where `a` is a function argument, it is trivial: The argument is either declared as `const` or else it isn’t. In other contexts, `a` might be `const` because you originally declared it as such (and initialized it via constructor arguments), or because it is a `const` reference data member in some other object, or for some other, more arcane, reason.

As you can see, getting `const` to protect the data is a little complicated. Judging from the large number of matrix/vector libraries that follow this scheme, many people feel that the payoff is worthwhile. We urge you *always* to declare as `const` those objects and variables that are not intended to be modified. You do this both at the time an object is actually created and in the arguments of function declarations and definitions. You won’t regret making a habit of `const` correctness.

In §1.4 we defined vector and matrix type names with trailing `_I` labels, for example, `VecDoub_I` and `MatInt_I`. The `_I`, which stands for “input to a function,” means that the type is declared as `const`. (This is already done in the `typedef` statement; you don’t have to repeat it.) The corresponding labels `_O` and `_IO` are to remind you when arguments are not just non-`const`, but will actually be modified by the function in whose argument list they appear.

Having rightly put all this emphasis on `const` correctness, duty compels us also to recognize the existence of an alternative philosophy, which is to stick with the more rudimentary view “`const` protects the container, not the contents.” In this case you would want only *one* form of `operator[]`, namely

```
T & operator[](const int i) const;
```

It would be invoked whether your vector was passed by `const` reference or not. In both cases element `i` is returned as potentially modifiable. While we are opposed to this philosophically, it turns out that it does make possible a tricky kind of automatic type conversion that allows you to use your favorite vector and matrix classes instead of `NRvector` and `NRmatrix`, even if your classes use a syntax completely different from what we have assumed. For information on this very specialized application, see [1].

### 1.5.3 Abstract Base Class (ABC), or Template?

There is sometimes more than one good way to achieve some end in C++. Heck, let's be honest: There is *always* more than one way. Sometimes the differences amount to small tweaks, but at other times they embody very different views about the language. When we make one such choice, and you prefer another, you are going to be quite annoyed with us. Our defense against this is to avoid foolish consistencies,\* and to illustrate as many viewpoints as possible.

A good example is the question of when to use an abstract base class (ABC) versus a template, when their capabilities overlap. Suppose we have a function `func` that can do its (useful) thing on, or using, several different types of objects, call them `ObjA`, `ObjB`, and `ObjC`. Moreover, `func` doesn't need to know much about the object it interacts with, only that it has some method `tellme`.

We could implement this setup as an abstract base class:

```
struct ObjABC {
    virtual void tellme() = 0;
};

struct ObjA : ObjABC {           Abstract Base Class for objects with tellme.
    ...
    void tellme() {...}
};
struct ObjB : ObjABC {           Derived class.
    ...
    void tellme() {...}
};
struct ObjC : ObjABC {           Derived class.
    ...
    void tellme() {...}
};

void func(ObjABC &x) {
    ...
    x.tellme();                 References the appropriate tellme.
}
```

On the other hand, using a template, we can write code for `func` without ever seeing (or even knowing the names of) the objects for which it is intended:

```
template<class T>
void func(T &x) {
    ...
    x.tellme();
}
```

That certainly seems easier! Is it better?

Maybe. A disadvantage of templates is that the template must be available to the compiler every time it encounters a call to `func`. This is because it actually compiles a different version of `func` for every different type of argument `T` that it encounters. Unless your code is so large that it cannot easily be compiled as a single unit, however, this is not much of a disadvantage. On the other side, favoring templates, is the fact that virtual functions incur a small run-time penalty when they are called. But this is rarely significant.

The deciding factors in this example relate to software engineering, not performance, and are hidden in the lines with ellipses (...). We haven't really told

---

\*“A foolish consistency is the hobgoblin of little minds.” —Emerson

you how closely related `ObjA`, `ObjB`, and `ObjC` are. If they are close, then the ABC approach offers possibilities for putting more than just `tellme` into the base class. Putting things into the base class, whether data or pure virtual methods, lets the compiler enforce consistency across the derived classes. If you later write another derived object `ObjD`, its consistency will also be enforced. For example, the compiler will require you to implement a method in every derived class corresponding to every pure virtual method in the base class.

By contrast, in the template approach, the only enforced consistency will be that the method `tellme` exists, and this will only be enforced at the point in the code where `func` is actually called with an `ObjD` argument (if such a point exists), not at the point where `ObjD` is defined. Consistency checking in the template approach is thus somewhat more haphazard.

Laid-back programmers will opt for templates. Up-tight programmers will opt for ABCs. We opt for... both, on different occasions. There can also be other reasons, having to do with subtle features of class derivation or of templates, for choosing one approach over the other. We will point these out as we encounter them in later chapters. For example, in Chapter 17 we define an abstract base class called `StepperBase` for the various “stepper” routines for solving ODEs. The derived classes implement particular stepping algorithms, and they are templated so they can accept function or functor arguments (see §1.3.3).

### 1.5.4 *NaN and Floating Point Exceptions*

We mentioned in §1.1.1 that the IEEE floating-point standard includes a representation for `NaN`, meaning “not a number.” `NaN` is distinct from positive and negative infinity, as well as from every representable number. It can be both a blessing and a curse.

The blessing is that it can be useful to have a value that can be used with meanings like “don’t process me” or “missing data” or “not yet initialized.” To use `NaN` in this fashion, you need to be able to *set* variables to it, and you need to be able to *test* for its having been set.

Setting is easy. The “approved” method is to use `numeric_limits`. In `nr3.h` the line

```
static const Doub NaN = numeric_limits<Doub>::quiet_NaN();
```

defines a global value `NaN`, so that you can write things like

```
x = NaN;
```

at will. If you ever encounter a compiler that doesn’t do this right (it’s a pretty obscure corner of the standard library!), then try either

```
UInt proto_nan[2]=0xffffffff, 0x7fffffff;
double NaN = *( double* )proto_nan;
```

(which assumes little-endian behavior; cf. §1.1.1) or the self-explanatory

```
Doub NaN = sqrt(-1.);
```

which may, however, throw an immediate exception (see below) and thus not work for this purpose. But, one way or another, you can generally figure out how to get a `NaN` constant into your environment.

Testing also requires a bit of (one-time) experimentation: According to the IEEE standard, `NaN` is guaranteed to be the only value that is not equal to itself!

So, the “approved” method of testing whether Doub value  $x$  has been set to NaN is

```
if (x != x) {...}           It's a NaN!
```

(or test for equality to determine that it’s not a NaN). Unfortunately, at time of writing, some otherwise perfectly good compilers don’t do this right. Instead, they provide a macro `isnan()` that returns `true` if the argument is NaN, otherwise `false`. (Check carefully whether the required `#include` is `math.h` or `float.h` — it varies.)

What, then, is the *curse* of NaN? It is that some compilers, notably Microsoft, have poorly thought-out default behaviors in distinguishing *quiet NaNs* from *signalling NaNs*. Both kinds of NaNs are defined in the IEEE floating-point standard. Quiet NaNs are supposed to be for uses like those above: You can set them, test them, and propagate them by assignment, or even through other floating operations. In such uses they are not supposed to signal an exception that causes your program to abort. Signalling NaNs, on the other hand, are, as the name implies, supposed to signal exceptions. Signalling NaNs should be generated by invalid operations, such as the square root or logarithm of a negative number, or `pow(0., 0.)`.

If all NaNs are treated as signalling exceptions, then you can’t make use of them as we have suggested above. That’s annoying, but OK. On the other hand, if all NaNs are treated as quiet (the Microsoft default at time of writing), then you will run long calculations only to find that all the results are NaN — and you have no way of locating the invalid operation that triggered the propagating cascade of (quiet) NaNs. That’s *not* OK. It makes debugging a nightmare. (You can get the same disease if other floating-point exceptions propagate, for example overflow or division-by-zero.)

Tricks for specific compilers are not within our normal scope. But this one is so essential that we make it an “exception”: If you are living on planet Microsoft, then the lines of code,

```
int cw = _controlfp(0,0);
cw &= ~(EM_INVALID | EM_OVERFLOW | EM_ZERODIVIDE );
_controlfp(cw,MCW_EM);
```

at the beginning of your program will turn NaNs from invalid operations, overflows, and divides-by-zero into signalling NaNs, and leave all the other NaNs quiet. There is a compiler switch, `_TURNONFPES_` in `nr3.h` that will do this for you automatically. (Further options are `EM_UNDERFLOW`, `EM_INEXACT`, and `EM_DENORMAL`, but we think these are best left quiet.)

### 1.5.5 Miscellany

- Bounds checking in vectors and matrices, that is, verifying that subscripts are in range, is expensive. It can easily double or triple the access time to subscripted elements. In their default configuration, the `NRvector` and `NRmatrix` classes never do bounds checking. However, `nr3.h` has a compiler switch, `_CHECKBOUNDS_`, that turns bounds checking on. This is implemented by pre-processor directives for conditional compilation so there is no performance penalty when you leave it turned off. This is ugly, but effective.

The `vector<>` class in the C++ Standard Library takes a different tack. If you access a vector element by the syntax `v[i]`, there is no bounds checking. If you instead use the `at()` method, as `v.at(i)`, then bounds checking is performed. The obvious weakness of this approach is that you can’t easily change a lengthy program from one method to the other, as you might want to

do when debugging.

- The importance to performance of avoiding unnecessary copying of large objects, such as vectors and matrices, cannot be overemphasized. As already mentioned, they should always be passed by reference in function arguments. But you also need to be careful about, or avoid completely, the use of functions whose return type is a large object. This is true even if the return type is a reference (which is a tricky business anyway). Our experience is that compilers often create temporary objects, using the copy constructor, when the need to do so is obscure or nonexistent. That is why we so frequently write `void` functions that have an argument of type (e.g.) `MatDoub_0` for returning the “answer.” (When we do use vector or matrix return types, our excuse is either that the code is pedagogical, or that the overhead is negligible compared to some big calculation that has just been done.)

You can check up on your compiler by instrumenting the vector and matrix classes: Add a static integer variable to the class definition, increment it within the copy constructor and assignment operator, and look at its value before and after operations that (you think) should not require any copies. You might be surprised.

- There are only two routines in *Numerical Recipes* that use three-dimensional arrays, `r1ft3` in §12.6, and `solvde` in §18.3. The file `nr3.h` includes a rudimentary class for three-dimensional arrays, mainly to service these two routines. In many applications, a better way to proceed is to declare a vector of matrices, for example,

```
vector<MatDoub> threedee(17);
for (Int i=0;i<17;i++) threedee[i].resize(19,21);
```

which creates, in effect, a three-dimensional array of size  $17 \times 19 \times 21$ . You can address individual components as `threedee[i][j][k]`.

- “Why no namespace?” Industrial-strength programmers will notice that, unlike the second edition, this third edition of *Numerical Recipes* does not shield its function and class names by a `NR::` namespace. Therefore, if you are so bold as to `#include` every single file in the book, you are dumping on the order of 500 names into the global namespace, definitely a bad idea!

The explanation, quite simply, is that the vast majority of our users are not industrial-strength programmers, and most found the `NR::` namespace annoying and confusing. As we emphasized, strongly, in §1.0.1, NR is not a program library. If you want to create your own personal namespace for NR, please go ahead.

- In the distant past, *Numerical Recipes* included provisions for unit- or one-based, instead of zero-based, array indices. The last such version was published in 1992. Zero-based arrays have become so universally accepted that we no longer support any other option.

#### CITED REFERENCES AND FURTHER READING:

Numerical Recipes Software 2007, “Using Other Vector and Matrix Libraries,” *Numerical Recipes Webnote No. 1*, at [http://www.nr.com/webnotes?1\[1\]](http://www.nr.com/webnotes?1[1])

# Solution of Linear Algebraic Equations

---

## 2.0 Introduction

The most basic task in linear algebra, and perhaps in all of scientific computing, is to solve for the unknowns in a set of linear algebraic equations. In general, a set of linear algebraic equations looks like this:

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \cdots + a_{0,N-1}x_{N-1} &= b_0 \\ a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \cdots + a_{1,N-1}x_{N-1} &= b_1 \\ a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + \cdots + a_{2,N-1}x_{N-1} &= b_2 \\ &\vdots &&\vdots \\ a_{M-1,0}x_0 + a_{M-1,1}x_1 + \cdots + a_{M-1,N-1}x_{N-1} &= b_{M-1} \end{aligned} \tag{2.0.1}$$

Here the  $N$  unknowns  $x_j$ ,  $j = 0, 1, \dots, N - 1$  are related by  $M$  equations. The coefficients  $a_{ij}$  with  $i = 0, 1, \dots, M - 1$  and  $j = 0, 1, \dots, N - 1$  are known numbers, as are the *right-hand side* quantities  $b_i$ ,  $i = 0, 1, \dots, M - 1$ .

If  $N = M$ , then there are as many equations as unknowns, and there is a good chance of solving for a unique solution set of  $x_j$ 's. Otherwise, if  $N \neq M$ , things are even more interesting; we'll have more to say about this below.

If we write the coefficients  $a_{ij}$  as a matrix, and the right-hand sides  $b_i$  as a column vector,

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,N-1} \\ a_{10} & a_{11} & \cdots & a_{1,N-1} \\ \vdots & & & \\ a_{M-1,0} & a_{M-1,1} & \cdots & a_{M-1,N-1} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{M-1} \end{bmatrix} \tag{2.0.2}$$

then equation (2.0.1) can be written in matrix form as

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.0.3}$$

Here, and throughout the book, we use a raised dot to denote matrix multiplication, or the multiplication of a matrix and a vector, or the dot product of two vectors.

This usage is nonstandard, but we think it adds clarity: the dot is, in all of these cases, a *contraction* operator that represents the sum over a pair of indices, for example

$$\begin{aligned}\mathbf{C} = \mathbf{A} \cdot \mathbf{B} &\iff c_{ik} = \sum_j a_{ij} b_{jk} \\ \mathbf{b} = \mathbf{A} \cdot \mathbf{x} &\iff b_i = \sum_j a_{ij} x_j \\ \mathbf{d} = \mathbf{x} \cdot \mathbf{A} &\iff d_j = \sum_i x_i a_{ij} \\ q = \mathbf{x} \cdot \mathbf{y} &\iff q = \sum_i x_i y_i\end{aligned}\tag{2.0.4}$$

In matrices, by convention, the first index on an element  $a_{ij}$  denotes its row and the second index its column. For most purposes you don't need to know how a matrix is stored in a computer's physical memory; you just reference matrix elements by their two-dimensional addresses, e.g.,  $a_{34} = \mathbf{a}[3][4]$ . This C++ notation can in fact hide a variety of subtle and versatile physical storage schemes, see §1.4 and §1.5.

### 2.0.1 Nonsingular versus Singular Sets of Equations

You might be wondering why, above, and for the case  $M = N$ , we credited only a “good” chance of solving for the unknowns. Analytically, there can fail to be a solution (or a unique solution) if one or more of the  $M$  equations is a linear combination of the others, a condition called *row degeneracy*, or if all equations contain certain variables only in exactly the same linear combination, called *column degeneracy*. It turns out that, for square matrices, row degeneracy implies column degeneracy, and vice versa. A set of equations that is degenerate is called *singular*. We will consider singular matrices in some detail in §2.6.

Numerically, at least two additional things prevent us from getting a good solution:

- While not exact linear combinations of each other, some of the equations may be so close to linearly dependent that roundoff errors in the machine render them linearly dependent at some stage in the solution process. In this case your numerical procedure will fail, and it can tell you that it has failed.
- Accumulated roundoff errors in the solution process can swamp the true solution. This problem particularly emerges if  $N$  is too large. The numerical procedure does not fail algorithmically. However, it returns a set of  $x$ 's that are wrong, as can be discovered by direct substitution back into the original equations. The closer a set of equations is to being singular, the more likely this is to happen, since increasingly close cancellations will occur during the solution. In fact, the preceding item can be viewed as the special case in which the loss of significance is unfortunately total.

Much of the sophistication of well-written “linear equation-solving packages” is devoted to the detection and/or correction of these two pathologies. It is difficult to give any firm guidelines for when such sophistication is needed, since there is no such thing as a “typical” linear problem. But here is a rough idea: Linear sets with  $N$  no larger than 20 or 50 are routine if they are not close to singular; they rarely

require more than the most straightforward methods, even in only single (that is, `float`) precision. With double precision, this number can readily be extended to  $N$  as large as perhaps 1000, after which point the limiting factor anyway soon becomes machine time, not accuracy.

Even larger linear sets,  $N$  in the thousands or millions, can be solved when the coefficients are sparse (that is, mostly zero), by methods that take advantage of the sparseness. We discuss this further in §2.7.

Unfortunately, one seems just as often to encounter linear problems that, by their underlying nature, are close to singular. In this case, you *might* need to resort to sophisticated methods even for the case of  $N = 10$  (though rarely for  $N = 5$ ). Singular value decomposition (§2.6) is a technique that can sometimes turn singular problems into nonsingular ones, in which case additional sophistication becomes unnecessary.

## 2.0.2 Tasks of Computational Linear Algebra

There is much more to linear algebra than just solving a single set of equations with a single right-hand side. Here, we list the major topics treated in this chapter. (Chapter 11 continues the subject with discussion of eigenvalue/eigenvector problems.)

When  $M = N$ :

- Solution of the matrix equation  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$  for an unknown vector  $\mathbf{x}$  (§2.1 – §2.10).
- Solution of more than one matrix equation  $\mathbf{A} \cdot \mathbf{x}_j = \mathbf{b}_j$ , for a set of vectors  $\mathbf{x}_j$ ,  $j = 0, 1, \dots$ , each corresponding to a different, known right-hand side vector  $\mathbf{b}_j$ . In this task the key simplification is that the matrix  $\mathbf{A}$  is held constant, while the right-hand sides, the  $\mathbf{b}$ 's, are changed (§2.1 – §2.10).
- Calculation of the matrix  $\mathbf{A}^{-1}$  that is the matrix inverse of a square matrix  $\mathbf{A}$ , i.e.,  $\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{1}$ , where  $\mathbf{1}$  is the identity matrix (all zeros except for ones on the diagonal). This task is equivalent, for an  $N \times N$  matrix  $\mathbf{A}$ , to the previous task with  $N$  different  $\mathbf{b}_j$ 's ( $j = 0, 1, \dots, N - 1$ ), namely the unit vectors ( $\mathbf{b}_j$  = all zero elements except for 1 in the  $j$ th component). The corresponding  $\mathbf{x}$ 's are then the columns of the matrix inverse of  $\mathbf{A}$  (§2.1 and §2.3).
- Calculation of the determinant of a square matrix  $\mathbf{A}$  (§2.3).

If  $M < N$ , or if  $M = N$  but the equations are degenerate, then there are effectively fewer equations than unknowns. In this case there can be either no solution, or else more than one solution vector  $\mathbf{x}$ . In the latter event, the solution space consists of a particular solution  $\mathbf{x}_p$  added to any linear combination of (typically)  $N - M$  vectors (which are said to be in the nullspace of the matrix  $\mathbf{A}$ ). The task of finding the solution space of  $\mathbf{A}$  involves

- Singular value decomposition of a matrix  $\mathbf{A}$  (§2.6).

If there are more equations than unknowns,  $M > N$ , there is in general no solution vector  $\mathbf{x}$  to equation (2.0.1), and the set of equations is said to be *overdetermined*. It happens frequently, however, that the best “compromise” solution is sought, the one that comes closest to satisfying all equations simultaneously. If closeness is defined in the least-squares sense, i.e., that the sum of the squares of the differences between the left- and right-hand sides of equation (2.0.1) be mini-

mized, then the overdetermined linear problem reduces to a (usually) solvable linear problem, called the

- Linear least-squares problem.

The reduced set of equations to be solved can be written as the  $N \times N$  set of equations

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{x} = (\mathbf{A}^T \cdot \mathbf{b}) \quad (2.0.5)$$

where  $\mathbf{A}^T$  denotes the transpose of the matrix  $\mathbf{A}$ . Equations (2.0.5) are called the *normal equations* of the linear least-squares problem. There is a close connection between singular value decomposition and the linear least-squares problem, and the latter is also discussed in §2.6. You should be warned that direct solution of the normal equations (2.0.5) is not generally the best way to find least-squares solutions.

Some other topics in this chapter include

- Iterative improvement of a solution (§2.5)
- Various special forms: symmetric positive-definite (§2.9), tridiagonal (§2.4), band-diagonal (§2.4), Toeplitz (§2.8), Vandermonde (§2.8), sparse (§2.7)
- Strassen's "fast matrix inversion" (§2.11).

### 2.0.3 Software for Linear Algebra

Going beyond what we can include in this book, several good software packages for linear algebra are available, though not always in C++. LAPACK, a successor to the venerable LINPACK, was developed at Argonne National Laboratories and deserves particular mention because it is published, documented, and available for free use. ScaLAPACK is a version available for parallel architectures. Packages available commercially include those in the IMSL and NAG libraries.

Sophisticated packages are designed with very large linear systems in mind. They therefore go to great effort to minimize not only the number of operations, but also the required storage. Routines for the various tasks are usually provided in several versions, corresponding to several possible simplifications in the form of the input coefficient matrix: symmetric, triangular, banded, positive-definite, etc. If you have a large matrix in one of these forms, you should certainly take advantage of the increased efficiency provided by these different routines, and not just use the form provided for general matrices.

There is also a great watershed dividing routines that are *direct* (i.e., execute in a predictable number of operations) from routines that are *iterative* (i.e., attempt to converge to the desired answer in however many steps are necessary). Iterative methods become preferable when the battle against loss of significance is in danger of being lost, either due to large  $N$  or because the problem is close to singular. We will treat iterative methods only incompletely in this book, in §2.7 and in Chapters 19 and 20. These methods are important but mostly beyond our scope. We will, however, discuss in detail a technique that is on the borderline between direct and iterative methods, namely the iterative improvement of a solution that has been obtained by direct methods (§2.5).

#### CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press).

- Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley).
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 4.
- Ueberhuber, C.W. 1997, *Numerical Computation: Methods, Software, and Analysis*, 2 vols. (Berlin: Springer), Chapter 13.
- Coleman, T.F., and Van Loan, C. 1988, *Handbook for Matrix Computations* (Philadelphia: S.I.A.M.).
- Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall).
- Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer).
- Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).
- Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 2.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), Chapter 9.

## 2.1 Gauss-Jordan Elimination

*Gauss-Jordan elimination* is probably the way you learned to solve linear equations in high school. (You may have learned it as “Gaussian elimination,” but, strictly speaking, that term refers to the somewhat different technique discussed in §2.2.) The basic idea is to add or subtract linear combinations of the given equations until each equation contains only one of the unknowns, thus giving an immediate solution. You might also have learned to use the same technique for calculating the inverse of a matrix.

For solving sets of linear equations, Gauss-Jordan elimination produces *both* the solution of the equations for one or more right-hand side vectors  $\mathbf{b}$ , and also the matrix inverse  $\mathbf{A}^{-1}$ . However, its principal deficiencies are (i) that it requires all the right-hand sides to be stored and manipulated at the same time, and (ii) that when the inverse matrix is *not* desired, Gauss-Jordan is three times slower than the best alternative technique for solving a single linear set (§2.3). The method’s principal strength is that it is as stable as any other direct method, perhaps even a bit more stable when full pivoting is used (see §2.1.2).

For inverting a matrix, Gauss-Jordan elimination is about as efficient as any other direct method. We know of no reason not to use it in this application if you are sure that the matrix inverse is what you really want.

You might wonder about deficiency (i) above: If we are getting the matrix inverse anyway, can’t we later let it multiply a new right-hand side to get an additional solution? This does work, but it gives an answer that is very susceptible to roundoff error and not nearly as good as if the new vector had been included with the set of right-hand side vectors in the first instance.

Thus, Gauss-Jordan elimination should not be your method of first choice for solving linear equations. The decomposition methods in §2.3 are better. Why do we discuss Gauss-Jordan at all? Because it is straightforward, solid as a rock, and a good place for us to introduce the important concept of *pivoting* which will also

be important for the methods described later. The actual sequence of operations performed in Gauss-Jordan elimination is very closely related to that performed by the routines in the next two sections.

### 2.1.1 Elimination on Column-Augmented Matrices

For clarity, and to avoid writing endless ellipses ( $\cdots$ ) we will write out equations only for the case of four equations and four unknowns, and with three different right-hand side vectors that are known in advance. You can write bigger matrices and extend the equations to the case of  $N \times N$  matrices, with  $M$  sets of right-hand side vectors, in completely analogous fashion. The routine implemented below in §2.1.2 is, of course, general.

Consider the linear matrix equation

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \left[ \begin{pmatrix} x_{00} \\ x_{10} \\ x_{20} \\ x_{30} \end{pmatrix} \sqcup \begin{pmatrix} x_{01} \\ x_{11} \\ x_{21} \\ x_{31} \end{pmatrix} \sqcup \begin{pmatrix} x_{02} \\ x_{12} \\ x_{22} \\ x_{32} \end{pmatrix} \sqcup \begin{pmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{pmatrix} \right] = \left[ \begin{pmatrix} b_{00} \\ b_{10} \\ b_{20} \\ b_{30} \end{pmatrix} \sqcup \begin{pmatrix} b_{01} \\ b_{11} \\ b_{21} \\ b_{31} \end{pmatrix} \sqcup \begin{pmatrix} b_{02} \\ b_{12} \\ b_{22} \\ b_{32} \end{pmatrix} \sqcup \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right] \quad (2.1.1)$$

Here the raised dot ( $\cdot$ ) signifies matrix multiplication, while the operator  $\sqcup$  just signifies column augmentation, that is, removing the abutting parentheses and making a wider matrix out of the operands of the  $\sqcup$  operator.

It should not take you long to write out equation (2.1.1) and to see that it simply states that  $x_{ij}$  is the  $i$ th component ( $i = 0, 1, 2, 3$ ) of the vector solution of the  $j$ th right-hand side ( $j = 0, 1, 2$ ), the one whose coefficients are  $b_{ij}$ ,  $i = 0, 1, 2, 3$ ; and that the matrix of unknown coefficients  $y_{ij}$  is the inverse matrix of  $a_{ij}$ . In other words, the matrix solution of

$$[\mathbf{A}] \cdot [\mathbf{x}_0 \sqcup \mathbf{x}_1 \sqcup \mathbf{x}_2 \sqcup \mathbf{Y}] = [\mathbf{b}_0 \sqcup \mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \mathbf{1}] \quad (2.1.2)$$

where  $\mathbf{A}$  and  $\mathbf{Y}$  are square matrices, the  $\mathbf{b}_i$ 's and  $\mathbf{x}_i$ 's are column vectors, and  $\mathbf{1}$  is the identity matrix, simultaneously solves the linear sets

$$\mathbf{A} \cdot \mathbf{x}_0 = \mathbf{b}_0 \quad \mathbf{A} \cdot \mathbf{x}_1 = \mathbf{b}_1 \quad \mathbf{A} \cdot \mathbf{x}_2 = \mathbf{b}_2 \quad (2.1.3)$$

and

$$\mathbf{A} \cdot \mathbf{Y} = \mathbf{1} \quad (2.1.4)$$

Now it is also elementary to verify the following facts about (2.1.1):

- Interchanging any two rows of  $\mathbf{A}$  and the corresponding rows of the  $\mathbf{b}$ 's and of  $\mathbf{1}$  does not change (or scramble in any way) the solution  $\mathbf{x}$ 's and  $\mathbf{Y}$ . Rather, it just corresponds to writing the same set of linear equations in a different order.
- Likewise, the solution set is unchanged and in no way scrambled if we replace any row in  $\mathbf{A}$  by a linear combination of itself and any other row, as long as we do the same linear combination of the rows of the  $\mathbf{b}$ 's and  $\mathbf{1}$  (which then is no longer the identity matrix, of course).

- Interchanging any two *columns* of  $\mathbf{A}$  gives the same solution set only if we simultaneously interchange corresponding *rows* of the  $\mathbf{x}$ 's and of  $\mathbf{Y}$ . In other words, this interchange scrambles the order of the rows in the solution. If we do this, we will need to unscramble the solution by restoring the rows to their original order.

Gauss-Jordan elimination uses one or more of the above operations to reduce the matrix  $\mathbf{A}$  to the identity matrix. When this is accomplished, the right-hand side becomes the solution set, as one sees instantly from (2.1.2).

## 2.1.2 Pivoting

In “Gauss-Jordan elimination with no pivoting,” only the second operation in the above list is used. The zeroth row is divided by the element  $a_{00}$  (this being a trivial linear combination of the zeroth row with any other row — zero coefficient for the other row). Then the right amount of the zeroth row is subtracted from each other row to make all the remaining  $a_{i0}$ 's zero. The zeroth column of  $\mathbf{A}$  now agrees with the identity matrix. We move to column 1 and divide row 1 by  $a_{11}$ , then subtract the right amount of row 1 from rows 0, 2, and 3, so as to make their entries in column 1 zero. Column 1 is now reduced to the identity form. And so on for columns 2 and 3. As we do these operations to  $\mathbf{A}$ , we of course also do the corresponding operations to the  $\mathbf{b}$ 's and to  $\mathbf{1}$  (which by now no longer resembles the identity matrix in any way!).

Obviously we will run into trouble if we ever encounter a zero element on the (then current) diagonal when we are going to divide by the diagonal element. (The element that we divide by, incidentally, is called the *pivot element* or *pivot*.) Not so obvious, but true, is the fact that Gauss-Jordan elimination with no pivoting (no use of the first or third procedures in the above list) is numerically unstable in the presence of any roundoff error, even when a zero pivot is not encountered. You must *never* do Gauss-Jordan elimination (or Gaussian elimination; see below) without pivoting!

So what is this magic pivoting? Nothing more than interchanging rows (*partial pivoting*) or rows and columns (*full pivoting*), so as to put a particularly desirable element in the diagonal position from which the pivot is about to be selected. Since we don't want to mess up the part of the identity matrix that we have already built up, we can choose among elements that are both (i) on rows below (or on) the one that is about to be normalized, and also (ii) on columns to the right (or on) the column we are about to eliminate. Partial pivoting is easier than full pivoting, because we don't have to keep track of the permutation of the solution vector. Partial pivoting makes available as pivots only the elements already in the correct column. It turns out that partial pivoting is “almost” as good as full pivoting, in a sense that can be made mathematically precise, but which need not concern us here (for discussion and references, see [1]). To show you both variants, we do full pivoting in the routine in this section and partial pivoting in §2.3.

We have to state how to recognize a particularly desirable pivot when we see one. The answer to this is not completely known theoretically. It is known, both theoretically and in practice, that simply picking the largest (in magnitude) available element as the pivot is a very good choice. A curiosity of this procedure, however, is that the choice of pivot will depend on the original scaling of the equations. If we take the third linear equation in our original set and multiply it by a factor of a million, it is almost guaranteed that it will contribute the first pivot; yet the underlying solution

of the equations is not changed by this multiplication! One therefore sometimes sees routines which choose as pivot that element which *would* have been largest if the original equations had all been scaled to have their largest coefficient normalized to unity. This is called *implicit pivoting*. There is some extra bookkeeping to keep track of the scale factors by which the rows would have been multiplied. (The routines in §2.3 include implicit pivoting, but the routine in this section does not.)

Finally, let us consider the storage requirements of the method. With a little reflection you will see that at every stage of the algorithm, *either* an element of  $\mathbf{A}$  is predictably a one or zero (if it is already in a part of the matrix that has been reduced to identity form) *or else* the exactly corresponding element of the matrix that started as 1 is predictably a one or zero (if its mate in  $\mathbf{A}$  has not been reduced to the identity form). Therefore the matrix 1 does not have to exist as separate storage: The matrix inverse of  $\mathbf{A}$  is gradually built up in  $\mathbf{A}$  as the original  $\mathbf{A}$  is destroyed. Likewise, the solution vectors  $\mathbf{x}$  can gradually replace the right-hand side vectors  $\mathbf{b}$  and share the same storage, since after each column in  $\mathbf{A}$  is reduced, the corresponding row entry in the  $\mathbf{b}$ 's is never again used.

Here is a routine that does Gauss-Jordan elimination with full pivoting, replacing its input matrices by the desired answers. Immediately following is an overloaded version for use when there are no right-hand sides, i.e., when you want only the matrix inverse.

gaussj.h

```
void gaussj(MatDoub_Io &a, MatDoub_Io &b)
Linear equation solution by Gauss-Jordan elimination, equation (2.1.1) above. The input matrix
is a[0..n-1][0..n-1]. b[0..n-1][0..m-1] is input containing the m right-hand side vectors.
On output, a is replaced by its matrix inverse, and b is replaced by the corresponding set of
solution vectors.
{
    Int i,icol,irow,j,k,l,ll,n=a.nrows(),m=b.ncols();
    Doub big,dum,pivinv;
    VecInt indxc(n),indxr(n),ipiv(n); These integer arrays are used for bookkeeping on
    for (j=0;j<n;j++) ipiv[j]=0;          the pivoting.
    for (i=0;i<n;i++) { This is the main loop over the columns to be
        big=0.0;                         reduced.
        for (j=0;j<n;j++)                 This is the outer loop of the search for a pivot
            if (ipiv[j] != 1)             element.
                for (k=0;k<n;k++) {
                    if (ipiv[k] == 0) {
                        if (abs(a[j][k]) >= big) {
                            big=abs(a[j][k]);
                            irow=j;
                            icol=k;
                        }
                    }
                }
            }
        ++(ipiv[icol]);
    }
}
We now have the pivot element, so we interchange rows, if needed, to put the pivot
element on the diagonal. The columns are not physically interchanged, only relabeled:
indxc[i], the column of the (i + 1)th pivot element, is the (i + 1)th column that is
reduced, while indxr[i] is the row in which that pivot element was originally located.
If indxr[i] ≠ indxc[i], there is an implied column interchange. With this form of
bookkeeping, the solution b's will end up in the correct order, and the inverse matrix
will be scrambled by columns.
if (irow != icol) {
    for (l=0;l<n;l++) SWAP(a[irow][l],a[icol][l]);
    for (l=0;l<m;l++) SWAP(b[irow][l],b[icol][l]);
}
indxr[i]=irow;

```

We are now ready to divide the pivot row by the pivot element, located at `irow` and `icol`.

```

    indx[i]=icol;
    if (a[icol][icol] == 0.0) throw("gaussj: Singular Matrix");
    pivinv=1.0/a[icol][icol];
    a[icol][icol]=1.0;
    for (l=0;l<n;l++) a[icol][l] *= pivinv;
    for (l=0;l<m;l++) b[icol][l] *= pivinv;
    for (ll=0;ll<n;ll++) Next, we reduce the rows...
        if (ll != icol) { ...except for the pivot one, of course.
            dum=a[ll][icol];
            a[ll][icol]=0.0;
            for (l=0;l<n;l++) a[ll][l] -= a[icol][l]*dum;
            for (l=0;l<m;l++) b[ll][l] -= b[icol][l]*dum;
        }
    }
}

This is the end of the main loop over columns of the reduction. It only remains to unscramble the solution in view of the column interchanges. We do this by interchanging pairs of columns in the reverse order that the permutation was built up.

for (l=n-1;l>=0;l--) {
    if (indx[1] != indx[1])
        for (k=0;k<n;k++)
            SWAP(a[k][indx[1]],a[k][indx[1]]);
    }
}

And we are done.
}

```

```

void gaussj(MatDoub_IO &a)
Overloaded version with no right-hand sides. Replaces a by its inverse.
{
    MatDoub b(a.nrows(),0);      Dummy vector with zero columns.
    gaussj(a,b);
}

```

### 2.1.3 Row versus Column Elimination Strategies

The above discussion can be amplified by a modest amount of formalism. Row operations on a matrix  $\mathbf{A}$  correspond to pre- (that is, left-) multiplication by some simple matrix  $\mathbf{R}$ . For example, the matrix  $\mathbf{R}$  with components

$$R_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and } i \neq 2, 4 \\ 1 & \text{if } i = 2, j = 4 \\ 1 & \text{if } i = 4, j = 2 \\ 0 & \text{otherwise} \end{cases} \quad (2.1.5)$$

effects the interchange of rows 2 and 4. Gauss-Jordan elimination by row operations alone (including the possibility of *partial* pivoting) consists of a series of such left-multiplications, yielding successively

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ (\cdots \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{R}_0 \cdot \mathbf{A}) \cdot \mathbf{x} &= \cdots \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{R}_0 \cdot \mathbf{b} \\ (1) \cdot \mathbf{x} &= \cdots \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{R}_0 \cdot \mathbf{b} \\ \mathbf{x} &= \cdots \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{R}_0 \cdot \mathbf{b} \end{aligned} \quad (2.1.6)$$

The key point is that since the  $\mathbf{R}$ 's build from right to left, the right-hand side is simply transformed at each stage from one vector to another.

Column operations, on the other hand, correspond to post-, or right-, multiplications by simple matrices, call them  $\mathbf{C}$ . The matrix in equation (2.1.5), if right-multiplied onto a matrix  $\mathbf{A}$ , will interchange *columns* 2 and 4 of  $\mathbf{A}$ . Elimination by column operations involves (conceptually) inserting a column operator, *and also its inverse*, between the matrix  $\mathbf{A}$  and the

unknown vector  $\mathbf{x}$ :

$$\begin{aligned}
 \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\
 \mathbf{A} \cdot \mathbf{C}_0 \cdot \mathbf{C}_0^{-1} \cdot \mathbf{x} &= \mathbf{b} \\
 \mathbf{A} \cdot \mathbf{C}_0 \cdot \mathbf{C}_1 \cdot \mathbf{C}_1^{-1} \cdot \mathbf{C}_0^{-1} \cdot \mathbf{x} &= \mathbf{b} \\
 (\mathbf{A} \cdot \mathbf{C}_0 \cdot \mathbf{C}_1 \cdot \mathbf{C}_2 \cdots) \cdots \mathbf{C}_2^{-1} \cdot \mathbf{C}_1^{-1} \cdot \mathbf{C}_0^{-1} \cdot \mathbf{x} &= \mathbf{b} \\
 (\mathbf{1}) \cdots \mathbf{C}_2^{-1} \cdot \mathbf{C}_1^{-1} \cdot \mathbf{C}_0^{-1} \cdot \mathbf{x} &= \mathbf{b}
 \end{aligned} \tag{2.1.7}$$

which (peeling off the  $\mathbf{C}^{-1}$ 's one at a time) implies a solution

$$\mathbf{x} = \mathbf{C}_0 \cdot \mathbf{C}_1 \cdot \mathbf{C}_2 \cdots \mathbf{b} \tag{2.1.8}$$

Notice the essential difference between equation (2.1.8) and equation (2.1.6). In the latter case, the  $\mathbf{C}$ 's must be applied to  $\mathbf{b}$  in the *reverse order* from that in which they become known. That is, they must all be stored along the way. This requirement greatly reduces the usefulness of column operations, generally restricting them to simple permutations, for example in support of full pivoting.

#### CITED REFERENCES AND FURTHER READING:

- Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press).[1]
- Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Example 5.2, p. 282.
- Bevington, P.R., and Robinson, D.K. 2002, *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed. (New York: McGraw-Hill), p. 247.
- Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.3–1.

## 2.2 Gaussian Elimination with Backsubstitution

Any discussion of *Gaussian elimination with backsubstitution* is primarily pedagogical. The method stands between full elimination schemes such as Gauss-Jordan, and triangular decomposition schemes such as will be discussed in the next section. Gaussian elimination reduces a matrix not all the way to the identity matrix, but only halfway, to a matrix whose components on the diagonal and above (say) remain nontrivial. Let us now see what advantages accrue.

Suppose that in doing Gauss-Jordan elimination, as described in §2.1, we at each stage subtract away rows only *below* the then-current pivot element. When  $a_{11}$  is the pivot element, for example, we divide the row 1 by its value (as before), but now use the pivot row to zero only  $a_{21}$  and  $a_{31}$ , not  $a_{01}$  (see equation 2.1.1). Suppose, also, that we do only partial pivoting, never interchanging columns, so that the order of the unknowns never needs to be modified.

Then, when we have done this for all the pivots, we will be left with a reduced equation that looks like this (in the case of a single right-hand side vector):

$$\begin{bmatrix} a'_{00} & a'_{01} & a'_{02} & a'_{03} \\ 0 & a'_{11} & a'_{12} & a'_{13} \\ 0 & 0 & a'_{22} & a'_{23} \\ 0 & 0 & 0 & a'_{33} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \end{bmatrix} \quad (2.2.1)$$

Here the primes signify that the  $a$ 's and  $b$ 's do not have their original numerical values, but have been modified by all the row operations in the elimination to this point. The procedure up to this point is termed *Gaussian elimination*.

### 2.2.1 Backsubstitution

But how do we solve for the  $x$ 's? The last  $x$  ( $x_3$  in this example) is already isolated, namely

$$x_3 = b'_3/a'_{33} \quad (2.2.2)$$

With the last  $x$  known we can move to the penultimate  $x$ ,

$$x_2 = \frac{1}{a'_{22}}[b'_2 - x_3 a'_{23}] \quad (2.2.3)$$

and then proceed with the  $x$  before that one. The typical step is

$$x_i = \frac{1}{a'_{ii}} \left[ b'_i - \sum_{j=i+1}^{N-1} a'_{ij} x_j \right] \quad (2.2.4)$$

The procedure defined by equation (2.2.4) is called *backsubstitution*. The combination of Gaussian elimination and backsubstitution yields a solution to the set of equations.

The advantage of Gaussian elimination and backsubstitution over Gauss-Jordan elimination is simply that the former is faster in raw operations count: The innermost loops of Gauss-Jordan elimination, each containing one subtraction and one multiplication, are executed  $N^3$  and  $N^2m$  times (where there are  $N$  equations and unknowns, and  $m$  different right-hand sides). The corresponding loops in Gaussian elimination are executed only  $\frac{1}{3}N^3$  times (only half the matrix is reduced, and the increasing numbers of predictable zeros reduce the count to one-third), and  $\frac{1}{2}N^2m$  times, respectively. Each backsubstitution of a right-hand side is  $\frac{1}{2}N^2$  executions of a similar loop (one multiplication plus one subtraction). For  $m \ll N$  (only a few right-hand sides) Gaussian elimination thus has about a factor three advantage over Gauss-Jordan. (We could reduce this advantage to a factor 1.5 by *not* computing the inverse matrix as part of the Gauss-Jordan scheme.)

For computing the inverse matrix (which we can view as the case of  $m = N$  right-hand sides, namely the  $N$  unit vectors that are the columns of the identity matrix), Gaussian elimination and backsubstitution at first glance require  $\frac{1}{3}N^3$  (matrix reduction) +  $\frac{1}{2}N^3$  (right-hand side manipulations) +  $\frac{1}{2}N^3$  ( $N$  backsubstitutions) =  $\frac{4}{3}N^3$  loop executions, which is more than the  $N^3$  for Gauss-Jordan. However, the unit vectors are quite special in containing all zeros except for one element. If this

is taken into account, the right-side manipulations can be reduced to only  $\frac{1}{6}N^3$  loop executions, and, for matrix inversion, the two methods have identical efficiencies.

Both Gaussian elimination and Gauss-Jordan elimination share the disadvantage that all right-hand sides must be known in advance. The *LU* decomposition method in the next section does not share that deficiency, and also has an equally small operations count, both for solution with any number of right-hand sides and for matrix inversion.

#### CITED REFERENCES AND FURTHER READING:

- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.3–1.
- Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), §2.1.
- Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.2.1.
- Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

## 2.3 LU Decomposition and Its Applications

Suppose we are able to write the matrix  $\mathbf{A}$  as a product of two matrices,

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \quad (2.3.1)$$

where  $\mathbf{L}$  is *lower triangular* (has elements only on the diagonal and below) and  $\mathbf{U}$  is *upper triangular* (has elements only on the diagonal and above). For the case of a  $4 \times 4$  matrix  $\mathbf{A}$ , for example, equation (2.3.1) would look like this:

$$\begin{bmatrix} \alpha_{00} & 0 & 0 & 0 \\ \alpha_{10} & \alpha_{11} & 0 & 0 \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \cdot \begin{bmatrix} \beta_{00} & \beta_{01} & \beta_{02} & \beta_{03} \\ 0 & \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & 0 & \beta_{33} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \quad (2.3.2)$$

We can use a decomposition such as (2.3.1) to solve the linear set

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \quad (2.3.3)$$

by first solving for the vector  $\mathbf{y}$  such that

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \quad (2.3.4)$$

and then solving

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \quad (2.3.5)$$

What is the advantage of breaking up one linear set into two successive ones? The advantage is that the solution of a triangular set of equations is quite trivial, as

we have already seen in §2.2 (equation 2.2.4). Thus, equation (2.3.4) can be solved by *forward substitution* as follows:

$$\begin{aligned} y_0 &= \frac{b_0}{\alpha_{00}} \\ y_i &= \frac{1}{\alpha_{ii}} \left[ b_i - \sum_{j=0}^{i-1} \alpha_{ij} y_j \right] \quad i = 1, 2, \dots, N-1 \end{aligned} \tag{2.3.6}$$

while (2.3.5) can then be solved by *backsubstitution* exactly as in equations (2.2.2) – (2.2.4),

$$\begin{aligned} x_{N-1} &= \frac{y_{N-1}}{\beta_{N-1,N-1}} \\ x_i &= \frac{1}{\beta_{ii}} \left[ y_i - \sum_{j=i+1}^{N-1} \beta_{ij} x_j \right] \quad i = N-2, N-3, \dots, 0 \end{aligned} \tag{2.3.7}$$

Equations (2.3.6) and (2.3.7) total (for each right-hand side  $\mathbf{b}$ )  $N^2$  executions of an inner loop containing one multiply and one add. If we have  $N$  right-hand sides that are the unit column vectors (which is the case when we are inverting a matrix), then taking into account the leading zeros reduces the total execution count of (2.3.6) from  $\frac{1}{2}N^3$  to  $\frac{1}{6}N^3$ , while (2.3.7) is unchanged at  $\frac{1}{2}N^3$ .

Notice that, once we have the *LU* decomposition of  $\mathbf{A}$ , we can solve with as many right-hand sides as we then care to, one at a time. This is a distinct advantage over the methods of §2.1 and §2.2.

### 2.3.1 Performing the LU Decomposition

How then can we solve for  $\mathbf{L}$  and  $\mathbf{U}$ , given  $\mathbf{A}$ ? First, we write out the  $i,j$ th component of equation (2.3.1) or (2.3.2). That component always is a sum beginning with

$$\alpha_{i0}\beta_{0j} + \dots = a_{ij}$$

The number of terms in the sum depends, however, on whether  $i$  or  $j$  is the smaller number. We have, in fact, the three cases,

$$i < j : \quad \alpha_{i0}\beta_{0j} + \alpha_{i1}\beta_{1j} + \dots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{2.3.8}$$

$$i = j : \quad \alpha_{i0}\beta_{0j} + \alpha_{i1}\beta_{1j} + \dots + \alpha_{ii}\beta_{jj} = a_{ij} \tag{2.3.9}$$

$$i > j : \quad \alpha_{i0}\beta_{0j} + \alpha_{i1}\beta_{1j} + \dots + \alpha_{ij}\beta_{jj} = a_{ij} \tag{2.3.10}$$

Equations (2.3.8) – (2.3.10) total  $N^2$  equations for the  $N^2 + N$  unknown  $\alpha$ 's and  $\beta$ 's (the diagonal being represented twice). Since the number of unknowns is greater than the number of equations, we are invited to specify  $N$  of the unknowns arbitrarily and then try to solve for the others. In fact, as we shall see, it is always possible to take

$$\alpha_{ii} \equiv 1 \quad i = 0, \dots, N-1 \tag{2.3.11}$$

A surprising procedure, now, is *Crout's algorithm*, which quite trivially solves the set of  $N^2 + N$  equations (2.3.8) – (2.3.11) for all the  $\alpha$ 's and  $\beta$ 's by just arranging the equations in a certain order! That order is as follows:

- Set  $\alpha_{ii} = 1, i = 0, \dots, N - 1$  (equation 2.3.11).
- For each  $j = 0, 1, 2, \dots, N - 1$  do these two procedures: First, for  $i = 0, 1, \dots, j$ , use (2.3.8), (2.3.9), and (2.3.11) to solve for  $\beta_{ij}$ , namely

$$\beta_{ij} = a_{ij} - \sum_{k=0}^{i-1} \alpha_{ik} \beta_{kj} \quad (2.3.12)$$

(When  $i = 0$  in 2.3.12 the summation term is taken to mean zero.) Second, for  $i = j + 1, j + 2, \dots, N - 1$  use (2.3.10) to solve for  $\alpha_{ij}$ , namely

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left( a_{ij} - \sum_{k=0}^{j-1} \alpha_{ik} \beta_{kj} \right) \quad (2.3.13)$$

Be sure to do both procedures before going on to the next  $j$ .

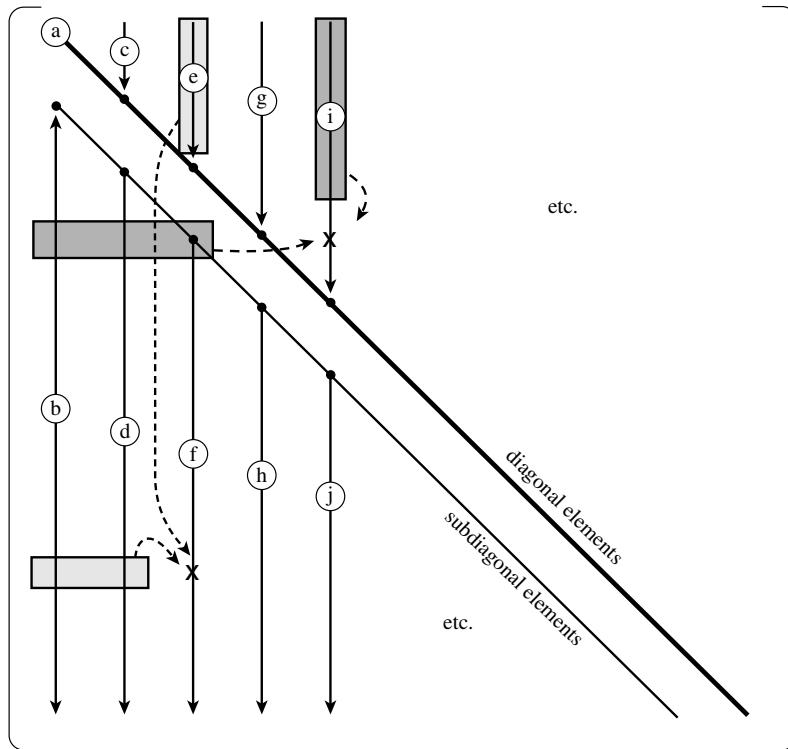
If you work through a few iterations of the above procedure, you will see that the  $\alpha$ 's and  $\beta$ 's that occur on the right-hand side of equations (2.3.12) and (2.3.13) are already determined by the time they are needed. You will also see that every  $a_{ij}$  is used only once and never again. This means that the corresponding  $\alpha_{ij}$  or  $\beta_{ij}$  can be stored in the location that the  $a$  used to occupy: the decomposition is “in place.” [The diagonal unity elements  $\alpha_{ii}$  (equation 2.3.11) are not stored at all.] In brief, Crout's method fills in the combined matrix of  $\alpha$ 's and  $\beta$ 's,

$$\begin{bmatrix} \beta_{00} & \beta_{01} & \beta_{02} & \beta_{03} \\ \alpha_{10} & \beta_{11} & \beta_{12} & \beta_{13} \\ \alpha_{20} & \alpha_{21} & \beta_{22} & \beta_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \beta_{33} \end{bmatrix} \quad (2.3.14)$$

by columns from left to right, and within each column from top to bottom (see Figure 2.3.1).

What about pivoting? Pivoting (i.e., selection of a salubrious pivot element for the division in equation 2.3.13) is absolutely essential for the stability of Crout's method. Only partial pivoting (interchange of rows) can be implemented efficiently. However this is enough to make the method stable. This means, incidentally, that we don't actually decompose the matrix  $A$  into  $LU$  form, but rather we decompose a rowwise permutation of  $A$ . (If we keep track of what that permutation is, this decomposition is just as useful as the original one would have been.)

Pivoting is slightly subtle in Crout's algorithm. The key point to notice is that equation (2.3.12) in the case of  $i = j$  (its final application) is *exactly the same* as equation (2.3.13) except for the division in the latter equation; in both cases the upper limit of the sum is  $k = j - 1 (= i - 1)$ . This means that we don't have to commit ourselves as to whether the diagonal element  $\beta_{jj}$  is the one that happens to fall on the diagonal in the first instance, or whether one of the (undivided)  $\alpha_{ij}$ 's below it in the column,  $i = j + 1, \dots, N - 1$ , is to be “promoted” to become the diagonal  $\beta$ . This can be decided after all the candidates in the column are in hand. As you should be able to guess by now, we will choose the largest one as the diagonal  $\beta$  (pivot



**Figure 2.3.1.** Crout’s algorithm for  $LU$  decomposition of a matrix. Elements of the original matrix are modified in the order indicated by lowercase letters: a, b, c, etc. Shaded boxes show the previously modified elements that are used in modifying two typical elements, each indicated by an “X”.

element), and then do all the divisions by that element *en masse*. This is *Crout’s method with partial pivoting*. Our implementation has one additional wrinkle: It initially finds the largest element in each row, and subsequently (when it is looking for the maximal pivot element) scales the comparison *as if* we had initially scaled all the equations to make their maximum coefficient equal to unity; this is the *implicit pivoting* mentioned in §2.1.

The inner loop of the  $LU$  decomposition, equations (2.3.12) and (2.3.13), resembles the inner loop of matrix multiplication. There is a triple loop over the indices  $i$ ,  $j$ , and  $k$ . There are six permutations of the order in which these loops can be done. The straightforward implementation of Crout’s algorithm corresponds to the  $ijk$  permutation, where the order of the indices is the order of the loops from outermost to innermost. On modern processors with a hierarchy of cache memory, and when matrices are stored by rows, the fastest execution time is usually the  $kij$  or  $ikj$  ordering. Pivoting is easier with  $kij$  ordering, so that is the implementation we use. This is called “outer product Gaussian elimination” by Golub and Van Loan [1].

$LU$  decomposition is well suited for implementation as an object (a `class` or `struct`). The constructor performs the decomposition, and the object itself stores the result. Then, a method for forward- and backsubstitution can be called once, or many times, to solve for one or more right-hand sides. Methods for additional functionality are also easy to include. The object’s declaration looks like this:

ludcmp.h

```
struct LUdcmp
Object for solving linear equations  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$  using  $LU$  decomposition, and related functions.
{
    Int n;
    MatDoub lu;                                Stores the decomposition.
    VecInt indx;                               Stores the permutation.
    Doub d;                                    Used by det.
    LUdcmp(MatDoub_I &a);                     Constructor. Argument is the matrix  $\mathbf{A}$ .
    void solve(VecDoub_I &b, VecDoub_0 &x);   Solve for a single right-hand side.
    void solve(MatDoub_I &b, MatDoub_0 &x);   Solve for multiple right-hand sides.
    void inverse(MatDoub_0 &ainv);             Calculate matrix inverse  $\mathbf{A}^{-1}$ .
    Doub det();                                Return determinant of  $\mathbf{A}$ .
    void mprove(VecDoub_I &b, VecDoub_10 &x); Discussed in §2.5.
    MatDoub_I &aref;                           Used only by mprove.
};
```

Here is the implementation of the constructor, whose argument is the input matrix that is to be  $LU$  decomposed. The input matrix is not altered; a copy is made, on which outer product Gaussian elimination is then done in-place.

ludcmp.h

```
LUdcmp::LUdcmp(MatDoub_I &a) : n(a.nrows()), lu(a), aref(a), indx(n) {
Given a matrix a[0..n-1][0..n-1], this routine replaces it by the  $LU$  decomposition of a
rowwise permutation of itself. a is input. On output, it is arranged as in equation (2.3.14)
above; indx[0..n-1] is an output vector that records the row permutation effected by the
partial pivoting; d is output as  $\pm 1$  depending on whether the number of row interchanges
was even or odd, respectively. This routine is used in combination with solve to solve linear
equations or invert a matrix.

    const Doub TINY=1.0e-40;                      A small number.
    Int i,imax,j,k;
    Doub big,temp;
    VecDoub vv(n);                                vv stores the implicit scaling of each row.
    d=1.0;                                         No row interchanges yet.
    for (i=0;i<n;i++) {                           Loop over rows to get the implicit scaling infor-
        big=0.0;                                     mation.
        for (j=0;j<n;j++)
            if ((temp=abs(lu[i][j])) > big) big=temp;
        if (big == 0.0) throw("Singular matrix in LUdcmp");
        No nonzero largest element.
        vv[i]=1.0/big;                               Save the scaling.
    }
    for (k=0;k<n;k++) {                           This is the outermost  $kij$  loop.
        big=0.0;                                     Initialize for the search for largest pivot element.
        for (i=k;i<n;i++) {
            temp=vv[i]*abs(lu[i][k]);
            if (temp > big) {
                big=temp;
                imax=i;
            }
        }
        if (k != imax) {                            Do we need to interchange rows?
            for (j=0;j<n;j++) {                    Yes, do so...
                temp=lu[imax][j];
                lu[imax][j]=lu[k][j];
                lu[k][j]=temp;
            }
            d = -d;                                ...and change the parity of d.
            vv[imax]=vv[k];                         Also interchange the scale factor.
        }
        indx[k]=imax;
        if (lu[k][k] == 0.0) lu[k][k]=TINY;
    }
If the pivot element is zero, the matrix is singular (at least to the precision of the
algorithm). For some applications on singular matrices, it is desirable to substitute
```

```

TINY for zero.
for (i=k+1;i<n;i++) {
    temp=lu[i][k] /= lu[k][k]; Divide by the pivot element.
    for (j=k+1;j<n;j++)           Innermost loop: reduce remaining submatrix.
        lu[i][j] -= temp*lu[k][j];
}
}
}

```

Once the LUdcmp object is constructed, two functions implementing equations (2.3.6) and (2.3.7) are available for solving linear equations. The first solves a single right-hand side vector **b** for a solution vector **x**. The second simultaneously solves multiple right-hand vectors, arranged as the columns of a matrix **B**. In other words, it calculates the matrix  $\mathbf{A}^{-1} \cdot \mathbf{B}$ .

```
void LUdcmp::solve(VecDoub_I &b, VecDoub_O &x)
```

ludcmp.h

Solves the set of  $n$  linear equations  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$  using the stored *LU* decomposition of **A**. **b**[0..n-1] is input as the right-hand side vector **b**, while **x** returns the solution vector **x**; **b** and **x** may reference the same vector, in which case the solution overwrites the input. This routine takes into account the possibility that **b** will begin with many zero elements, so it is efficient for use in matrix inversion.

```
{
    Int i,ii=0,ip,j;
    Doub sum;
    if (b.size() != n || x.size() != n)
        throw("LUdcmp::solve bad sizes");
    for (i=0;i<n;i++) x[i] = b[i];
    for (i=0;i<n;i++) {           When ii is set to a positive value, it will become the
        ip=indx[i];                 index of the first nonvanishing element of b. We now
        sum=x[ip];                  do the forward substitution, equation (2.3.6). The
        x[ip]=x[i];                  only new wrinkle is to unscramble the permutation
        if (ii != 0)                  as we go.
            for (j=ii-1;j<i;j++) sum -= lu[i][j]*x[j];
        else if (sum != 0.0)         A nonzero element was encountered, so from now on we
            ii=i+1;                   will have to do the sums in the loop above.
        x[i]=sum;
    }
    for (i=n-1;i>=0;i--) {       Now we do the backsubstitution, equation (2.3.7).
        sum=x[i];
        for (j=i+1;j<n;j++) sum -= lu[i][j]*x[j];
        x[i]=sum/lu[i][i];          Store a component of the solution vector X.
    }                               All done!
}
```

```
void LUdcmp::solve(MatDoub_I &b, MatDoub_O &x)
```

Solves  $m$  sets of  $n$  linear equations  $\mathbf{A} \cdot \mathbf{X} = \mathbf{B}$  using the stored *LU* decomposition of **A**. The matrix **b**[0..n-1][0..m-1] inputs the right-hand sides, while **x**[0..n-1][0..m-1] returns the solution  $\mathbf{A}^{-1} \cdot \mathbf{B}$ . **b** and **x** may reference the same matrix, in which case the solution overwrites the input.

```
{
    int i,j,m=b.ncols();
    if (b.nrows() != n || x.nrows() != n || b.ncols() != x.ncols())
        throw("LUdcmp::solve bad sizes");
    VecDoub xx(n);
    for (j=0;j<m;j++) {           Copy and solve each column in turn.
        for (i=0;i<n;i++) xx[i] = b[i][j];
        solve(xx,xx);
        for (i=0;i<n;i++) x[i][j] = xx[i];
    }
}
```

The  $LU$  decomposition in `LUDcmp` requires about  $\frac{1}{3}N^3$  executions of the inner loops (each with one multiply and one add). This is thus the operation count for solving one (or a few) right-hand sides, and is a factor of 3 better than the Gauss-Jordan routine `gaussj` that was given in §2.1, and a factor of 1.5 better than a Gauss-Jordan routine (not given) that does not compute the inverse matrix. For inverting a matrix, the total count (including the forward- and backsubstitution as discussed following equation 2.3.7 above) is  $(\frac{1}{3} + \frac{1}{6} + \frac{1}{2})N^3 = N^3$ , the same as `gaussj`.

To summarize, this is the preferred way to solve the linear set of equations  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ :

```
const Int n = ...
MatDoub a(n,n);
VecDoub b(n),x(n);
...
LUDcmp alu(a);
alu.solve(b,x);
```

The answer will be given back in `x`. Your original matrix `a` and vector `b` are not altered. If you need to recover the storage in the object `alu`, then start a temporary scope with “`{`” before `alu` is declared, and end that scope with “`}`” when you want `alu` to be destroyed.

If you subsequently want to solve a set of equations with the same `A` but a different right-hand side `b`, you repeat *only*

```
alu.solve(b,x);
```

### 2.3.2 Inverse of a Matrix

`LUDcmp` has a member function that gives the inverse of the matrix `A`. Simply, it creates an identity matrix and then invokes the appropriate `solve` method.

`ludcmp.h`

```
void LUDcmp::inverse(MatDoub_ &ainv)
Using the stored LU decomposition, return in ainv the matrix inverse A-1.
{
    Int i,j;
    ainv.resize(n,n);
    for (i=0;i<n;i++) {
        for (j=0;j<n;j++) ainv[i][j] = 0.;
        ainv[i][i] = 1.;
    }
    solve(ainv,ainv);
}
```

The matrix `ainv` will now contain the inverse of the original matrix `a`. Alternatively, there is nothing wrong with using a Gauss-Jordan routine like `gaussj` (§2.1) to invert a matrix in place, destroying the original. Both methods have practically the same operations count.

### 2.3.3 Determinant of a Matrix

The determinant of an  $LU$  decomposed matrix is just the product of the diagonal elements,

$$\det = \prod_{j=0}^{N-1} \beta_{jj} \quad (2.3.15)$$

We don't, recall, compute the decomposition of the original matrix, but rather a decomposition of a rowwise permutation of it. Luckily, we have kept track of whether the number of row interchanges was even or odd, so we just preface the product by the corresponding sign. (You now finally know the purpose of **d** in the **LUDcmp** structure.)

**Doub LUDcmp::det()**

Using the stored *LU* decomposition, return the determinant of the matrix **A**.

```
{
    Doub dd = d;
    for (Int i=0;i<n;i++) dd *= lu[i][i];
    return dd;
}
```

**ludcmp.h**

For a matrix of any substantial size, it is quite likely that the determinant will overflow or underflow your computer's floating-point dynamic range. In such a case you can easily add another member function that, e.g., divides by powers of ten, to keep track of the scale separately, or, e.g., accumulates the sum of logarithms of the absolute values of the factors and the sign separately.

### 2.3.4 Complex Systems of Equations

If your matrix **A** is real, but the right-hand side vector is complex, say **b** + *i* **d**, then (i) *LU* decompose **A** in the usual way, (ii) backsubstitute **b** to get the real part of the solution vector, and (iii) backsubstitute **d** to get the imaginary part of the solution vector.

If the matrix itself is complex, so that you want to solve the system

$$(\mathbf{A} + i\mathbf{C}) \cdot (\mathbf{x} + i\mathbf{y}) = (\mathbf{b} + i\mathbf{d}) \quad (2.3.16)$$

then there are two possible ways to proceed. The best way is to rewrite **LUDcmp** with complex routines. Complex modulus substitutes for absolute value in the construction of the scaling vector **vv** and in the search for the largest pivot elements. Everything else goes through in the obvious way, with complex arithmetic used as needed.

A quick-and-dirty way to solve complex systems is to take the real and imaginary parts of (2.3.16), giving

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} - \mathbf{C} \cdot \mathbf{y} &= \mathbf{b} \\ \mathbf{C} \cdot \mathbf{x} + \mathbf{A} \cdot \mathbf{y} &= \mathbf{d} \end{aligned} \quad (2.3.17)$$

which can be written as a  $2N \times 2N$  set of *real* equations,

$$\begin{pmatrix} \mathbf{A} & -\mathbf{C} \\ \mathbf{C} & \mathbf{A} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ \mathbf{d} \end{pmatrix} \quad (2.3.18)$$

and then solved with **LUDcmp**'s routines in their present forms. This scheme is a factor of 2 inefficient in storage, since **A** and **C** are stored twice. It is also a factor of 2 inefficient in time, since the complex multiplies in a complexified version of the routines would each use 4 real multiplies, while the solution of a  $2N \times 2N$  problem involves 8 times the work of an  $N \times N$  one. If you can tolerate these factor-of-two inefficiencies, then equation (2.3.18) is an easy way to proceed.

#### CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), Chapter 4.[1]

- Anderson, E., et al. 1999, LAPACK User's Guide, 3rd ed. (Philadelphia: S.I.A.M.). Online with software at 2007+, <http://www.netlib.org/lapack>.
- Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §3.3, and p. 50.
- Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 9, 16, and 18.
- Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §4.1.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.11.
- Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

## 2.4 Tridiagonal and Band-Diagonal Systems of Equations

The special case of a system of linear equations that is *tridiagonal*, that is, has nonzero elements only on the diagonal plus or minus one column, is one that occurs frequently. Also common are systems that are *band-diagonal*, with nonzero elements only along a few diagonal lines adjacent to the main diagonal (above and below).

For tridiagonal sets, the procedures of *LU* decomposition, forward- and back-substitution each take only  $O(N)$  operations, and the whole solution can be encoded very concisely. The resulting routine `tridag` is one that we will use in later chapters.

Naturally, one does not reserve storage for the full  $N \times N$  matrix, but only for the nonzero components, stored as three vectors. The set of equations to be solved is

$$\begin{bmatrix} b_0 & c_0 & 0 & \cdots \\ a_1 & b_1 & c_1 & \cdots \\ & \cdots & & \\ & \cdots & a_{N-2} & b_{N-2} & c_{N-2} \\ & \cdots & 0 & a_{N-1} & b_{N-1} \end{bmatrix} \cdot \begin{bmatrix} u_0 \\ u_1 \\ \cdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ \cdots \\ r_{N-2} \\ r_{N-1} \end{bmatrix} \quad (2.4.1)$$

Notice that  $a_0$  and  $c_{N-1}$  are undefined and are not referenced by the routine that follows.

`tridag.h`

```
void tridag(VecDoub_I &a, VecDoub_I &b, VecDoub_I &c, VecDoub_I &r, VecDoub_O &u)
Solves for a vector u[0..n-1] the tridiagonal linear set given by equation (2.4.1). a[0..n-1],
b[0..n-1], c[0..n-1], and r[0..n-1] are input vectors and are not modified.
{
    Int j,n=a.size();
    Doub bet;
    VecDoub gam(n);                                One vector of workspace, gam, is needed.
    if (b[0] == 0.0) throw("Error 1 in tridag");
    If this happens, then you should rewrite your equations as a set of order N - 1, with u1
    trivially eliminated.
    u[0]=r[0]/(bet=b[0]);
    for (j=1;j<n;j++) {                          Decomposition and forward substitution.
        gam[j]=c[j-1]/bet;
        bet=b[j]-a[j]*gam[j];
    }
}
```

```

    if (bet == 0.0) throw("Error 2 in tridag");
    u[j]=(r[j]-a[j]*u[j-1])/bet;           Algorithm fails; see below.
}
for (j=(n-2);j>=0;j--)
    u[j] = gam[j+1]*u[j+1];             Backsubstitution.
}

```

There is no pivoting in `tridag`. It is for this reason that `tridag` can fail even when the underlying matrix is nonsingular: A zero pivot can be encountered even for a nonsingular matrix. In practice, this is not something to lose sleep about. The kinds of problems that lead to tridiagonal linear sets usually have additional properties which guarantee that the algorithm in `tridag` will succeed. For example, if

$$|b_j| > |a_j| + |c_j| \quad j = 0, \dots, N - 1 \quad (2.4.2)$$

(called *diagonal dominance*), then it can be shown that the algorithm cannot encounter a zero pivot.

It is possible to construct special examples in which the lack of pivoting in the algorithm causes numerical instability. In practice, however, such instability is almost never encountered — unlike the general matrix problem where pivoting is essential.

The tridiagonal algorithm is the rare case of an algorithm that, in practice, is more robust than theory says it should be. Of course, should you ever encounter a problem for which `tridag` fails, you can instead use the more general method for band-diagonal systems, described below (the `Bandec` object).

Some other matrix forms consisting of tridiagonal with a small number of additional elements (e.g., upper right and lower left corners) also allow rapid solution; see §2.7.

### 2.4.1 Parallel Solution of Tridiagonal Systems

It is possible to solve tridiagonal systems doing many of the operations in parallel. We illustrate by the special case with  $N = 7$ :

$$\begin{bmatrix} b_0 & c_0 & & & & & \\ a_1 & b_1 & c_1 & & & & \\ & a_2 & b_2 & c_2 & & & \\ & & a_3 & b_3 & c_3 & & \\ & & & a_4 & b_4 & c_4 & \\ & & & & a_5 & b_5 & c_5 \\ & & & & & a_6 & b_6 \end{bmatrix} \cdot \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{bmatrix} \quad (2.4.3)$$

The basic idea is to partition the problem into even and odd elements, recurse to solve the latter, and then solve the former in parallel. Specifically, we first rewrite equation (2.4.3) by permuting its rows and columns,

$$\begin{bmatrix} b_0 & & c_0 & & & & \\ & b_2 & & a_2 & c_2 & & \\ & & b_4 & & a_4 & c_4 & \\ & & & b_6 & & a_6 & \\ a_1 & c_1 & & b_1 & & & \\ a_3 & c_3 & & b_3 & & & \\ a_5 & c_5 & & b_5 & & & \end{bmatrix} \cdot \begin{bmatrix} u_0 \\ u_2 \\ u_4 \\ u_6 \\ u_1 \\ u_3 \\ u_5 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_2 \\ r_4 \\ r_6 \\ r_1 \\ r_3 \\ r_5 \end{bmatrix} \quad (2.4.4)$$

Now observe that, by row operations that subtract multiples of the first four rows from each of the last three rows, we can eliminate all nonzero elements in the lower-left quadrant. The price we pay is bringing some new elements into the lower-right quadrant, whose

nonzero elements we now call  $x$ 's,  $y$ 's, and  $z$ 's. We call the modified right-hand sides  $q$ . The transformed problem is now

$$\begin{bmatrix} b_0 & & c_0 & & & \\ & b_2 & & a_2 & c_2 & \\ & & b_4 & & a_4 & c_4 \\ & & & b_6 & & a_6 \\ & & & & y_0 & z_0 \\ & & & & x_1 & y_1 & z_1 \\ & & & & x_2 & y_2 \end{bmatrix} \cdot \begin{bmatrix} u_0 \\ u_2 \\ u_4 \\ u_6 \\ u_1 \\ u_3 \\ u_5 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_2 \\ r_4 \\ r_6 \\ q_0 \\ q_1 \\ q_2 \end{bmatrix} \quad (2.4.5)$$

Notice that the last three rows form a new, smaller, tridiagonal problem, which we can solve simply by recursing. Once its solution is known, the first four rows can be solved by a simple, parallelizable, substitution. For discussion of this and related methods for parallelizing tridiagonal systems, and references to the literature, see [2].

## 2.4.2 Band-Diagonal Systems

Where tridiagonal systems have nonzero elements only on the diagonal plus or minus one, band-diagonal systems are slightly more general and have (say)  $m_1 \geq 0$  nonzero elements immediately to the left of (below) the diagonal and  $m_2 \geq 0$  nonzero elements immediately to its right (above it). Of course, this is only a useful classification if  $m_1$  and  $m_2$  are both  $\ll N$ . In that case, the solution of the linear system by  $LU$  decomposition can be accomplished much faster, and in much less storage, than for the general  $N \times N$  case.

The precise definition of a band-diagonal matrix with elements  $a_{ij}$  is that

$$a_{ij} = 0 \quad \text{when} \quad j > i + m_2 \quad \text{or} \quad i > j + m_1 \quad (2.4.6)$$

Band-diagonal matrices are stored and manipulated in a so-called compact form, which results if the matrix is tilted  $45^\circ$  clockwise, so that its nonzero elements lie in a long, narrow matrix with  $m_1 + 1 + m_2$  columns and  $N$  rows. This is best illustrated by an example: The band-diagonal matrix

$$\begin{pmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 5 & 0 & 0 & 0 & 0 \\ 9 & 2 & 6 & 5 & 0 & 0 & 0 \\ 0 & 3 & 5 & 8 & 9 & 0 & 0 \\ 0 & 0 & 7 & 9 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 8 & 4 & 6 \\ 0 & 0 & 0 & 0 & 2 & 4 & 4 \end{pmatrix} \quad (2.4.7)$$

which has  $N = 7$ ,  $m_1 = 2$ , and  $m_2 = 1$ , is stored compactly as the  $7 \times 4$  matrix,

$$\begin{pmatrix} x & x & 3 & 1 \\ x & 4 & 1 & 5 \\ 9 & 2 & 6 & 5 \\ 3 & 5 & 8 & 9 \\ 7 & 9 & 3 & 2 \\ 3 & 8 & 4 & 6 \\ 2 & 4 & 4 & x \end{pmatrix} \quad (2.4.8)$$

Here  $x$  denotes elements that are wasted space in the compact format; these will not be referenced by any manipulations and can have arbitrary values. Notice that the diagonal of the original matrix appears in column  $m_1$ , with subdiagonal elements to its left and superdiagonal elements to its right.

The simplest manipulation of a band-diagonal matrix, stored compactly, is to multiply it by a vector to its right. Although this is algorithmically trivial, you might want to study the following routine as an example of how to pull nonzero elements  $a_{ij}$  out of the compact storage format in an orderly fashion.

```

void banmul(MatDoub_I &a, const Int m1, const Int m2, VecDoub_I &x,
            VecDoub_0 &b)                                banded.h
Matrix multiply  $\mathbf{b} = \mathbf{A} \cdot \mathbf{x}$ , where  $\mathbf{A}$  is band-diagonal with  $m_1$  rows below the diagonal and
m2 rows above. The input vector is  $x[0..n-1]$  and the output vector is  $b[0..n-1]$ . The array  $a[0..n-1][0..m1+m2]$  stores  $\mathbf{A}$  as follows: The diagonal elements are in  $a[0..n-1][m1]$ . Subdiagonal elements are in  $a[j..n-1][0..m1-1]$  with  $j > 0$  appropriate to the number of elements on each subdiagonal. Superdiagonal elements are in  $a[0..j][m1+1..m1+m2]$  with
 $j < n-1$  appropriate to the number of elements on each superdiagonal.
{
    Int i,j,k,tmploop,n=a.nrows();
    for (i=0;i<n;i++) {
        k=i-m1;
        tmploop=MIN(m1+m2+1,Int(n-k));
        b[i]=0.0;
        for (j=MAX(0,-k);j<tmploop;j++) b[i] += a[i][j]*x[j+k];
    }
}

```

It is not possible to store the  $LU$  decomposition of a band-diagonal matrix  $\mathbf{A}$  quite as compactly as the compact form of  $\mathbf{A}$  itself. The decomposition (essentially by Crout's method; see §2.3) produces additional nonzero "fill-ins." One straightforward storage scheme is to store the upper triangular factor ( $U$ ) in a space with the same shape as  $\mathbf{A}$ , and to store the lower triangular factor ( $L$ ) in a separate compact matrix of size  $N \times m_1$ . The diagonal elements of  $U$  (whose product, times  $d = \pm 1$ , gives the determinant) are in the first column of  $U$ .

Here is an object, analogous to `LUdcmp` in §2.3, for solving band-diagonal linear equations:

```

struct Bandec {
Object for solving linear equations  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$  for a band-diagonal matrix  $\mathbf{A}$ , using  $LU$  decom-
position.
    Int n,m1,m2;
    MatDoub au,al;                                Upper and lower triangular matrices, stored compactly.
    VecInt indx;
    Doub d;
    Bandec(MatDoub_I &a, const int mm1, const int mm2); Constructor.
    void solve(VecDoub_I &b, VecDoub_0 &x);      Solve a right-hand side vector.
    Doub det();                                     Return determinant of  $\mathbf{A}$ .
};

```

The constructor takes as arguments the compactly stored matrix  $\mathbf{A}$ , and the integers  $m_1$  and  $m_2$ . (One could of course define a "band-diagonal matrix object" to encapsulate these quantities, but in this brief treatment we want to keep things simple.)

```

Bandec::Bandec(MatDoub_I &a, const Int mm1, const Int mm2)                                banded.h
    : n(a.nrows()), au(a), m1(mm1), m2(mm2), al(n,m1), indx(n)
Constructor. Given an  $n \times n$  band-diagonal matrix  $\mathbf{A}$  with  $m_1$  subdiagonal rows and  $m_2$  superdiagonal rows, compactly stored in the array  $a[0..n-1][0..m1+m2]$  as described in the comment for routine banmul, an  $LU$  decomposition of a rowwise permutation of  $\mathbf{A}$  is constructed. The upper and lower triangular matrices are stored in au and al, respectively. The stored vector indx[0..n-1] records the row permutation effected by the partial pivoting; d is  $\pm 1$  depending on whether the number of row interchanges was even or odd, respectively.
{
    const Doub TINY=1.0e-40;
    Int i,j,k,l,mm;
    Doub dum;
    mm=m1+m2+1;
    l=m1;
    for (i=0;i<m1;i++) {                         Rearrange the storage a bit.
        for (j=m1-i;j<mm;j++) au[i][j-1]=au[i][j];
        l--;
        for (j=mm-l-1;j<mm;j++) au[i][j]=0.0;
    }
}

```

```

d=1.0;
l=m1;
for (k=0;k<n;k++) {                                For each row...
    dum=au[k][0];
    i=k;
    if (1<n) l++;
    for (j=k+1;j<l;j++) {                          Find the pivot element.
        if (abs(au[j][0]) > abs(dum)) {
            dum=au[j][0];
            i=j;
        }
    }
    indx[k]=i+1;
    if (dum == 0.0) au[k][0]=TINY;
    Matrix is algorithmically singular, but proceed anyway with TINY pivot (desirable in
    some applications).
    if (i != k) {                                Interchange rows.
        d = -d;
        for (j=0;j<mm;j++) SWAP(au[k][j],au[i][j]);
    }
    for (i=k+1;i<l;i++) {                        Do the elimination.
        dum=au[i][0]/au[k][0];
        al[k][i-k-1]=dum;
        for (j=1;j<mm;j++) au[i][j-1]=au[i][j]-dum*au[k][j];
        au[i][mm-1]=0.0;
    }
}
}

```

Some pivoting is possible within the storage limitations of `bandec`, and the above routine does take advantage of the opportunity. In general, when `TINY` is returned as a diagonal element of  $U$ , then the original matrix (perhaps as modified by roundoff error) is in fact singular. In this regard, `bandec` is somewhat more robust than `tridag` above, which can fail algorithmically even for nonsingular matrices; `bandec` is thus also useful (with  $m_1 = m_2 = 1$ ) for some ill-behaved tridiagonal systems.

Once the matrix  $\mathbf{A}$  has been decomposed, any number of right-hand sides can be solved in turn by repeated calls to the `solve` method, the forward- and backsubstitution routine analogous to its same-named cousin in §2.3.

`banded.h`

```

void Bandec::solve(VecDoub_I &b, VecDoub_O &x)
Given a right-hand side vector b[0..n-1], solves the band-diagonal linear equations  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ .
The solution vector  $\mathbf{x}$  is returned as x[0..n-1].
{
    Int i,j,k,l,mm;
    Doub dum;
    mm=m1+m2+1;
    l=m1;
    for (k=0;k<n;k++) x[k] = b[k];
    for (k=0;k<n;k++) {                      Forward substitution, unscrambling the permuted rows
        j=indx[k]-1;                         as we go.
        if (j!=k) SWAP(x[k],x[j]);
        if (1<n) l++;
        for (j=k+1;j<l;j++) x[j] -= al[k][j-k-1]*x[k];
    }
    l=1;
    for (i=n-1;i>=0;i--) {                  Backsubstitution.
        dum=x[i];
        for (k=1;k<l;k++) dum -= au[i][k]*x[k+i];
        x[i]=dum/au[i][0];
        if (1<mm) l++;
    }
}

```

And, finally, a method for getting the determinant:

```
Doub Bandec::det() {
    Using the stored decomposition, return the determinant of the matrix A.
    Doub dd = d;
    for (int i=0;i<n;i++) dd *= au[i][0];
    return dd;
}
```

banded.h

The routines in `Bandec` are based on the Handbook routines `bandet1` and `bansol1` in [1].

#### CITED REFERENCES AND FURTHER READING:

- Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems*; reprinted 1991 (New York: Dover), p. 74.
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), Example 5.4.3, p. 166.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.11.
- Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer), Chapter I/6.[1]
- Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §4.3.
- Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2: Architecture, Programming, and Algorithms* (Bristol and Philadelphia: Adam Hilger), §5.4.[2]

## 2.5 Iterative Improvement of a Solution to Linear Equations

Obviously it is not easy to obtain greater precision for the solution of a linear set than the precision of your computer's floating-point word. Unfortunately, for large sets of linear equations, it is not always easy to obtain precision equal to, or even comparable to, the computer's limit. In direct methods of solution, roundoff errors accumulate, and they are magnified to the extent that your matrix is close to singular. You can easily lose two or three significant figures for matrices that (you thought) were *far* from singular.

If this happens to you, there is a neat trick to restore the full machine precision, called *iterative improvement* of the solution. The theory is straightforward (see Figure 2.5.1): Suppose that a vector  $\mathbf{x}$  is the exact solution of the linear set

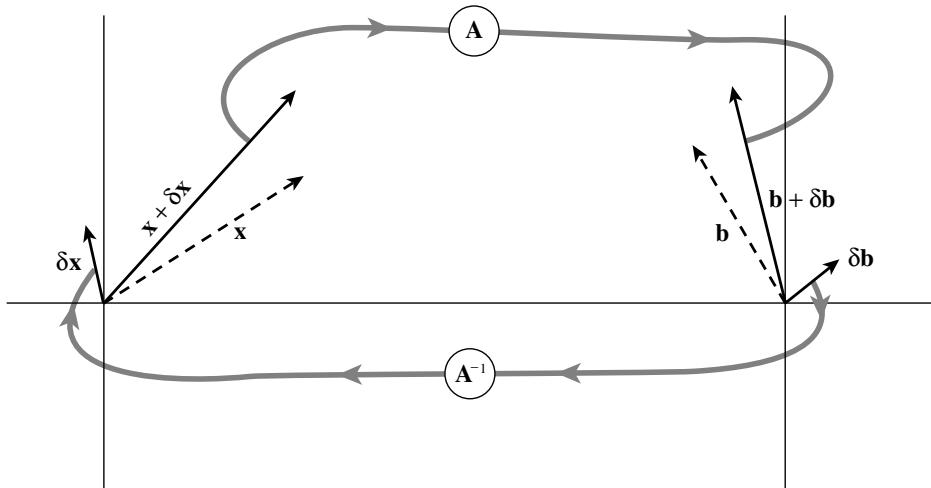
$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \quad (2.5.1)$$

You don't, however, know  $\mathbf{x}$ . You only know some slightly wrong solution  $\mathbf{x} + \delta\mathbf{x}$ , where  $\delta\mathbf{x}$  is the unknown error. When multiplied by the matrix  $\mathbf{A}$ , your slightly wrong solution gives a product slightly discrepant from the desired right-hand side  $\mathbf{b}$ , namely

$$\mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \quad (2.5.2)$$

Subtracting (2.5.1) from (2.5.2) gives

$$\mathbf{A} \cdot \delta\mathbf{x} = \delta\mathbf{b} \quad (2.5.3)$$



**Figure 2.5.1.** Iterative improvement of the solution to  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ . The first guess  $\mathbf{x} + \delta\mathbf{x}$  is multiplied by  $\mathbf{A}$  to produce  $\mathbf{b} + \delta\mathbf{b}$ . The known vector  $\mathbf{b}$  is subtracted, giving  $\delta\mathbf{b}$ . The linear set with this right-hand side is inverted, giving  $\delta\mathbf{x}$ . This is subtracted from the first guess giving an improved solution  $\mathbf{x}$ .

But (2.5.2) can also be solved, trivially, for  $\delta\mathbf{b}$ . Substituting this into (2.5.3) gives

$$\mathbf{A} \cdot \delta\mathbf{x} = \mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) - \mathbf{b} \quad (2.5.4)$$

In this equation, the whole right-hand side is known, since  $\mathbf{x} + \delta\mathbf{x}$  is the wrong solution that you want to improve. It is good to calculate the right-hand side in higher precision than the original solution, if you can, since there will be a lot of cancellation in the subtraction of  $\mathbf{b}$ . Then, we need only solve (2.5.4) for the error  $\delta\mathbf{x}$ , and then subtract this from the wrong solution to get an improved solution.

An important extra benefit occurs if we obtained the original solution by *LU* decomposition. In this case we already have the *LU* decomposed form of  $\mathbf{A}$ , and all we need do to solve (2.5.4) is compute the right-hand side, forward- and backsubstitute.

Because so much of the necessary machinery is already in *LUDcmp*, we implement iterative improvement as a member function of that class. Since iterative improvement requires the matrix  $\mathbf{A}$  (as well as its *LU* decomposition), we have, with foresight, caused *LUDcmp* to save a reference to the matrix *a* from which it was constructed. If you plan to use iterative improvement, you must not modify *a* or let it go out of scope. (No other method in *LUDcmp* makes use of this reference to *a*.)

*ludcmp.h*

```
void LUDcmp::improve(VecDoub_I &b, VecDoub_I &x)
```

Improves a solution vector  $x[0..n-1]$  of the linear set of equations  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ . The vectors  $b[0..n-1]$  and  $x[0..n-1]$  are input. On output,  $x[0..n-1]$  is modified, to an improved set of values.

```
{
```

```
    Int i,j;
```

```
    VecDoub r(n);
```

```
    for (i=0;i<n;i++) {
```

```
        Ldoub sdp = -b[i];
```

```
        for (j=0;j<n;j++)
```

```
            sdp += (Ldoub)aref[i][j] * (Ldoub)x[j];
```

Calculate the right-hand side, accumulating  
the residual in higher precision.

```

    r[i]=sdp;
}
solve(r,r);
for (i=0;i<n;i++) x[i] -= r[i];
}

```

Solve for the error term,  
and subtract it from the old solution.

Iterative improvement is *highly* recommended: It is a process of order only  $N^2$  operations (multiply vector by matrix, forward- and backsubstitute — see discussion following equation 2.3.7); it never hurts; and it can really give you your money's worth if it saves an otherwise ruined solution on which you have already spent of order  $N^3$  operations.

You can call `mprove` several times in succession if you want. Unless you are starting quite far from the true solution, one call is generally enough; but a second call to verify convergence can be reassuring.

If you cannot compute the right-hand side in equation (2.5.4) in higher precision, iterative refinement will still often improve the quality of a solution, although not in all cases as much as if higher precision is available. Many textbooks assert the contrary, but you will find the proof in [1].

## 2.5.1 More on Iterative Improvement

It is illuminating (and will be useful later in the book) to give a somewhat more solid analytical foundation for equation (2.5.4), and also to give some additional results. Implicit in the previous discussion was the notion that the solution vector  $\mathbf{x} + \delta\mathbf{x}$  has an error term; but we neglected the fact that the *LU* decomposition of  $\mathbf{A}$  is itself not exact.

A different analytical approach starts with some matrix  $\mathbf{B}_0$  that is assumed to be an *approximate* inverse of the matrix  $\mathbf{A}$ , so that  $\mathbf{B}_0 \cdot \mathbf{A}$  is approximately the identity matrix  $\mathbf{1}$ . Define the *residual matrix*  $\mathbf{R}$  of  $\mathbf{B}_0$  as

$$\mathbf{R} \equiv \mathbf{1} - \mathbf{B}_0 \cdot \mathbf{A} \quad (2.5.5)$$

which is supposed to be “small” (we will be more precise below). Note that therefore

$$\mathbf{B}_0 \cdot \mathbf{A} = \mathbf{1} - \mathbf{R} \quad (2.5.6)$$

Next consider the following formal manipulation:

$$\begin{aligned} \mathbf{A}^{-1} &= \mathbf{A}^{-1} \cdot (\mathbf{B}_0^{-1} \cdot \mathbf{B}_0) = (\mathbf{A}^{-1} \cdot \mathbf{B}_0^{-1}) \cdot \mathbf{B}_0 = (\mathbf{B}_0 \cdot \mathbf{A})^{-1} \cdot \mathbf{B}_0 \\ &= (\mathbf{1} - \mathbf{R})^{-1} \cdot \mathbf{B}_0 = (\mathbf{1} + \mathbf{R} + \mathbf{R}^2 + \mathbf{R}^3 + \dots) \cdot \mathbf{B}_0 \end{aligned} \quad (2.5.7)$$

We can define the  $n$ th partial sum of the last expression by

$$\mathbf{B}_n \equiv (\mathbf{1} + \mathbf{R} + \dots + \mathbf{R}^n) \cdot \mathbf{B}_0 \quad (2.5.8)$$

so that  $\mathbf{B}_\infty \rightarrow \mathbf{A}^{-1}$ , if the limit exists.

It now is straightforward to verify that equation (2.5.8) satisfies some interesting recurrence relations. As regards solving  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ , where  $\mathbf{x}$  and  $\mathbf{b}$  are vectors, define

$$\mathbf{x}_n \equiv \mathbf{B}_n \cdot \mathbf{b} \quad (2.5.9)$$

Then it is easy to show that

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{B}_0 \cdot (\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_n) \quad (2.5.10)$$

This is immediately recognizable as equation (2.5.4), with  $-\delta\mathbf{x} = \mathbf{x}_{n+1} - \mathbf{x}_n$ , and with  $\mathbf{B}_0$  taking the role of  $\mathbf{A}^{-1}$ . We see, therefore, that equation (2.5.4) does not require that the *LU* decomposition of  $\mathbf{A}$  be exact, but only that the implied residual  $\mathbf{R}$  be small. In rough terms,

if the residual is smaller than the square root of your computer's roundoff error, then after one application of equation (2.5.10) (that is, going from  $\mathbf{x}_0 \equiv \mathbf{B}_0 \cdot \mathbf{b}$  to  $\mathbf{x}_1$ ) the first neglected term, of order  $\mathbf{R}^2$ , will be smaller than the roundoff error. Equation (2.5.10), like equation (2.5.4), moreover, can be applied more than once, since it uses only  $\mathbf{B}_0$ , and not any of the higher  $\mathbf{B}$ 's.

A much more surprising recurrence that follows from equation (2.5.8) is one that more than *doubles* the order  $n$  at each stage:

$$\mathbf{B}_{2n+1} = 2\mathbf{B}_n - \mathbf{B}_n \cdot \mathbf{A} \cdot \mathbf{B}_n \quad n = 0, 1, 3, 7, \dots \quad (2.5.11)$$

Repeated application of equation (2.5.11), from a suitable starting matrix  $\mathbf{B}_0$ , converges *quadratically* to the unknown inverse matrix  $\mathbf{A}^{-1}$  (see §9.4 for the definition of "quadratically"). Equation (2.5.11) goes by various names, including *Schultz's Method* and *Hotelling's Method*; see Pan and Reif [2] for references. In fact, equation (2.5.11) is simply the iterative Newton-Raphson method of root finding (§9.4) applied to matrix inversion.

Before you get too excited about equation (2.5.11), however, you should notice that it involves two full matrix multiplications at each iteration. Each matrix multiplication involves  $N^3$  adds and multiplies. But we already saw in §2.1 – §2.3 that direct inversion of  $\mathbf{A}$  requires only  $N^3$  adds and  $N^3$  multiplies *in toto*. Equation (2.5.11) is therefore practical only when special circumstances allow it to be evaluated much more rapidly than is the case for general matrices. We will meet such circumstances later, in §13.10.

In the spirit of delayed gratification, let us nevertheless pursue the two related issues: When does the series in equation (2.5.7) converge; and what is a suitable initial guess  $\mathbf{B}_0$  (if, for example, an initial  $LU$  decomposition is not feasible)?

We can define the norm of a matrix as the largest amplification of length that it is able to induce on a vector,

$$\|\mathbf{R}\| \equiv \max_{\mathbf{v} \neq 0} \frac{|\mathbf{R} \cdot \mathbf{v}|}{|\mathbf{v}|} \quad (2.5.12)$$

If we let equation (2.5.7) act on some arbitrary right-hand side  $\mathbf{b}$ , as one wants a matrix inverse to do, it is obvious that a sufficient condition for convergence is

$$\|\mathbf{R}\| < 1 \quad (2.5.13)$$

Pan and Reif [2] point out that a suitable initial guess for  $\mathbf{B}_0$  is any sufficiently small constant  $\epsilon$  times the matrix transpose of  $\mathbf{A}$ , that is,

$$\mathbf{B}_0 = \epsilon \mathbf{A}^T \quad \text{or} \quad \mathbf{R} = \mathbf{1} - \epsilon \mathbf{A}^T \cdot \mathbf{A} \quad (2.5.14)$$

To see why this is so involves concepts from Chapter 11; we give here only the briefest sketch:  $\mathbf{A}^T \cdot \mathbf{A}$  is a symmetric, positive-definite matrix, so it has real, positive eigenvalues. In its diagonal representation,  $\mathbf{R}$  takes the form

$$\mathbf{R} = \text{diag}(1 - \epsilon \lambda_0, 1 - \epsilon \lambda_1, \dots, 1 - \epsilon \lambda_{N-1}) \quad (2.5.15)$$

where all the  $\lambda_i$ 's are positive. Evidently any  $\epsilon$  satisfying  $0 < \epsilon < 2/(\max_i \lambda_i)$  will give  $\|\mathbf{R}\| < 1$ . It is not difficult to show that the optimal choice for  $\epsilon$ , giving the most rapid convergence for equation (2.5.11), is

$$\epsilon = 2 / (\max_i \lambda_i + \min_i \lambda_i) \quad (2.5.16)$$

Rarely does one know the eigenvalues of  $\mathbf{A}^T \cdot \mathbf{A}$  in equation (2.5.16). Pan and Reif derive several interesting bounds, which are computable directly from  $\mathbf{A}$ . The following choices guarantee the convergence of  $\mathbf{B}_n$  as  $n \rightarrow \infty$ :

$$\epsilon \leq 1 / \sum_{j,k} a_{jk}^2 \quad \text{or} \quad \epsilon \leq 1 / \left( \max_i \sum_j |a_{ij}| \times \max_j \sum_i |a_{ij}| \right) \quad (2.5.17)$$

The latter expression is truly a remarkable formula, which Pan and Reif derive by noting that the vector norm in equation (2.5.12) need not be the usual  $L_2$  norm, but can instead be either the  $L_\infty$  (max) norm, or the  $L_1$  (absolute value) norm. See their work for details.

Another approach, with which we have had some success, is to estimate the largest eigenvalue statistically, by calculating  $s_i \equiv |\mathbf{A} \cdot \mathbf{v}_i|^2$  for several unit vector  $\mathbf{v}_i$ 's with randomly chosen directions in  $N$ -space. The largest eigenvalue  $\lambda$  can then be bounded by the maximum of  $2 \max s_i$  and  $2N\text{Var}(s_i)/\mu(s_i)$ , where  $\text{Var}$  and  $\mu$  denote the sample variance and mean, respectively.

#### CITED REFERENCES AND FURTHER READING:

- Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.3.4, p. 55.
- Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §3.5.3.
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §5.5.6, p. 183.
- Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 13.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.5, p. 437.
- Higham, N.J. 1997, “Iterative Refinement for Linear Systems and LAPACK,” *IMA Journal of Numerical Analysis*, vol. 17, pp. 495–509.[1]
- Pan, V., and Reif, J. 1985, “Efficient Parallel Solution of Linear Systems,” in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* (New York: Association for Computing Machinery).[2]

## 2.6 Singular Value Decomposition

There exists a very powerful set of techniques for dealing with sets of equations or matrices that are either singular or else numerically very close to singular. In many cases where Gaussian elimination and  $LU$  decomposition fail to give satisfactory results, this set of techniques, known as *singular value decomposition*, or *SVD*, will diagnose for you precisely what the problem is. In some cases, SVD will not only diagnose the problem, it will also solve it, in the sense of giving you a useful numerical answer, although, as we shall see, not necessarily “the” answer that you thought you should get.

SVD is also the method of choice for solving most *linear least-squares* problems. We will outline the relevant theory in this section, but defer detailed discussion of the use of SVD in this application to Chapter 15, whose subject is the **parametric modeling of data**.

SVD methods are based on the following theorem of linear algebra, whose proof is beyond our scope: Any  $M \times N$  matrix  $\mathbf{A}$  can be written as the product of an  $M \times N$  column-orthogonal matrix  $\mathbf{U}$ , an  $N \times N$  diagonal matrix  $\mathbf{W}$  with positive or zero elements (the *singular values*), and the transpose of an  $N \times N$  orthogonal matrix  $\mathbf{V}$ . The various shapes of these matrices are clearer when shown as tableaus. If  $M > N$  (which corresponds to the *overdetermined* situation of more equations than

unknowns), the decomposition looks like this:

$$\begin{pmatrix} \mathbf{A} \end{pmatrix} = \begin{pmatrix} \mathbf{U} \end{pmatrix} \cdot \begin{pmatrix} w_0 & & & \\ & w_1 & & \\ & & \dots & \\ & & & w_{N-1} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{V}^T \end{pmatrix} \quad (2.6.1)$$

If  $M < N$  (the *undetermined* situation of fewer equations than unknowns), it looks like this:

$$\begin{pmatrix} \mathbf{A} \end{pmatrix} = \begin{pmatrix} \mathbf{U} \end{pmatrix} \cdot \begin{pmatrix} w_0 & & & \\ & w_1 & & \\ & & \dots & \\ & & & w_{N-1} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{V}^T \end{pmatrix} \quad (2.6.2)$$

The matrix  $\mathbf{V}$  is orthogonal in the sense that its columns are orthonormal,

$$\sum_{j=0}^{N-1} V_{jk} V_{jn} = \delta_{kn} \quad \begin{matrix} 0 \leq k \leq N-1 \\ 0 \leq n \leq N-1 \end{matrix} \quad (2.6.3)$$

that is,  $\mathbf{V}^T \cdot \mathbf{V} = \mathbf{1}$ . Since  $\mathbf{V}$  is square, it is also row-orthonormal,  $\mathbf{V} \cdot \mathbf{V}^T = \mathbf{1}$ . When  $M \geq N$ , the matrix  $\mathbf{U}$  is also column-orthogonal,

$$\sum_{i=0}^{M-1} U_{ik} U_{in} = \delta_{kn} \quad \begin{matrix} 0 \leq k \leq N-1 \\ 0 \leq n \leq N-1 \end{matrix} \quad (2.6.4)$$

that is,  $\mathbf{U}^T \cdot \mathbf{U} = \mathbf{1}$ . In the case  $M < N$ , however, two things happen: (i) The singular values  $w_j$  for  $j = M, \dots, N-1$  are all zero, and (ii) the corresponding columns of  $\mathbf{U}$  are also zero. Equation (2.6.4) then holds only for  $k, n \leq M-1$ .

The decomposition (2.6.1) or (2.6.2) can always be done, no matter how singular the matrix is, and it is “almost” unique. That is to say, it is unique up to (i) making the same permutation of the columns of  $\mathbf{U}$ , elements of  $\mathbf{W}$ , and columns of  $\mathbf{V}$  (or rows of  $\mathbf{V}^T$ ); or (ii) performing an orthogonal rotation on any set of columns of  $\mathbf{U}$  and  $\mathbf{V}$  whose corresponding elements of  $\mathbf{W}$  happen to be exactly equal. (A special case is multiplying any column of  $\mathbf{U}$ , and the corresponding column of  $\mathbf{V}$  by  $-1$ .) A consequence of the permutation freedom is that for the case  $M < N$ , a numerical algorithm for the decomposition need not return zero  $w_j$ ’s in the canonical positions  $j = M, \dots, N-1$ ; the  $N-M$  zero singular values can be scattered among all positions  $j = 0, 1, \dots, N-1$ , and one needs to perform a sort to get the canonical order. In any case, it is conventional to sort *all* the singular values into descending order.

A Webnote [1] gives the details of the routine that actually performs SVD on an arbitrary matrix  $\mathbf{A}$ , yielding  $\mathbf{U}$ ,  $\mathbf{W}$ , and  $\mathbf{V}$ . The routine is based on a routine

by Forsythe et al. [2], which is in turn based on the original routine of Golub and Reinsch, found, in various forms, in [4-6] and elsewhere. These references include extensive discussion of the algorithm used. As much as we dislike the use of black-box routines, we need to ask you to accept this one, since it would take us too far afield to cover its necessary background material here. The algorithm is very stable, and it is very unusual for it ever to misbehave. Most of the concepts that enter the algorithm (Householder reduction to bidiagonal form, diagonalization by  $QR$  procedure with shifts) will be discussed further in Chapter 11.

As we did for  $LU$  decomposition, we encapsulate the singular value decomposition and also the methods that depend on it into an object,  $\text{SVD}$ . We give its declaration here. The rest of this section will give the details on how to use it.

```
struct SVD {
    Object for singular value decomposition of a matrix A, and related functions.
    Int m,n;                                     The matrices U and V.
    MatDoub u,v;                               The diagonal matrix W.
    VecDoub w;
    Doub eps, tsh;
    SVD(MatDoub_I &a) : m(a.nrows()), n(a.ncols()), u(a), v(n,n), w(n) {
        Constructor. The single argument is A. The SVD computation is done by decompose, and
        the results are sorted by reorder.
        eps = numeric_limits<Doub>::epsilon();
        decompose();
        reorder();
        tsh = 0.5*sqrt(m+n+1.)*w[0]*eps;      Default threshold for nonzero singular
        values.
    }

    void solve(VecDoub_I &b, VecDoub_O &x, Doub thresh);
    void solve(MatDoub_I &b, MatDoub_O &x, Doub thresh);
    Solve with (apply the pseudoinverse to) one or more right-hand sides.

    Int rank(Doub thresh);                         Quantities associated with the range and
    Int nullity(Doub thresh);                      nullspace of A.
    MatDoub range(Doub thresh);
    MatDoub nullspace(Doub thresh);

    Doub inv_condition() {                          Return reciprocal of the condition num-
        return (w[0] <= 0. || w[n-1] <= 0.) ? 0. : w[n-1]/w[0];   ber of A.
    }

    void decompose();                            Functions used by the constructor.
    void reorder();
    Doub pythag(const Doub a, const Doub b);
};
```

## 2.6.1 Range, Nullspace, and All That

Consider the familiar set of simultaneous equations

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \quad (2.6.5)$$

where  $\mathbf{A}$  is an  $M \times N$  matrix, and  $\mathbf{x}$  and  $\mathbf{b}$  are vectors of dimension  $N$  and  $M$  respectively. Equation (2.6.5) defines  $\mathbf{A}$  as a linear mapping from an  $N$ -dimensional vector space to (generally) an  $M$ -dimensional one. But the map *might* be able to reach only a lesser-dimensional subspace of the full  $M$ -dimensional one. That subspace is called the *range* of  $\mathbf{A}$ . The dimension of the range is called the *rank* of  $\mathbf{A}$ . The rank

of  $\mathbf{A}$  is equal to its number of linearly independent columns, and also (perhaps less obviously) to its number of linearly independent rows. If  $\mathbf{A}$  is not identically zero, its rank is at least 1, and at most  $\min(M, N)$ .

Sometimes there are nonzero vectors  $\mathbf{x}$  that are mapped to zero by  $\mathbf{A}$ , that is,  $\mathbf{A} \cdot \mathbf{x} = 0$ . The space of such vectors (a subspace of the  $N$ -dimensional space that  $\mathbf{x}$  lives in) is called the *nullspace* of  $\mathbf{A}$ , and its dimension is called  $\mathbf{A}$ 's *nullity*. The nullity can have any value from zero to  $N$ . The [rank-nullity theorem](#) states that, for any  $\mathbf{A}$ , the rank plus the nullity is  $N$ , the number of columns.

An important special case is  $M = N$ , so the  $\mathbf{A}$  is square,  $N \times N$ . If the rank of  $\mathbf{A}$  is  $N$ , its maximum possible value, then  $\mathbf{A}$  is nonsingular and invertible:  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$  has a unique solution for any  $\mathbf{b}$ , and only the zero vector is mapped to zero. This is a case where *LU* decomposition (§2.3) is the preferred solution method for  $\mathbf{x}$ . However, if  $\mathbf{A}$  has rank less than  $N$  (i.e., has nullity greater than zero), then two things happen: (i) most right-hand side vectors  $\mathbf{b}$  yield no solution, but (ii) some have multiple solutions (in fact a whole subspace of them). We consider this situation further, below.

What has all this to do with singular value decomposition? SVD explicitly constructs orthonormal bases for the nullspace and range of a matrix! Specifically, the columns of  $\mathbf{U}$  whose same-numbered elements  $w_j$  are *nonzero* are an orthonormal set of basis vectors that span the range; the columns of  $\mathbf{V}$  whose same-numbered elements  $w_j$  are *zero* are an orthonormal basis for the nullspace. Our SVD object has methods that return the rank or nullity (integers), and also the range and nullspace, each of these packaged as a matrix whose columns form an orthonormal basis for the respective subspace.

#### svd.h

```
Int SVD::rank(Doub thresh = -1.) {
```

Return the rank of  $\mathbf{A}$ , after zeroing any singular values smaller than  $\text{thresh}$ . If  $\text{thresh}$  is negative, a default value based on estimated roundoff is used.

```
    Int j,nr=0;
    tsh = (thresh >= 0. ? thresh : 0.5*sqrt(m+n+1.)*w[0]*eps);
    for (j=0;j<n;j++) if (w[j] > tsh) nr++;
    return nr;
}
```

```
Int SVD::nullity(Doub thresh = -1.) {
```

Return the nullity of  $\mathbf{A}$ , after zeroing any singular values smaller than  $\text{thresh}$ . Default value as above.

```
    Int j,nn=0;
    tsh = (thresh >= 0. ? thresh : 0.5*sqrt(m+n+1.)*w[0]*eps);
    for (j=0;j<n;j++) if (w[j] <= tsh) nn++;
    return nn;
}
```

```
MatDoub SVD::range(Doub thresh = -1.){
```

Give an orthonormal basis for the range of  $\mathbf{A}$  as the columns of a returned matrix.  $\text{thresh}$  as above.

```
    Int i,j,nr=0;
    MatDoub rnge(m,rank(thresh));
    for (j=0;j<n;j++) {
        if (w[j] > tsh) {
            for (i=0;i<m;i++) rnge[i][nr] = u[i][j];
            nr++;
        }
    }
    return rnge;
}
```

---

```
MatDoub SVD::nullspace(Doub thresh = -1.){
```

Give an orthonormal basis for the nullspace of  $\mathbf{A}$  as the columns of a returned matrix. `thresh` as above.

```
    Int j,jj,nn=0;
    MatDoub nullsp(n,nullity(thresh));
    for (j=0;j<n;j++) {
        if (w[j] <= tsh) {
            for (jj=0;jj<n;jj++) nullsp[jj][nn] = v[jj][j];
            nn++;
        }
    }
    return nullsp;
}
```

The meaning of the optional parameter `thresh` is discussed below.

## 2.6.2 SVD of a Square Matrix

We return to the case of a square  $N \times N$  matrix  $\mathbf{A}$ .  $\mathbf{U}$ ,  $\mathbf{V}$ , and  $\mathbf{W}$  are also square matrices of the same size. Their inverses are also trivial to compute:  $\mathbf{U}$  and  $\mathbf{V}$  are orthogonal, so their inverses are equal to their transposes;  $\mathbf{W}$  is diagonal, so its inverse is the diagonal matrix whose elements are the reciprocals of the elements  $w_j$ . From (2.6.1) it now follows immediately that the inverse of  $\mathbf{A}$  is

$$\mathbf{A}^{-1} = \mathbf{V} \cdot [\text{diag}(1/w_j)] \cdot \mathbf{U}^T \quad (2.6.6)$$

The only thing that can go wrong with this construction is for one of the  $w_j$ 's to be zero, or (numerically) for it to be so small that its value is dominated by roundoff error and therefore unknowable. If more than one of the  $w_j$ 's has this problem, then the matrix is even more singular. So, first of all, SVD gives you a clear diagnosis of the situation.

Formally, the *condition number* of a matrix is defined as the ratio of the largest (in magnitude) of the  $w_j$ 's to the smallest of the  $w_j$ 's. A matrix is singular if its condition number is infinite, and it is *ill-conditioned* if its condition number is too large, that is, if its reciprocal approaches the machine's floating-point precision (for example, less than about  $10^{-15}$  for values of type `double`). A function returning the condition number (or, rather, its reciprocal, to avoid overflow) is implemented in SVD.

Now let's have another look at solving the set of simultaneous linear equations (2.6.5) in the case that  $\mathbf{A}$  is singular. We already saw that the set of *homogeneous* equations, where  $\mathbf{b} = 0$ , is solved immediately by SVD. The solution is any linear combination of the columns returned by the `nullspace` method above.

When the vector  $\mathbf{b}$  on the right-hand side is not zero, the important question is whether it lies in the range of  $\mathbf{A}$  or not. If it does, then the singular set of equations *does* have a solution  $\mathbf{x}$ ; in fact it has more than one solution, since any vector in the nullspace (any column of  $\mathbf{V}$  with a corresponding zero  $w_j$ ) can be added to  $\mathbf{x}$  in any linear combination.

If we want to single out one particular member of this solution set of vectors as a representative, we might want to pick the one with the smallest length  $|\mathbf{x}|^2$ . Here is how to find that vector using SVD: Simply *replace*  $1/w_j$  by zero if  $w_j = 0$ . (It is not very often that one gets to set  $\infty = 0$ !) Then compute, working from right to

left,

$$\mathbf{x} = \mathbf{V} \cdot [\text{diag}(1/w_j)] \cdot (\mathbf{U}^T \cdot \mathbf{b}) \quad (2.6.7)$$

This will be the solution vector of smallest length; the columns of  $\mathbf{V}$  that are in the nullspace complete the specification of the solution set.

Proof: Consider  $|\mathbf{x} + \mathbf{x}'|$ , where  $\mathbf{x}'$  lies in the nullspace. Then, if  $\mathbf{W}^{-1}$  denotes the modified inverse of  $\mathbf{W}$  with some elements zeroed,

$$\begin{aligned} |\mathbf{x} + \mathbf{x}'| &= \left| \mathbf{V} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{x}' \right| \\ &= \left| \mathbf{V} \cdot (\mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}') \right| \\ &= \left| \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}' \right| \end{aligned} \quad (2.6.8)$$

Here the first equality follows from (2.6.7), and the second and third from the orthonormality of  $\mathbf{V}$ . If you now examine the two terms that make up the sum on the right-hand side, you will see that the first one has nonzero  $j$  components only where  $w_j \neq 0$ , while the second one, since  $\mathbf{x}'$  is in the nullspace, has nonzero  $j$  components only where  $w_j = 0$ . Therefore the minimum length obtains for  $\mathbf{x}' = 0$ , q.e.d.

If  $\mathbf{b}$  is not in the range of the singular matrix  $\mathbf{A}$ , then the set of equations (2.6.5) has no solution. But here is some good news: If  $\mathbf{b}$  is not in the range of  $\mathbf{A}$ , then equation (2.6.7) can still be used to construct a “solution” vector  $\mathbf{x}$ . This vector  $\mathbf{x}$  will not exactly solve  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ . But, among all possible vectors  $\mathbf{x}$ , it will do the closest possible job in the least-squares sense. In other words, (2.6.7) finds

$$\mathbf{x} \quad \text{which minimizes} \quad r \equiv |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}| \quad (2.6.9)$$

The number  $r$  is called the *residual* of the solution.

The proof is similar to (2.6.8): Suppose we modify  $\mathbf{x}$  by adding some arbitrary  $\mathbf{x}'$ . Then  $\mathbf{A} \cdot \mathbf{x} - \mathbf{b}$  is modified by adding some  $\mathbf{b}' \equiv \mathbf{A} \cdot \mathbf{x}'$ . Obviously  $\mathbf{b}'$  is in the range of  $\mathbf{A}$ . We then have

$$\begin{aligned} |\mathbf{A} \cdot \mathbf{x} - \mathbf{b} + \mathbf{b}'| &= \left| (\mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V}^T) \cdot (\mathbf{V} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b}) - \mathbf{b} + \mathbf{b}' \right| \\ &= \left| (\mathbf{U} \cdot \mathbf{W} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T - 1) \cdot \mathbf{b} + \mathbf{b}' \right| \\ &= \left| \mathbf{U} \cdot [(\mathbf{W} \cdot \mathbf{W}^{-1} - 1) \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{U}^T \cdot \mathbf{b}'] \right| \\ &= \left| (\mathbf{W} \cdot \mathbf{W}^{-1} - 1) \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{U}^T \cdot \mathbf{b}' \right| \end{aligned} \quad (2.6.10)$$

Now,  $(\mathbf{W} \cdot \mathbf{W}^{-1} - 1)$  is a diagonal matrix that has nonzero  $j$  components only for  $w_j = 0$ , while  $\mathbf{U}^T \mathbf{b}'$  has nonzero  $j$  components only for  $w_j \neq 0$ , since  $\mathbf{b}'$  lies in the range of  $\mathbf{A}$ . Therefore the minimum obtains for  $\mathbf{b}' = 0$ , q.e.d.

Equation (2.6.7), which is also equation (2.6.6) applied associatively to  $\mathbf{b}$ , is thus very general. If no  $w_j$ 's are zero, it solves a nonsingular system of linear equations. If some  $w_j$ 's are zero, and their reciprocals are made zero, then it gives a “best” solution, either the one of shortest length among many, or the one of minimum residual when no exact solution exists. Equation (2.6.6), with the singular  $1/w_j$ 's zeroized, is called the *Moore-Penrose inverse* or *pseudoinverse* of  $\mathbf{A}$ .

Equation (2.6.7) is implemented in the SVD object as the method `solve`. (As in `LUDcmp`, we also include an overloaded form that solves for multiple right-hand sides simultaneously.) The argument `thresh` inputs a value below which  $w_j$ 's are to be considered as being zero; if you omit this argument, or set it to a negative value, then the program uses a default value based on expected roundoff error.

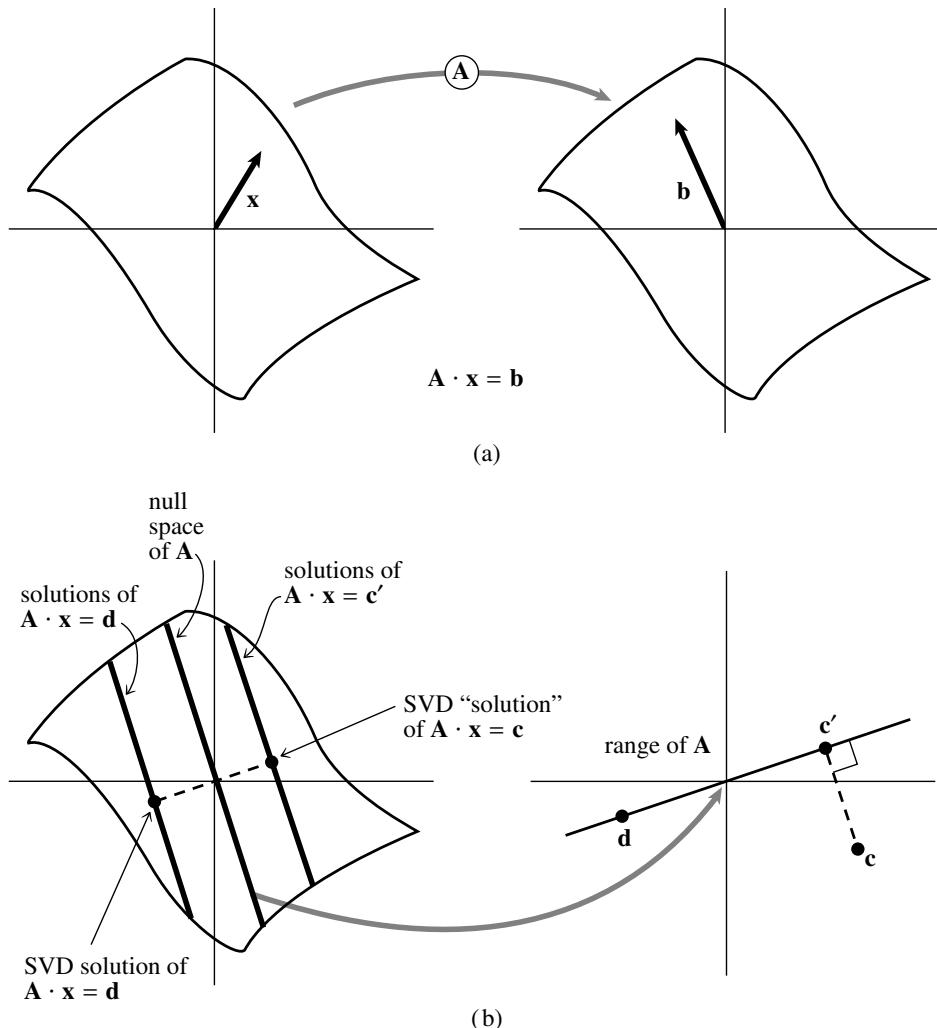
```
void SVD::solve(VecDoub_I &b, VecDoub_O &x, Doub thresh = -1.) {
    Solve  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$  for a vector  $\mathbf{x}$  using the pseudoinverse of  $\mathbf{A}$  as obtained by SVD. If positive,
    thresh is the threshold value below which singular values are considered as zero. If thresh is
    negative, a default based on expected roundoff error is used.
    Int i,j,jj;
    Doub s;
    if (b.size() != m || x.size() != n) throw("SVD::solve bad sizes");
    VecDoub tmp(n);
    tsh = (thresh >= 0. ? thresh : 0.5*sqrt(m+n+1.)*w[0]*eps);
    for (j=0;j<n;j++) {           Calculate  $U^T B$ .
        s=0.0;
        if (w[j] > tsh) {         Nonzero result only if  $w_j$  is nonzero.
            for (i=0;i<m;i++) s += u[i][j]*b[i];
            s /= w[j];             This is the divide by  $w_j$ .
        }
        tmp[j]=s;
    }
    for (j=0;j<n;j++) {           Matrix multiply by  $V$  to get answer.
        s=0.0;
        for (jj=0;jj<n;jj++) s += v[j][jj]*tmp[jj];
        x[j]=s;
    }
}

void SVD::solve(MatDoub_I &b, MatDoub_O &x, Doub thresh = -1.)
    Solves  $m$  sets of  $n$  equations  $\mathbf{A} \cdot \mathbf{X} = \mathbf{B}$  using the pseudoinverse of  $\mathbf{A}$ . The right-hand sides are
    input as b[0..n-1][0..m-1], while x[0..n-1][0..m-1] returns the solutions. thresh as above.
{
    int i,j,m=b.ncols();
    if (b.nrows() != n || x.nrows() != n || b.ncols() != x.ncols())
        throw("SVD::solve bad sizes");
    VecDoub xx(n);
    for (j=0;j<m;j++) {           Copy and solve each column in turn.
        for (i=0;i<n;i++) xx[i] = b[i][j];
        solve(xx,xx,thresh);
        for (i=0;i<n;i++) x[i][j] = xx[i];
    }
}
```

svd.h

Figure 2.6.1 summarizes the situation for the SVD of square matrices.

There are cases in which you may want to set the value of `thresh` to larger than its default. (You can retrieve the default as the member value `tsh`.) In the discussion since equation (2.6.5), we have been pretending that a matrix either is singular or else isn't. Numerically, however, the more common situation is that some of the  $w_j$ 's are very small but nonzero, so that the matrix is ill-conditioned. In that case, the direct solution methods of  $LU$  decomposition or Gaussian elimination may actually give a formal solution to the set of equations (that is, a zero pivot may not be encountered); but the solution vector may have wildly large components whose algebraic cancellation, when multiplying by the matrix  $\mathbf{A}$ , may give a very poor approximation to the right-hand vector  $\mathbf{b}$ . In such cases, the solution vector  $\mathbf{x}$  obtained by *zeroing* the small  $w_j$ 's and then using equation (2.6.7) is very often better (in the sense of the residual  $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$  being smaller) than *both* the direct-method



**Figure 2.6.1.** (a) A nonsingular matrix  $\mathbf{A}$  maps a vector space into one of the same dimension. The vector  $\mathbf{x}$  is mapped into  $\mathbf{b}$ , so that  $\mathbf{x}$  satisfies the equation  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ . (b) A singular matrix  $\mathbf{A}$  maps a vector space into one of lower dimensionality, here a plane into a line, called the “range” of  $\mathbf{A}$ . The “nullspace” of  $\mathbf{A}$  is mapped to zero. The solutions of  $\mathbf{A} \cdot \mathbf{x} = \mathbf{d}$  consist of any one particular solution plus any vector in the nullspace, here forming a line parallel to the nullspace. Singular value decomposition (SVD) selects the particular solution closest to zero, as shown. The point  $\mathbf{c}$  lies outside of the range of  $\mathbf{A}$ , so  $\mathbf{A} \cdot \mathbf{x} = \mathbf{c}$  has no solution. SVD finds the least-squares best compromise solution, namely a solution of  $\mathbf{A} \cdot \mathbf{x} = \mathbf{c}'$ , as shown.

solution and the SVD solution where the small  $w_j$ 's are left nonzero.

It may seem paradoxical that this can be so, since zeroing a singular value corresponds to throwing away one linear combination of the set of equations that we are trying to solve. The resolution of the paradox is that we are throwing away precisely a combination of equations that is so corrupted by roundoff error as to be at best useless; usually it is worse than useless since it “pulls” the solution vector way off toward infinity along some direction that is almost a nullspace vector. In doing this,

it compounds the roundoff problem and makes the residual  $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$  larger.

You therefore have the opportunity of deciding at what threshold `thresh` to zero the small  $w_j$ 's, based on some idea of what size of computed residual  $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$  is acceptable.

For discussion of how the singular value decomposition of a matrix is related to its eigenvalues and eigenvectors, see §11.0.6.

### 2.6.3 SVD for Fewer Equations than Unknowns

If you have fewer linear equations  $M$  than unknowns  $N$ , then you are not expecting a unique solution. Usually there will be an  $N - M$  dimensional family of solutions (which is the nullity, absent any other degeneracies), but the number could be larger. If you want to find this whole solution space, then SVD can readily do the job: Use `solve` to get one (the shortest) solution, then use `nullspace` to get a set of basis vectors for the nullspace. Your solutions are the former plus any linear combination of the latter.

### 2.6.4 SVD for More Equations than Unknowns

This situation will occur in Chapter 15, when we wish to find the least-squares solution to an overdetermined set of linear equations. In tableau, the equations to be solved are

$$\left( \begin{array}{c} \\ \mathbf{A} \\ \end{array} \right) \cdot \left( \begin{array}{c} \\ \mathbf{x} \\ \end{array} \right) = \left( \begin{array}{c} \\ \mathbf{b} \\ \end{array} \right) \quad (2.6.11)$$

The proofs that we gave above for the square case apply without modification to the case of more equations than unknowns. The least-squares solution vector  $\mathbf{x}$  is given by applying the pseudoinverse (2.6.7), which, with nonsquare matrices, looks like this,

$$\left( \begin{array}{c} \\ \mathbf{x} \\ \end{array} \right) = \left( \begin{array}{c} \\ \mathbf{V} \\ \end{array} \right) \cdot \left( \begin{array}{c} \\ \text{diag}(1/w_j) \\ \end{array} \right) \cdot \left( \begin{array}{c} \\ \mathbf{U}^T \\ \end{array} \right) \cdot \left( \begin{array}{c} \\ \mathbf{b} \\ \end{array} \right) \quad (2.6.12)$$

In general, the matrix  $\mathbf{W}$  will not be singular, and no  $w_j$ 's will need to be set to zero. Occasionally, however, there might be column degeneracies in  $\mathbf{A}$ . In this case you will need to zero some small  $w_j$  values after all. The corresponding column in  $\mathbf{V}$  gives the linear combination of  $\mathbf{x}$ 's that is then ill-determined even by the supposedly overdetermined set.

Sometimes, although you do not need to zero any  $w_j$ 's for *computational* reasons, you may nevertheless want to take note of any that are unusually small: Their corresponding columns in  $\mathbf{V}$  are linear combinations of  $\mathbf{x}$ 's that are insensitive to your data. In fact, you may then wish to zero these  $w_j$ 's, by increasing the value of `thresh`, to reduce the number of free parameters in the fit. These matters are discussed more fully in Chapter 15.

### 2.6.5 Constructing an Orthonormal Basis

Suppose that you have  $N$  vectors in an  $M$ -dimensional vector space, with  $N \leq M$ . Then the  $N$  vectors span some subspace of the full vector space. Often you want to construct an orthonormal set of  $N$  vectors that span the same subspace. The elementary textbook way to do this is by Gram-Schmidt orthogonalization, starting with one vector and then expanding the subspace one dimension at a time. Numerically, however, because of the build-up of roundoff errors, naive Gram-Schmidt orthogonalization is *terrible*.

The right way to construct an orthonormal basis for a subspace is by SVD: Form an  $M \times N$  matrix  $\mathbf{A}$  whose  $N$  columns are your vectors. Construct an SVD object from the matrix. The columns of the matrix  $\mathbf{U}$  are your desired orthonormal basis vectors.

You might also want to check the  $w_j$ 's for zero values. If any occur, then the spanned subspace was not, in fact,  $N$ -dimensional; the columns of  $\mathbf{U}$  corresponding to zero  $w_j$ 's should be discarded from the orthonormal basis set. The method `range` does this.

*QR* factorization, discussed in §2.10, also constructs an orthonormal basis; see [3].

### 2.6.6 Approximation of Matrices

Note that equation (2.6.1) can be rewritten to express any matrix  $A_{ij}$  as a sum of outer products of columns of  $\mathbf{U}$  and rows of  $\mathbf{V}^T$ , with the “weighting factors” being the singular values  $w_j$ ,

$$A_{ij} = \sum_{k=0}^{N-1} w_k U_{ik} V_{jk} \quad (2.6.13)$$

If you ever encounter a situation where *most* of the singular values  $w_j$  of a matrix  $\mathbf{A}$  are very small, then  $\mathbf{A}$  will be well-approximated by only a few terms in the sum (2.6.13). This means that you have to store only a few columns of  $\mathbf{U}$  and  $\mathbf{V}$  (the same  $k$  ones) and you will be able to recover, with good accuracy, the whole matrix.

Note also that it is very efficient to multiply such an approximated matrix by a vector  $\mathbf{x}$ : You just dot  $\mathbf{x}$  with each of the stored columns of  $\mathbf{V}$ , multiply the resulting scalar by the corresponding  $w_k$ , and accumulate that multiple of the corresponding column of  $\mathbf{U}$ . If your matrix is approximated by a small number  $K$  of singular values, then this computation of  $\mathbf{A} \cdot \mathbf{x}$  takes only about  $K(M + N)$  multiplications, instead of  $MN$  for the full matrix.

### 2.6.7 Newer Algorithms

Analogous to the newer methods for eigenvalues of symmetric tridiagonal matrices mentioned in §11.4.4, there are newer methods for SVD. There is a divide-and-conquer algorithm, implemented in LAPACK as dgesdd, which is typically faster by a factor of about 5 for large matrices than the algorithm we give (which is similar to the LAPACK routine dgesvd). Another routine based on the MRRR algorithm (see §11.4.4) promises to be even better, but it is not available in LAPACK as of 2006. It will appear as routine dbdscr.

#### CITED REFERENCES AND FURTHER READING:

- Numerical Recipes Software 2007, “SVD Implementation Code,” *Numerical Recipes Webnote No. 2*, at <http://www.nr.com/webnotes?2> [1]
- Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.[2]
- Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §8.6 and Chapter 12 (SVD). *QR* decomposition is discussed in §5.2.6.[3]
- Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 1995 (Philadelphia: S.I.A.M.), Chapter 18.
- Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer), Chapter I.10 by G.H. Golub and C. Reinsch.[4]
- Anderson, E., et al. 1999, LAPACK User’s Guide, 3rd ed. (Philadelphia: S.I.A.M.). Online with software at 2007+, <http://www.netlib.org/lapack>.[5]
- Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer).
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §6.7.[6]

## 2.7 Sparse Linear Systems

A system of linear equations is called *sparse* if only a relatively small number of its matrix elements  $a_{ij}$  are nonzero. It is wasteful to use general methods of linear algebra on such problems, because most of the  $O(N^3)$  arithmetic operations devoted to solving the set of equations or inverting the matrix involve zero operands. Furthermore, you might wish to work problems so large as to tax your available memory space, and it is wasteful to reserve storage for unfruitful zero elements. Note that there are two distinct (and not always compatible) goals for any sparse matrix method: saving time and/or saving space.

We considered one archetypal sparse form in §2.4, the band-diagonal matrix. In the tridiagonal case, e.g., we saw that it was possible to save both time (order  $N$  instead of  $N^3$ ) and space (order  $N$  instead of  $N^2$ ). The method of solution was not different in principle from the general method of *LU* decomposition; it was just applied cleverly, and with due attention to the bookkeeping of zero elements. Many practical schemes for dealing with sparse problems have this same character. They are fundamentally decomposition schemes, or else elimination schemes akin to Gauss-Jordan, but carefully optimized so as to minimize the number of so-called

*fill-ins*, initially zero elements that must become nonzero during the solution process, and for which storage must be reserved.

Direct methods for solving sparse equations, then, depend crucially on the precise pattern of sparsity of the matrix. Patterns that occur frequently, or that are useful as way stations in the reduction of more general forms, already have special names and special methods of solution. We do not have space here for any detailed review of these. References listed at the end of this section will furnish you with an “in” to the specialized literature, and the following list of buzz words (and Figure 2.7.1) will at least let you hold your own at cocktail parties:

- tridiagonal
- band-diagonal (or banded) with bandwidth  $M$
- band triangular
- block diagonal
- block tridiagonal
- block triangular
- cyclic banded
- singly (or doubly) bordered block diagonal
- singly (or doubly) bordered block triangular
- singly (or doubly) bordered band-diagonal
- singly (or doubly) bordered band triangular
- other (!)

You should also be aware of some of the special sparse forms that occur in the solution of partial differential equations in two or more dimensions. See Chapter 20.

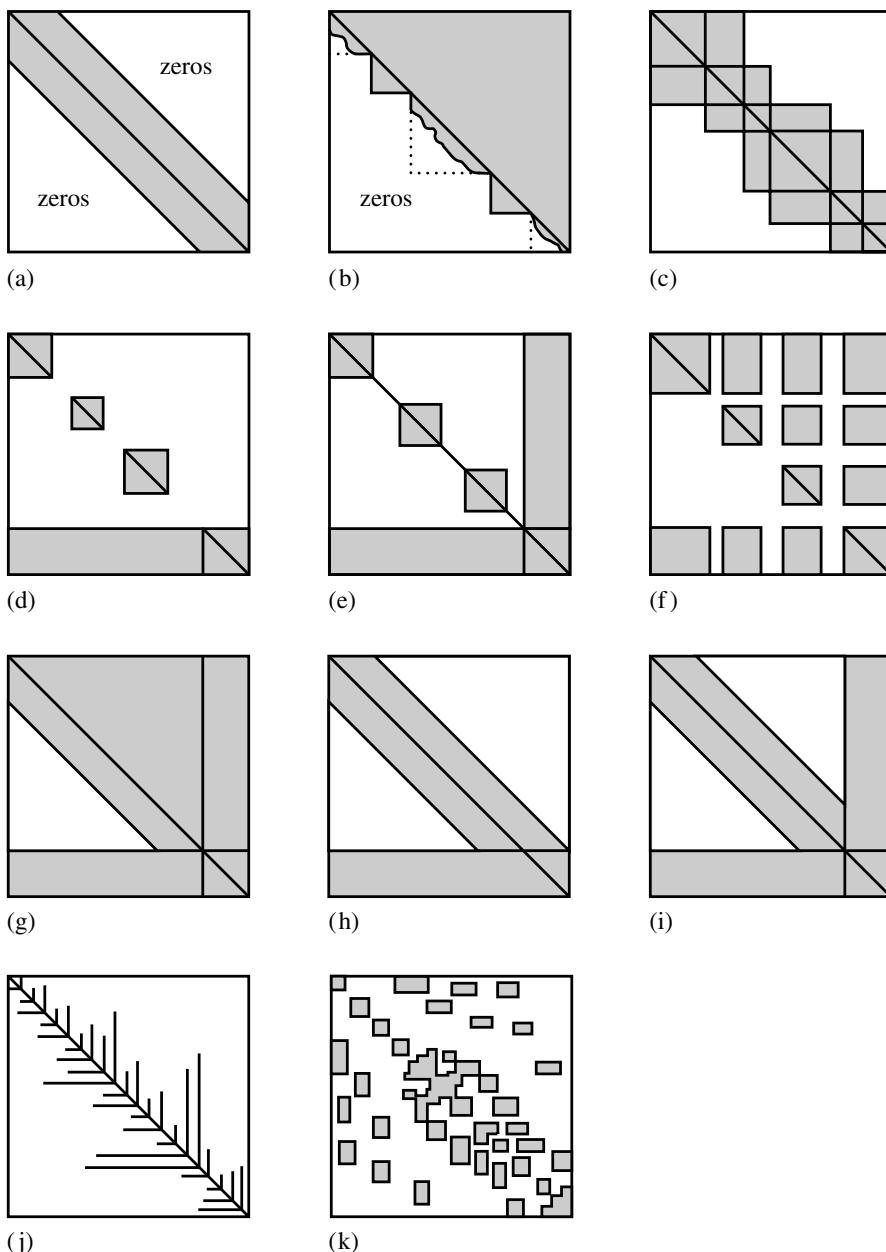
If your particular pattern of sparsity is not a simple one, then you may wish to try an *analyze/factorize/operate* package, which automates the procedure of figuring out how fill-ins are to be minimized. The *analyze* stage is done once only for each pattern of sparsity. The *factorize* stage is done once for each particular matrix that fits the pattern. The *operate* stage is performed once for each right-hand side to be used with the particular matrix. Consult [2,3] for references on this. The NAG library [4] has an *analyze/factorize/operate* capability. A substantial collection of routines for sparse matrix calculation is also available from IMSL [5] as the *Yale Sparse Matrix Package* [6].

You should be aware that the special order of interchanges and eliminations, prescribed by a sparse matrix method so as to minimize fill-ins and arithmetic operations, generally acts to decrease the method’s numerical stability as compared to, e.g., regular *LU* decomposition with pivoting. Scaling your problem so as to make its nonzero matrix elements have comparable magnitudes (if you can do it) will sometimes ameliorate this problem.

In the remainder of this section, we present some concepts that are applicable to some general classes of sparse matrices, and which do not necessarily depend on details of the pattern of sparsity.

### 2.7.1 Sherman-Morrison Formula

Suppose that you have already obtained, by herculean effort, the inverse matrix  $\mathbf{A}^{-1}$  of a square matrix  $\mathbf{A}$ . Now you want to make a “small” change in  $\mathbf{A}$ , for example change one element  $a_{ij}$ , or a few elements, or one row, or one column. Is there any



**Figure 2.7.1.** Some standard forms for sparse matrices. (a) Band-diagonal; (b) block triangular; (c) block tridiagonal; (d) singly bordered block diagonal; (e) doubly bordered block diagonal; (f) singly bordered block triangular; (g) bordered band-triangular; (h) and (i) singly and doubly bordered band-diagonal; (j) and (k) other! (after Tewarson) [1].

way of calculating the corresponding change in  $\mathbf{A}^{-1}$  without repeating your difficult labors? Yes, if your change is of the form

$$\mathbf{A} \rightarrow (\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \quad (2.7.1)$$

for some vectors  $\mathbf{u}$  and  $\mathbf{v}$ . If  $\mathbf{u}$  is a unit vector  $\mathbf{e}_i$ , then (2.7.1) adds the components of  $\mathbf{v}$  to the  $i$ th row. (Recall that  $\mathbf{u} \otimes \mathbf{v}$  is a matrix whose  $i,j$ th element is the product of the  $i$ th component of  $\mathbf{u}$  and the  $j$ th component of  $\mathbf{v}$ .) If  $\mathbf{v}$  is a unit vector  $\mathbf{e}_j$ , then (2.7.1) adds the components of  $\mathbf{u}$  to the  $j$ th column. If both  $\mathbf{u}$  and  $\mathbf{v}$  are proportional to unit vectors  $\mathbf{e}_i$  and  $\mathbf{e}_j$ , respectively, then a term is added only to the element  $a_{ij}$ .

The *Sherman-Morrison* formula gives the inverse  $(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})^{-1}$  and is derived briefly as follows:

$$\begin{aligned} (\mathbf{A} + \mathbf{u} \otimes \mathbf{v})^{-1} &= (\mathbf{1} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v})^{-1} \cdot \mathbf{A}^{-1} \\ &= (\mathbf{1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} - \dots) \cdot \mathbf{A}^{-1} \\ &= \mathbf{A}^{-1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1} (1 - \lambda + \lambda^2 - \dots) \\ &= \mathbf{A}^{-1} - \frac{(\mathbf{A}^{-1} \cdot \mathbf{u}) \otimes (\mathbf{v} \cdot \mathbf{A}^{-1})}{1 + \lambda} \end{aligned} \quad (2.7.2)$$

where

$$\lambda \equiv \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \quad (2.7.3)$$

The second line of (2.7.2) is a formal power series expansion. In the third line, the associativity of outer and inner products is used to factor out the scalars  $\lambda$ .

The use of (2.7.2) is this: Given  $\mathbf{A}^{-1}$  and the vectors  $\mathbf{u}$  and  $\mathbf{v}$ , we need only perform two matrix multiplications and a vector dot product,

$$\mathbf{z} \equiv \mathbf{A}^{-1} \cdot \mathbf{u} \quad \mathbf{w} \equiv (\mathbf{A}^{-1})^T \cdot \mathbf{v} \quad \lambda = \mathbf{v} \cdot \mathbf{z} \quad (2.7.4)$$

to get the desired change in the inverse

$$\mathbf{A}^{-1} \rightarrow \mathbf{A}^{-1} - \frac{\mathbf{z} \otimes \mathbf{w}}{1 + \lambda} \quad (2.7.5)$$

The whole procedure requires only  $3N^2$  multiplies and a like number of adds (an even smaller number if  $\mathbf{u}$  or  $\mathbf{v}$  is a unit vector).

The Sherman-Morrison formula can be directly applied to a class of sparse problems. If you already have a fast way of calculating the inverse of  $\mathbf{A}$  (e.g., a tridiagonal matrix or some other standard sparse form), then (2.7.4) – (2.7.5) allow you to build up to your related but more complicated form, adding for example a row or column at a time. Notice that you can apply the Sherman-Morrison formula more than once successively, using at each stage the most recent update of  $\mathbf{A}^{-1}$  (equation 2.7.5). Of course, if you have to modify *every* row, then you are back to an  $N^3$  method. The constant in front of the  $N^3$  is only a few times worse than the better direct methods, but you have deprived yourself of the stabilizing advantages of pivoting — so be careful.

For some other sparse problems, the Sherman-Morrison formula cannot be directly applied for the simple reason that storage of the whole inverse matrix  $\mathbf{A}^{-1}$  is not feasible. If you want to add only a single correction of the form  $\mathbf{u} \otimes \mathbf{v}$  and solve the linear system

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \cdot \mathbf{x} = \mathbf{b} \quad (2.7.6)$$

then you proceed as follows. Using the fast method that is presumed available for the matrix  $\mathbf{A}$ , solve the two auxiliary problems

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \quad \mathbf{A} \cdot \mathbf{z} = \mathbf{u} \quad (2.7.7)$$

for the vectors  $\mathbf{y}$  and  $\mathbf{z}$ . In terms of these,

$$\mathbf{x} = \mathbf{y} - \left[ \frac{\mathbf{v} \cdot \mathbf{y}}{1 + (\mathbf{v} \cdot \mathbf{z})} \right] \mathbf{z} \quad (2.7.8)$$

as we see by multiplying (2.7.2) on the right by  $\mathbf{b}$ .

## 2.7.2 Cyclic Tridiagonal Systems

So-called *cyclic tridiagonal systems* occur quite frequently and are a good example of how to use the Sherman-Morrison formula in the manner just described. The equations have the form

$$\begin{bmatrix} b_0 & c_0 & 0 & \cdots & & \beta \\ a_1 & b_1 & c_1 & \cdots & & \\ & \cdots & & & & \\ & \cdots & a_{N-2} & b_{N-2} & c_{N-2} & \\ \alpha & \cdots & 0 & a_{N-1} & b_{N-1} & \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_{N-2} \\ x_{N-1} \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ \cdots \\ r_{N-2} \\ r_{N-1} \end{bmatrix} \quad (2.7.9)$$

This is a tridiagonal system, except for the matrix elements  $\alpha$  and  $\beta$  in the corners. Forms like this are typically generated by finite differencing differential equations with periodic boundary conditions (§20.4).

We use the Sherman-Morrison formula, treating the system as tridiagonal plus a correction. In the notation of equation (2.7.6), define vectors  $\mathbf{u}$  and  $\mathbf{v}$  to be

$$\mathbf{u} = \begin{bmatrix} \gamma \\ 0 \\ \vdots \\ 0 \\ \alpha \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ \beta/\gamma \end{bmatrix} \quad (2.7.10)$$

Here  $\gamma$  is arbitrary for the moment. Then the matrix  $\mathbf{A}$  is the tridiagonal part of the matrix in (2.7.9), with two terms modified:

$$b'_0 = b_0 - \gamma, \quad b'_{N-1} = b_{N-1} - \alpha\beta/\gamma \quad (2.7.11)$$

We now solve equations (2.7.7) with the standard tridiagonal algorithm and then get the solution from equation (2.7.8).

The routine `cyclic` below implements this algorithm. We choose the arbitrary parameter  $\gamma = -b_0$  to avoid loss of precision by subtraction in the first of equations (2.7.11). In the unlikely event that this causes loss of precision in the second of these equations, you can make a different choice.

```
void cyclic(VecDoub_I &a, VecDoub_I &b, VecDoub_I &c, const Doub alpha,           tridag.h
           const Doub beta, VecDoub_I &r, VecDoub_O &x)
Solves for a vector x[0..n-1] the "cyclic" set of linear equations given by equation (2.7.9). a,
b, c, and r are input vectors, all dimensioned as [0..n-1], while alpha and beta are the corner
entries in the matrix. The input is not modified.
{
    Int i,n=a.size();
    Doub fact,gamma;
    if (n <= 2) throw("n too small in cyclic");
}
```

```

VecDoub bb(n),u(n),z(n);
gamma = -b[0];
bb[0]=b[0]-gamma;
bb[n-1]=b[n-1]-alpha*beta/gamma;
for (i=1;i<n-1;i++) bb[i]=b[i];
tridag(a,bb,c,r,x);
u[0]=gamma;
u[n-1]=alpha;
for (i=1;i<n-1;i++) u[i]=0.0;
tridag(a,bb,c,u,z);
fact=(x[0]+beta*x[n-1]/gamma)/
(1.0+z[0]+beta*z[n-1]/gamma);
for (i=0;i<n;i++) x[i] -= fact*z[i];
}

```

Avoid subtraction error in forming  $bb[0]$ .  
Set up the diagonal of the modified tridiagonal system.

Solve  $\mathbf{A} \cdot \mathbf{x} = \mathbf{r}$ .  
Set up the vector  $\mathbf{u}$ .

Solve  $\mathbf{A} \cdot \mathbf{z} = \mathbf{u}$ .  
Form  $\mathbf{v} \cdot \mathbf{x} / (1 + \mathbf{v} \cdot \mathbf{z})$ .

Now get the solution vector  $\mathbf{x}$ .

### 2.7.3 Woodbury Formula

If you want to add more than a single correction term, then you cannot use (2.7.8) repeatedly, since without storing a new  $\mathbf{A}^{-1}$  you will not be able to solve the auxiliary problems (2.7.7) efficiently after the first step. Instead, you need the *Woodbury formula*, which is the block-matrix version of the Sherman-Morrison formula,

$$\begin{aligned} & (\mathbf{A} + \mathbf{U} \cdot \mathbf{V}^T)^{-1} \\ &= \mathbf{A}^{-1} - \left[ \mathbf{A}^{-1} \cdot \mathbf{U} \cdot (1 + \mathbf{V}^T \cdot \mathbf{A}^{-1} \cdot \mathbf{U})^{-1} \cdot \mathbf{V}^T \cdot \mathbf{A}^{-1} \right] \end{aligned} \quad (2.7.12)$$

Here  $\mathbf{A}$  is, as usual, an  $N \times N$  matrix, while  $\mathbf{U}$  and  $\mathbf{V}$  are  $N \times P$  matrices with  $P < N$  and usually  $P \ll N$ . The inner piece of the correction term may become clearer if written as the tableau,

$$\left[ \begin{array}{c} \mathbf{U} \end{array} \right] \cdot \left[ \begin{array}{c} 1 + \mathbf{V}^T \cdot \mathbf{A}^{-1} \cdot \mathbf{U} \end{array} \right]^{-1} \cdot \left[ \begin{array}{c} \mathbf{V}^T \end{array} \right] \quad (2.7.13)$$

where you can see that the matrix whose inverse is needed is only  $P \times P$  rather than  $N \times N$ .

The relation between the Woodbury formula and successive applications of the Sherman-Morrison formula is now clarified by noting that, if  $\mathbf{U}$  is the matrix formed by columns out of the  $P$  vectors  $\mathbf{u}_0, \dots, \mathbf{u}_{P-1}$ , and  $\mathbf{V}$  is the matrix formed by columns out of the  $P$  vectors  $\mathbf{v}_0, \dots, \mathbf{v}_{P-1}$ ,

$$\mathbf{U} \equiv \left[ \begin{array}{c} \mathbf{u}_0 \\ \vdots \\ \mathbf{u}_{P-1} \end{array} \right] \quad \mathbf{V} \equiv \left[ \begin{array}{c} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_{P-1} \end{array} \right] \quad (2.7.14)$$

then two ways of expressing the same correction to  $\mathbf{A}$  are

$$\left( \mathbf{A} + \sum_{k=0}^{P-1} \mathbf{u}_k \otimes \mathbf{v}_k \right) = (\mathbf{A} + \mathbf{U} \cdot \mathbf{V}^T) \quad (2.7.15)$$

(Note that the subscripts on  $\mathbf{u}$  and  $\mathbf{v}$  do *not* denote components, but rather distinguish the different column vectors.)

Equation (2.7.15) reveals that, if you have  $\mathbf{A}^{-1}$  in storage, then you can either make the  $P$  corrections in one fell swoop by using (2.7.12) and inverting a  $P \times P$  matrix, or else make them by applying (2.7.5)  $P$  successive times.

If you don't have storage for  $\mathbf{A}^{-1}$ , then you *must* use (2.7.12) in the following way: To solve the linear equation

$$\left( \mathbf{A} + \sum_{k=0}^{P-1} \mathbf{u}_k \otimes \mathbf{v}_k \right) \cdot \mathbf{x} = \mathbf{b} \quad (2.7.16)$$

first solve the  $P$  auxiliary problems

$$\begin{aligned} \mathbf{A} \cdot \mathbf{z}_0 &= \mathbf{u}_0 \\ \mathbf{A} \cdot \mathbf{z}_1 &= \mathbf{u}_1 \\ &\dots \\ \mathbf{A} \cdot \mathbf{z}_{P-1} &= \mathbf{u}_{P-1} \end{aligned} \quad (2.7.17)$$

and construct the matrix  $\mathbf{Z}$  by columns from the  $\mathbf{z}$ 's obtained,

$$\mathbf{Z} \equiv \begin{bmatrix} \mathbf{z}_0 \\ \mathbf{z}_1 \\ \vdots \\ \mathbf{z}_{P-1} \end{bmatrix} \quad (2.7.18)$$

Next, do the  $P \times P$  matrix inversion

$$\mathbf{H} \equiv (\mathbf{1} + \mathbf{V}^T \cdot \mathbf{Z})^{-1} \quad (2.7.19)$$

Finally, solve the one further auxiliary problem

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \quad (2.7.20)$$

In terms of these quantities, the solution is given by

$$\mathbf{x} = \mathbf{y} - \mathbf{Z} \cdot \left[ \mathbf{H} \cdot (\mathbf{V}^T \cdot \mathbf{y}) \right] \quad (2.7.21)$$

## 2.7.4 Inversion by Partitioning

Once in a while, you will encounter a matrix (not even necessarily sparse) that can be inverted efficiently by partitioning. Suppose that the  $N \times N$  matrix  $\mathbf{A}$  is partitioned into

$$\mathbf{A} = \begin{bmatrix} \mathbf{P} & \mathbf{Q} \\ \mathbf{R} & \mathbf{S} \end{bmatrix} \quad (2.7.22)$$

where  $\mathbf{P}$  and  $\mathbf{S}$  are square matrices of size  $p \times p$  and  $s \times s$ , respectively ( $p + s = N$ ). The matrices  $\mathbf{Q}$  and  $\mathbf{R}$  are not necessarily square and have sizes  $p \times s$  and  $s \times p$ , respectively.

If the inverse of  $\mathbf{A}$  is partitioned in the same manner,

$$\mathbf{A}^{-1} = \begin{bmatrix} \tilde{\mathbf{P}} & \tilde{\mathbf{Q}} \\ \tilde{\mathbf{R}} & \tilde{\mathbf{S}} \end{bmatrix} \quad (2.7.23)$$

then  $\tilde{\mathbf{P}}$ ,  $\tilde{\mathbf{Q}}$ ,  $\tilde{\mathbf{R}}$ ,  $\tilde{\mathbf{S}}$ , which have the same sizes as  $\mathbf{P}$ ,  $\mathbf{Q}$ ,  $\mathbf{R}$ ,  $\mathbf{S}$ , respectively, can be found by either the formulas

$$\begin{aligned}\tilde{\mathbf{P}} &= (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \\ \tilde{\mathbf{Q}} &= -(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \cdot (\mathbf{Q} \cdot \mathbf{S}^{-1}) \\ \tilde{\mathbf{R}} &= -(\mathbf{S}^{-1} \cdot \mathbf{R}) \cdot (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \\ \tilde{\mathbf{S}} &= \mathbf{S}^{-1} + (\mathbf{S}^{-1} \cdot \mathbf{R}) \cdot (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \cdot (\mathbf{Q} \cdot \mathbf{S}^{-1})\end{aligned}\quad (2.7.24)$$

or else by the equivalent formulas

$$\begin{aligned}\tilde{\mathbf{P}} &= \mathbf{P}^{-1} + (\mathbf{P}^{-1} \cdot \mathbf{Q}) \cdot (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \cdot (\mathbf{R} \cdot \mathbf{P}^{-1}) \\ \tilde{\mathbf{Q}} &= -(\mathbf{P}^{-1} \cdot \mathbf{Q}) \cdot (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \\ \tilde{\mathbf{R}} &= -(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \cdot (\mathbf{R} \cdot \mathbf{P}^{-1}) \\ \tilde{\mathbf{S}} &= (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1}\end{aligned}\quad (2.7.25)$$

The parentheses in equations (2.7.24) and (2.7.25) highlight repeated factors that you may wish to compute only once. (Of course, by associativity, you can instead do the matrix multiplications in any order you like.) The choice between using equations (2.7.24) and (2.7.25) depends on whether you want  $\tilde{\mathbf{P}}$  or  $\tilde{\mathbf{S}}$  to have the simpler formula; or on whether the repeated expression  $(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1}$  is easier to calculate than the expression  $(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1}$ ; or on the relative sizes of  $\mathbf{P}$  and  $\mathbf{S}$ ; or on whether  $\mathbf{P}^{-1}$  or  $\mathbf{S}^{-1}$  is already known.

Another sometimes useful formula is for the determinant of the partitioned matrix,

$$\det \mathbf{A} = \det \mathbf{P} \det(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q}) = \det \mathbf{S} \det(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R}) \quad (2.7.26)$$

## 2.7.5 Indexed Storage of Sparse Matrices

We have already seen (§2.4) that tri- or band-diagonal matrices can be stored in a compact format that allocates storage only to elements that can be nonzero, plus perhaps a few wasted locations to make the bookkeeping easier. What about more general sparse matrices? When a sparse matrix of dimension  $M \times N$  contains only a few times  $M$  or  $N$  nonzero elements (a typical case), it is surely inefficient — and often physically impossible — to allocate storage for all  $MN$  elements. Even if one did allocate such storage, it would be inefficient or prohibitive in machine time to loop over all of it in search of nonzero elements.

Obviously some kind of indexed storage scheme is required, one that stores only nonzero matrix elements, along with sufficient auxiliary information to determine where an element logically belongs and how the various elements can be looped over in common matrix operations. Unfortunately, there is no one standard scheme in general use. Each scheme has its own pluses and minuses, depending on the application.

Before we look at sparse matrices, let's consider the simpler problem of a *sparse vector*. The obvious data structure is a list of the nonzero values and another list of the corresponding locations:

sparse.h

```
struct NRsparseCol  
Sparse vector data structure.  
{  
    Int nrows;  
    Int nvals;
```

```

VecInt row_ind;           Row indices of nonzeros.
VecDoub val;             Array of nonzero values.

NRsparseCol(Int m, Int nnvals) : nrows(m), nvals(nnvals),
row_ind(nnvals,0),val(nnvals,0.0) {}    Constructor. Initializes vector to zero.

NRsparseCol() : nrows(0),nvals(0),row_ind(),val() {}   Default constructor.

void resize(Int m, Int nnvals) {
    nrows = m;
    nvals = nnvals;
    row_ind.assign(nnvals,0);
    val.assign(nnvals,0.0);
}

};


```

While we think of this as defining a column vector, you can use exactly the same data structure for a row vector — just mentally interchange the meaning of row and column for the variables. For matrices, however, we have to decide ahead of time whether to use row-oriented or column-oriented storage.

One simple scheme is to use a vector of sparse columns:

```

NRvector<NRsparseCol *> a;
for (i=0;i<n;i++) {
    nvals=...
    a[i]=new NRsparseCol(m,nvals);
}

```

Each column is filled with statements like

```

count=0;
for (j=...) {
    a[i]→row_ind[count]=...
    a[i]→val[count]=...
    count++;
}

```

This data structure is good for an algorithm that primarily works with columns of the matrix, but it is not very efficient when one needs to loop over all elements of the matrix.

A good general storage scheme is the *compressed column storage* format. It is sometimes called the Harwell-Boeing format, after the two large organizations that first systematically provided a standard collection of sparse matrices for research purposes. In this scheme, three vectors are used: **val** for the nonzero values as they are traversed column by column, **row\_ind** for the corresponding row indices of each value, and **col\_ptr** for the locations in the other two arrays that start a column. In other words, if **val**[k]=a[i][j], then **row\_ind**[k]=i. The first nonzero in column j is at **col\_ptr**[j]. The last is at **col\_ptr**[j+1]-1. Note that **col\_ptr**[0] is always 0, and by convention we define **col\_ptr**[n] equal to the number of nonzeros. Note also that the dimension of the **col\_ptr** array is  $N + 1$ , not  $N$ . The advantage of this scheme is that it requires storage of only about two times the number of nonzero matrix elements. (Other methods can require as much as three or five times.)

As an example, consider the matrix

$$\begin{bmatrix} 3.0 & 0.0 & 1.0 & 2.0 & 0.0 \\ 0.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 7.0 & 5.0 & 9.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 6.0 & 5.0 \end{bmatrix} \quad (2.7.27)$$

In compressed column storage mode, matrix (2.7.27) is represented by two arrays of length 9 and an array of length 6, as follows

|            |     |     |     |     |     |     |     |     |     |
|------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| index k    | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   |
| val[k]     | 3.0 | 4.0 | 7.0 | 1.0 | 5.0 | 2.0 | 9.0 | 6.0 | 5.0 |
| row_ind[k] | 0   | 1   | 2   | 0   | 2   | 0   | 2   | 4   | 4   |

|            |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|
| index i    | 0 | 1 | 2 | 3 | 4 | 5 |
| col_ptr[i] | 0 | 1 | 3 | 5 | 8 | 9 |

(2.7.28)

Notice that, according to the storage rules, the value of  $N$  (namely 5) is the maximum valid index in `col_ptr`. The value of `col_ptr[5]` is 9, the length of the other two arrays. The elements 1.0 and 5.0 in column number 2, for example, are located in positions  $\text{col\_ptr}[2] \leq k < \text{col\_ptr}[3]$ .

Here is a data structure to handle this storage scheme:

```
sparse.h struct NRsparseMat
Sparse matrix data structure for compressed column storage.
{
    Int nrows;                      Number of rows.
    Int ncols;                      Number of columns.
    Int nnvals;                     Maximum number of nonzeros.
    VecInt col_ptr;                 Pointers to start of columns. Length is ncols+1.
    VecInt row_ind;                Row indices of nonzeros.
    VecDoub val;                   Array of nonzero values.

    NRsparseMat();                  Default constructor.
    NRsparseMat(Int m,Int n,Int nnvals);   Constructor. Initializes vector to zero.
    VecDoub ax(const VecDoub &x) const;   Multiply A by a vector x[0..ncols-1].
    VecDoub atx(const VecDoub &x) const;  Multiply AT by a vector x[0..nrows-1].
    NRsparseMat transpose() const;      Form AT.
};
```

The code for the constructors is standard:

```
sparse.h NRsparseMat::NRsparseMat() : nrows(0),ncols(0),nnvals(0),col_ptr(),
                                         row_ind(),val() {}
NRsparseMat::NRsparseMat(Int m,Int n,Int nnvals) : nrows(m),ncols(n),
                                         nnvals(nnvals),col_ptr(n+1,0),row_ind(nnvals,0),val(nnvals,0.0) {}
```

The single most important use of a matrix in compressed column storage mode is to multiply a vector to its right. Don't implement this by traversing the rows of  $A$ , which is extremely inefficient in this storage mode. Here's the right way to do it:

```
sparse.h VecDoub NRsparseMat::ax(const VecDoub &x) const {
    VecDoub y(nrows,0.0);
    for (Int j=0;j<ncols;j++) {
        for (Int i=col_ptr[j];i<col_ptr[j+1];i++)
            y[row_ind[i]] += val[i]*x[j];
    }
    return y;
}
```

Some inefficiency occurs because of the indirect addressing. While there are other storage modes that minimize this, they have their own drawbacks.

It is also simple to multiply the *transpose* of a matrix by a vector to its right, since we just traverse the columns directly. (Indirect addressing is still required.) Note that the transpose matrix is not actually constructed.

```
VecDoub NRsparseMat::atx(const VecDoub &x) const {           sparse.h
    VecDoub y(ncols);
    for (Int i=0;i<ncols;i++) {
        y[i]=0.0;
        for (Int j=col_ptr[i];j<col_ptr[i+1];j++)
            y[i] += val[j]*x[row_ind[j]];
    }
    return y;
}
```

Because the choice of compressed column storage treats rows and columns quite differently, it is rather an involved operation to construct the transpose of a matrix, given the matrix itself in compressed column storage mode. When the operation cannot be avoided, it is

```
NRsparseMat NRsparseMat::transpose() const {           sparse.h
    Int i,j,k,index,m=nrows,n=ncols;
    NRsparseMat at(n,m,nvals);                         Initialized to zero.
    First find the column lengths for  $\mathbf{A}^T$ , i.e. the row lengths of  $\mathbf{A}$ .
    VecInt count(m,0);                                Temporary counters for each row of  $\mathbf{A}$ .
    for (i=0;i<n;i++)
        for (j=col_ptr[i];j<col_ptr[i+1];j++) {
            k=row_ind[j];
            count[k]++;
        }
    for (j=0;j<m;j++)                                     Now set at.col_ptr. 0th entry stays 0.
        at.col_ptr[j+1]=at.col_ptr[j]+count[j];
    for(j=0;j<m;j++)                                     Reset counters to zero.
        count[j]=0;
    for (i=0;i<n;i++)                                    Main loop.
        for (j=col_ptr[i];j<col_ptr[i+1];j++) {
            k=row_ind[j];
            index=at.col_ptr[k]+count[k];   Element's position in column of  $\mathbf{A}^T$ .
            at.row_ind[index]=i;
            at.val[index]=val[j];
            count[k]++;
        }
    return at;
}
```

The only sparse matrix-matrix multiply routine we give is to form the product  $\mathbf{AD}\mathbf{A}^T$ , where  $\mathbf{D}$  is a diagonal matrix. This particular product is used to form the so-called normal equations in the interior-point method for linear programming (§10.11). We encapsulate the algorithm in its own structure, ADAT:

```
struct ADAT {                                              sparse.h
    const NRsparseMat &a,&at;                           Store references to  $\mathbf{A}$  and  $\mathbf{A}^T$ .
    NRsparseMat *adat;                                  This will hold  $\mathbf{AD}\mathbf{A}^T$ .
    ADAT(const NRsparseMat &A,const NRsparseMat &AT);
    Allocates compressed column storage for  $\mathbf{AD}\mathbf{A}^T$ , where  $\mathbf{A}$  and  $\mathbf{A}^T$  are input in compressed
    column format, and fills in values of col_ptr and row_ind. Each column must be in sorted
    order in input matrices. Matrix is output with each column sorted.
    void updateD(const VecDoub &D);
    Computes  $\mathbf{AD}\mathbf{A}^T$ , where  $\mathbf{D}$  is a diagonal matrix. This function can be called repeatedly
    to update  $\mathbf{AD}\mathbf{A}^T$  for fixed  $\mathbf{A}$ .
    NRsparseMat &ref();
    Returns reference to adat, which holds  $\mathbf{AD}\mathbf{A}^T$ .
    ~ADAT();
};
```

The algorithm proceeds in two steps. First, the nonzero pattern of  $\mathbf{A}\mathbf{A}^T$  is found by a call to the constructor. Since  $\mathbf{D}$  is diagonal,  $\mathbf{A}\mathbf{A}^T$  and  $\mathbf{A}\mathbf{D}\mathbf{A}^T$  have the same nonzero structure. Algorithms using ADAT will typically have both  $\mathbf{A}$  and  $\mathbf{A}^T$  available, so we pass them both to the constructor rather than recompute  $\mathbf{A}^T$  from  $\mathbf{A}$ . The constructor allocates storage and assigns values to `col_ptr` and `row_ind`. The structure of ADAT is returned with columns in sorted order because routines like the AMD ordering algorithm used in §10.11 require it.

```
sparse.h ADAT::ADAT(const NRsparseMat &A,const NRsparseMat &AT) : a(A), at(AT) {
    Int h,i,j,k,l,nvals,m=AT.ncols;
    VecInt done(m);
    for (i=0;i<m;i++) done[i]=-1;
    nvals=0;
    for (j=0;j<m;j++) {
        for (i=AT.col_ptr[j];i<AT.col_ptr[j+1];i++) {
            k=AT.row_ind[i];
            for (l=A.col_ptr[k];l<A.col_ptr[k+1];l++) {
                h=A.row_ind[l];
                if (done[h] != j) {
                    done[h]=j;
                    nvals++;
                }
            }
        }
    }
    adat = new NRsparseMat(m,m,nvals);      Allocate storage for ADAT.
    for (i=0;i<m;i++) done[i]=-1;           Re-initialize.
    nvals=0;
    Second pass: Determine columns of adat. Code is identical to first pass except adat->col_ptr
    and adat->row_ind get assigned at appropriate places.
    for (j=0;j<m;j++) {
        adat->col_ptr[j]=nvals;
        for (i=AT.col_ptr[j];i<AT.col_ptr[j+1];i++) {
            k=AT.row_ind[i];
            for (l=A.col_ptr[k];l<A.col_ptr[k+1];l++) {
                h=A.row_ind[l];
                if (done[h] != j) {
                    done[h]=j;
                    adat->row_ind[nvals]=h;
                    nvals++;
                }
            }
        }
    }
    adat->col_ptr[m]=nvals;                  Set last value.
    for (j=0;j<m;j++) {                     Sort columns
        i=adat->col_ptr[j];
        Int size=adat->col_ptr[j+1]-i;
        if (size > 1) {
            VecInt col(size,&adat->row_ind[i]);
            sort(col);
            for (k=0;k<size;k++)
                adat->row_ind[i+k]=col[k];
        }
    }
}
```

The next routine, `updateD`, actually fills in the values in the `val` array. It can be called repeatedly to update  $\mathbf{A}\mathbf{D}\mathbf{A}^T$  for fixed  $\mathbf{A}$ .

```

void ADAT::updateD(const VecDoub &D) {                                     sparse.h
    Int h,i,j,k,l,m=a.nrows,n=a.ncols;
    VecDoub temp(n),temp2(m,0.0);
    for (i=0;i<m;i++) {                                         Outer loop over columns of  $\mathbf{A}^T$ .
        for (j=at.col_ptr[i];j< at.col_ptr[i+1];j++) {
            k=at.row_ind[j];
            for (l=a.col_ptr[k];l<a.col_ptr[k+1];l++) {       Scale elements of each column with  $\mathbf{D}$  and
                temp[k]=at.val[j]*D[l];                         store in temp.
            }
        for (j=at.col_ptr[i];j<at.col_ptr[i+1];j++) {           Go down column again.
            k=at.row_ind[j];
            for (l=a.col_ptr[k];l<a.col_ptr[k+1];l++) {       Go down column  $k$  in
                h=a.row_ind[l];
                temp2[h] += temp[k]*a.val[l];                  All terms from temp[k] used here.
            }
        }
        for (j=adat->col_ptr[i];j<adat->col_ptr[i+1];j++) {   Store temp2 in column of answer.
            k=adat->row_ind[j];
            adat->val[j]=temp2[k];
            temp2[k]=0.0;                                       Restore temp2.
        }
    }
}

```

The final two functions are simple. The `ref` routine returns a reference to the matrix  $\mathbf{ADAT}^T$  stored in the structure for other routines that may need to work with it. And the destructor releases the storage.

```

NRsparseMat & ADAT::ref() {                                              sparse.h
    return *adat;
}

ADAT::~ADAT() {
    delete adat;
}

```

By the way, if you invoke `ADAT` with *different* matrices  $\mathbf{A}$  and  $\mathbf{B}^T$ , everything will work fine as long as  $\mathbf{A}$  and  $\mathbf{B}$  have the same nonzero pattern.

In *Numerical Recipes* second edition, we gave a related sparse matrix storage mode in which the diagonal of the matrix is stored first, followed by the off-diagonal elements. We now feel that the added complexity of that scheme is not worthwhile for any of the uses in this book. For a discussion of this and other storage schemes, see [7,8]. To see how to work with the diagonal in the compressed column mode, look at the code for `asolve` at the end of this section.

## 2.7.6 Conjugate Gradient Method for a Sparse System

So-called *conjugate gradient methods* provide a quite general means for solving the  $N \times N$  linear system

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.7.29}$$

The attractiveness of these methods for large sparse systems is that they reference  $\mathbf{A}$  only through its multiplication of a vector, or the multiplication of its transpose and a vector. As we have seen, these operations can be very efficient for a properly stored sparse matrix. You, the “owner” of the matrix  $\mathbf{A}$ , can be asked to provide functions that perform these sparse matrix multiplications as efficiently as possible. We, the “grand strategists,” supply an abstract base class, `Linbcg` below, that contains the method for solving the set of linear equations, (2.7.29), using your functions.

The simplest, “ordinary” conjugate gradient algorithm [9-11] solves (2.7.29) only in the case that  $\mathbf{A}$  is symmetric and positive-definite. It is based on the idea of minimizing the function

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \cdot \mathbf{x} \tag{2.7.30}$$

This function is minimized when its gradient

$$\nabla f = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \quad (2.7.31)$$

is zero, which is equivalent to (2.7.29). The minimization is carried out by generating a succession of search directions  $\mathbf{p}_k$  and improved minimizers  $\mathbf{x}_k$ . At each stage a quantity  $\alpha_k$  is found that minimizes  $f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$ , and  $\mathbf{x}_{k+1}$  is set equal to the new point  $\mathbf{x}_k + \alpha_k \mathbf{p}_k$ . The  $\mathbf{p}_k$  and  $\mathbf{x}_k$  are built up in such a way that  $\mathbf{x}_{k+1}$  is also the minimizer of  $f$  over the whole vector space of directions already taken,  $\{\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{k-1}\}$ . After  $N$  iterations you arrive at the minimizer over the entire vector space, i.e., the solution to (2.7.29).

Later, in §10.8, we will generalize this “ordinary” conjugate gradient algorithm to the minimization of arbitrary nonlinear functions. Here, where our interest is in solving linear, but not necessarily positive-definite or symmetric, equations, a different generalization is important, the *biconjugate gradient method*. This method does not, in general, have a simple connection with function minimization. It constructs four sequences of vectors,  $\mathbf{r}_k, \bar{\mathbf{r}}_k, \mathbf{p}_k, \bar{\mathbf{p}}_k, k = 0, 1, \dots$ . You supply the initial vectors  $\mathbf{r}_0$  and  $\bar{\mathbf{r}}_0$ , and set  $\mathbf{p}_0 = \mathbf{r}_0, \bar{\mathbf{p}}_0 = \bar{\mathbf{r}}_0$ . Then you carry out the following recurrence:

$$\begin{aligned} \alpha_k &= \frac{\bar{\mathbf{r}}_k \cdot \mathbf{r}_k}{\bar{\mathbf{p}}_k \cdot \mathbf{A} \cdot \mathbf{p}_k} \\ \mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k \mathbf{A} \cdot \mathbf{p}_k \\ \bar{\mathbf{r}}_{k+1} &= \bar{\mathbf{r}}_k - \alpha_k \mathbf{A}^T \cdot \bar{\mathbf{p}}_k \\ \beta_k &= \frac{\bar{\mathbf{r}}_{k+1} \cdot \mathbf{r}_{k+1}}{\bar{\mathbf{r}}_k \cdot \mathbf{r}_k} \\ \mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \\ \bar{\mathbf{p}}_{k+1} &= \bar{\mathbf{r}}_{k+1} + \beta_k \bar{\mathbf{p}}_k \end{aligned} \quad (2.7.32)$$

This sequence of vectors satisfies the *biorthogonality* condition

$$\bar{\mathbf{r}}_i \cdot \mathbf{r}_j = \mathbf{r}_i \cdot \bar{\mathbf{r}}_j = 0, \quad j < i \quad (2.7.33)$$

and the *biconjugacy* condition

$$\bar{\mathbf{p}}_i \cdot \mathbf{A} \cdot \mathbf{p}_j = \mathbf{p}_i \cdot \mathbf{A}^T \cdot \bar{\mathbf{p}}_j = 0, \quad j < i \quad (2.7.34)$$

There is also a mutual orthogonality,

$$\bar{\mathbf{r}}_i \cdot \mathbf{p}_j = \mathbf{r}_i \cdot \bar{\mathbf{p}}_j = 0, \quad j < i \quad (2.7.35)$$

The proof of these properties proceeds by straightforward induction [12]. As long as the recurrence does not break down earlier because one of the denominators is zero, it must terminate after  $m \leq N$  steps with  $\mathbf{r}_m = \bar{\mathbf{r}}_m = 0$ . This is basically because after at most  $N$  steps you run out of new orthogonal directions to the vectors you’ve already constructed.

To use the algorithm to solve the system (2.7.29), make an initial guess  $\mathbf{x}_0$  for the solution. Choose  $\mathbf{r}_0$  to be the *residual*

$$\mathbf{r}_0 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_0 \quad (2.7.36)$$

and choose  $\bar{\mathbf{r}}_0 = \mathbf{r}_0$ . Then form the sequence of improved estimates

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (2.7.37)$$

while carrying out the recurrence (2.7.32). Equation (2.7.37) guarantees that  $\mathbf{r}_{k+1}$  from the recurrence is in fact the residual  $\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{k+1}$  corresponding to  $\mathbf{x}_{k+1}$ . Since  $\mathbf{r}_m = 0, \mathbf{x}_m$  is the solution to equation (2.7.29).

While there is no guarantee that this whole procedure will not break down or become unstable for general  $\mathbf{A}$ , in practice this is rare. More importantly, the exact termination in at most  $N$  iterations occurs only with exact arithmetic. Roundoff error means that you should

regard the process as a genuinely iterative procedure, to be halted when some appropriate error criterion is met.

The ordinary conjugate gradient algorithm is the special case of the biconjugate gradient algorithm when  $\mathbf{A}$  is symmetric, and we choose  $\bar{\mathbf{r}}_0 = \mathbf{r}_0$ . Then  $\bar{\mathbf{r}}_k = \mathbf{r}_k$  and  $\bar{\mathbf{p}}_k = \mathbf{p}_k$  for all  $k$ ; you can omit computing them and halve the work of the algorithm. This conjugate gradient version has the interpretation of minimizing equation (2.7.30). If  $\mathbf{A}$  is positive-definite as well as symmetric, the algorithm cannot break down (in theory!). The `solve` routine `Linbcg` below indeed reduces to the ordinary conjugate gradient method if you input a symmetric  $\mathbf{A}$ , but it does all the redundant computations.

Another variant of the general algorithm corresponds to a symmetric but non-positive-definite  $\mathbf{A}$ , with the choice  $\bar{\mathbf{r}}_0 = \mathbf{A} \cdot \mathbf{r}_0$  instead of  $\bar{\mathbf{r}}_0 = \mathbf{r}_0$ . In this case  $\bar{\mathbf{r}}_k = \mathbf{A} \cdot \mathbf{r}_k$  and  $\bar{\mathbf{p}}_k = \mathbf{A} \cdot \mathbf{p}_k$  for all  $k$ . This algorithm is thus equivalent to the ordinary conjugate gradient algorithm, but with all dot products  $\mathbf{a} \cdot \mathbf{b}$  replaced by  $\mathbf{a} \cdot \mathbf{A} \cdot \mathbf{b}$ . It is called the *minimum residual* algorithm, because it corresponds to successive minimizations of the function

$$\Phi(\mathbf{x}) = \frac{1}{2} \mathbf{r} \cdot \mathbf{r} = \frac{1}{2} |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|^2 \quad (2.7.38)$$

where the successive iterates  $\mathbf{x}_k$  minimize  $\Phi$  over the same set of search directions  $\mathbf{p}_k$  generated in the conjugate gradient method. This algorithm has been generalized in various ways for unsymmetric matrices. The *generalized minimum residual* method (GMRES; see [13,14]) is probably the most robust of these methods.

Note that equation (2.7.38) gives

$$\nabla \Phi(\mathbf{x}) = \mathbf{A}^T \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) \quad (2.7.39)$$

For any nonsingular matrix  $\mathbf{A}$ ,  $\mathbf{A}^T \cdot \mathbf{A}$  is symmetric and positive-definite. You might therefore be tempted to solve equation (2.7.29) by applying the ordinary conjugate gradient algorithm to the problem

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{x} = \mathbf{A}^T \cdot \mathbf{b} \quad (2.7.40)$$

Don't! The condition number of the matrix  $\mathbf{A}^T \cdot \mathbf{A}$  is the square of the condition number of  $\mathbf{A}$  (see §2.6 for definition of condition number). A large condition number both increases the number of iterations required and limits the accuracy to which a solution can be obtained. It is almost always better to apply the biconjugate gradient method to the original matrix  $\mathbf{A}$ .

So far we have said nothing about the *rate* of convergence of these methods. The ordinary conjugate gradient method works well for matrices that are well-conditioned, i.e., “close” to the identity matrix. This suggests applying these methods to the *preconditioned* form of equation (2.7.29),

$$(\tilde{\mathbf{A}}^{-1} \cdot \mathbf{A}) \cdot \mathbf{x} = \tilde{\mathbf{A}}^{-1} \cdot \mathbf{b} \quad (2.7.41)$$

The idea is that you might already be able to solve your linear system easily for some  $\tilde{\mathbf{A}}$  close to  $\mathbf{A}$ , in which case  $\tilde{\mathbf{A}}^{-1} \cdot \mathbf{A} \approx \mathbf{I}$ , allowing the algorithm to converge in fewer steps. The matrix  $\tilde{\mathbf{A}}$  is called a *preconditioner* [9], and the overall scheme given here is known as the *preconditioned biconjugate gradient method* or PBCG. In the code below, the user-supplied routine `atimes` does sparse matrix multiplication by  $\mathbf{A}$ , while the user-supplied routine `asolve` effects matrix multiplication by the inverse of the preconditioner  $\tilde{\mathbf{A}}^{-1}$ .

For efficient implementation, the PBCG algorithm introduces an additional set of vectors  $\mathbf{z}_k$  and  $\bar{\mathbf{z}}_k$  defined by

$$\tilde{\mathbf{A}} \cdot \mathbf{z}_k = \mathbf{r}_k \quad \text{and} \quad \tilde{\mathbf{A}}^T \cdot \bar{\mathbf{z}}_k = \bar{\mathbf{r}}_k \quad (2.7.42)$$

and modifies the definitions of  $\alpha_k$ ,  $\beta_k$ ,  $\mathbf{p}_k$ , and  $\bar{\mathbf{p}}_k$  in equation (2.7.32):

$$\begin{aligned} \alpha_k &= \frac{\bar{\mathbf{r}}_k \cdot \mathbf{z}_k}{\bar{\mathbf{p}}_k \cdot \mathbf{A} \cdot \mathbf{p}_k} \\ \beta_k &= \frac{\bar{\mathbf{r}}_{k+1} \cdot \mathbf{z}_{k+1}}{\bar{\mathbf{r}}_k \cdot \mathbf{z}_k} \\ \mathbf{p}_{k+1} &= \mathbf{z}_{k+1} + \beta_k \mathbf{p}_k \\ \bar{\mathbf{p}}_{k+1} &= \bar{\mathbf{z}}_{k+1} + \beta_k \bar{\mathbf{p}}_k \end{aligned} \quad (2.7.43)$$

To use Linbcg, below, you will need to supply routines that solve the auxiliary linear systems (2.7.42). If you have no idea what to use for the preconditioner  $\tilde{\mathbf{A}}$ , then use the diagonal part of  $\mathbf{A}$ , or even the identity matrix, in which case the burden of convergence will be entirely on the biconjugate gradient method itself.

Linbcg's routine `solve`, below, is based on a program originally written by Anne Greenbaum. (See [11] for a different, less sophisticated, implementation.) There are a few wrinkles you should know about.

What constitutes “good” convergence is rather application-dependent. The routine `solve` therefore provides for four possibilities, selected by setting the flag `itol` on input. If `itol=1`, iteration stops when the quantity  $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}| / |\mathbf{b}|$  is less than the input quantity `tol`. If `itol=2`, the required criterion is

$$|\tilde{\mathbf{A}}^{-1} \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b})| / |\tilde{\mathbf{A}}^{-1} \cdot \mathbf{b}| < \text{tol} \quad (2.7.44)$$

If `itol=3`, the routine uses its own estimate of the error in  $\mathbf{x}$  and requires its magnitude, divided by the magnitude of  $\mathbf{x}$ , to be less than `tol`. The setting `itol=4` is the same as `itol=3`, except that the largest (in absolute value) component of the error and largest component of  $\mathbf{x}$  are used instead of the vector magnitude (that is, the  $L_\infty$  norm instead of the  $L_2$  norm). You may need to experiment to find which of these convergence criteria is best for your problem.

On output, `err` is the tolerance actually achieved. If the returned count `iter` does not indicate that the maximum number of allowed iterations `itmax` was exceeded, then `err` should be less than `tol`. If you want to do further iterations, leave all returned quantities as they are and call the routine again. The routine loses its memory of the spanned conjugate gradient subspace between calls, however, so you should not force it to return more often than about every  $N$  iterations.

```
linbcg.h struct Linbcg {
    Abstract base class for solving sparse linear equations by the preconditioned biconjugate gradient
    method. To use, declare a derived class in which the methods atimes and asolve are defined
    for your problem, along with any data that they need. Then call the solve method.
    virtual void asolve(VecDoub_I &b, VecDoub_O &x, const Int itrnsp) = 0;
    virtual void atimes(VecDoub_I &x, VecDoub_O &r, const Int itrnsp) = 0;
    void solve(VecDoub_I &b, VecDoub_IO &x, const Int itol, const Doub tol,
               const Int itmax, Int &iter, Doub &err);
    Doub snrm(VecDoub_I &sx, const Int itol);           Utility used by solve.
};

void Linbcg::solve(VecDoub_I &b, VecDoub_IO &x, const Int itol, const Doub tol,
                   const Int itmax, Int &iter, Doub &err)
Solves  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$  for  $\mathbf{x}[0..n-1]$ , given  $\mathbf{b}[0..n-1]$ , by the iterative biconjugate gradient method.
On input  $\mathbf{x}[0..n-1]$  should be set to an initial guess of the solution (or all zeros); itol is 1,2,3,
or 4, specifying which convergence test is applied (see text); itmax is the maximum number
of allowed iterations; and tol is the desired convergence tolerance. On output,  $\mathbf{x}[0..n-1]$  is
reset to the improved solution, iter is the number of iterations actually taken, and err is the
estimated error. The matrix  $\mathbf{A}$  is referenced only through the user-supplied routines atimes,
which computes the product of either  $\mathbf{A}$  or its transpose on a vector, and asolve, which solves
 $\tilde{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b}$  or  $\tilde{\mathbf{A}}^T \cdot \mathbf{x} = \mathbf{b}$  for some preconditioner matrix  $\tilde{\mathbf{A}}$  (possibly the trivial diagonal part
of  $\mathbf{A}$ ). This routine can be called repeatedly, with itmax≤ $n$ , to monitor how err decreases;
or it can be called once with a sufficiently large value of itmax so that convergence to tol is
achieved.
{
    Doub ak,akden,bk,bkden=1.0,bknum,bnrm,dxnrm,xnrm,zm1nrm,znrm;
    const Doub EPS=1.0e-14;
    Int j,n=b.size();
    VecDoub p(n),pp(n),r(n),rr(n),z(n),zz(n);
    iter=0;
    atimes(x,r,0);                                Calculate initial residual.
    Input to atimes is x[0..n-1], output is r[0..n-1];
    for (j=0;j<n;j++) {
        r[j]=b[j]-r[j];
        rr[j]=r[j];
    }
}
```

```

//atimes(r,rr,0);
if (itol == 1) {
    bnrm=snrm(b,itol);
    asolve(r,z,0);
}
else if (itol == 2) {
    asolve(b,z,0);
    bnrm=snrm(z,itol);
    asolve(r,z,0);
}
else if (itol == 3 || itol == 4) {
    asolve(b,z,0);
    bnrm=snrm(z,itol);
    asolve(r,z,0);
    znmr=snrm(z,itol);
} else throw("illegal itol in linbcg");
while (iter < itmax) {           Main loop.
    ++iter;
    asolve(rr,zz,1);           Final 1 indicates use of transpose matrix  $\tilde{\mathbf{A}}^T$ .
    for (bknum=0.0,j=0;j<n;j++) bknum += z[j]*rr[j];
    Calculate coefficient bk and direction vectors p and pp.
    if (iter == 1) {
        for (j=0;j<n;j++) {
            p[j]=z[j];
            pp[j]=zz[j];
        }
    } else {
        bk=bknum/bkden;
        for (j=0;j<n;j++) {
            p[j]=bk*p[j]+z[j];
            pp[j]=bk*pp[j]+zz[j];
        }
    }
    bkden=bknum;                 Calculate coefficient ak, new iterate x, and new
    atimes(p,z,0);             residuals r and rr.
    for (akden=0.0,j=0;j<n;j++) akden += z[j]*pp[j];
    ak=bknum/akden;
    atimes(pp,zz,1);
    for (j=0;j<n;j++) {
        x[j] += ak*p[j];
        r[j] -= ak*z[j];
        rr[j] -= ak*zz[j];
    }
    asolve(r,z,0);           Solve  $\tilde{\mathbf{A}} \cdot \mathbf{z} = \mathbf{r}$  and check stopping criterion.
    if (itol == 1)
        err=snrm(r,itol)/bnrm;
    else if (itol == 2)
        err=snrm(z,itol)/bnrm;
    else if (itol == 3 || itol == 4) {
        zm1nrm=znrm;
        znrm=snrm(z,itol);
        if (abs(zm1nrm-znrm) > EPS*znrm) {
            dxnrm=abs(ak)*snrm(p,itol);
            err=znrm/abs(zm1nrm-znrm)*dxnrm;
        } else {
            err=znrm/bnrm;           Error may not be accurate, so loop again.
            continue;
        }
        xnrm=snrm(x,itol);
        if (err <= 0.5*xnrm) err /= xnrm;
        else {
            err=znrm/bnrm;           Error may not be accurate, so loop again.
            continue;
        }
    }
}

```

```

        }
        if (err <= tol) break;
    }
}

```

The `solve` routine uses this short utility for computing vector norms:

```
linbcg.h Doub Linbcg::snrm(VecDoub_I &sx, const Int itol)
Compute one of two norms for a vector sx[0..n-1], as signaled by itol. Used by solve.
{
    Int i,isamax,n=sx.size();
    Doub ans;
    if (itol <= 3) {
        ans = 0.0;
        for (i=0;i<n;i++) ans += SQR(sx[i]);      Vector magnitude norm.
        return sqrt(ans);
    } else {
        isamax=0;
        for (i=0;i<n;i++) {                      Largest component norm.
            if (abs(sx[i]) > abs(sx[isamax])) isamax=i;
        }
        return abs(sx[isamax]);
    }
}
```

Here is an example of a derived class that solves  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$  for a matrix  $\mathbf{A}$  in `NRsparseMat`'s compressed column sparse format. A naive diagonal preconditioner is used.

```
asolve.h struct NRsparseLinbcg : Linbcg {
    NRsparseMat &mat;
    Int n;
    NRsparseLinbcg(NRsparseMat &matrix) : mat(matrix), n(mat.nrows) {}
The constructor just binds a reference to your sparse matrix, making it available to asolve
and atimes. To solve for a right-hand side, you call this object's solve method, as defined
in the base class.
    void atimes(VecDoub_I &x, VecDoub_O &r, const Int itrnsp) {
        if (itrnsp) r=mat.atx(x);
        else r=mat.ax(x);
    }
    void asolve(VecDoub_I &b, VecDoub_O &x, const Int itrnsp) {
        Int i,j;
        Doub diag;
        for (i=0;i<n;i++) {
            diag=0.0;
            for (j=mat.col_ptr[i];j<mat.col_ptr[i+1];j++)
                if (mat.row_ind[j] == i) {
                    diag=mat.val[j];
                    break;
                }
            x[i]=(diag != 0.0 ? b[i]/diag : b[i]);
        }
    }
};
```

The matrix  $\tilde{\mathbf{A}}$  is the diagonal part of  $\mathbf{A}$ . Since the transpose matrix has the same diagonal, the flag `itrnsp` is not used in this example.

For another example of using a class derived from `Linbcg` to solve a sparse matrix problem, see §3.8.

#### CITED REFERENCES AND FURTHER READING:

Tewarson, R.P. 1973, *Sparse Matrices* (New York: Academic Press).[1]

- Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter I.3 (by J.K. Reid).[2]
- George, A., and Liu, J.W.H. 1981, *Computer Solution of Large Sparse Positive Definite Systems* (Englewood Cliffs, NJ: Prentice-Hall).[3]
- NAG Fortran Library (Oxford, UK: Numerical Algorithms Group), see 2007+, <http://www.nag.co.uk>.[4]
- IMSL Math/Library Users Manual (Houston: IMSL Inc.), see 2007+, <http://www.vni.com/products/imsl>.[5]
- Eisenstat, S.C., Gursky, M.C., Schultz, M.H., and Sherman, A.H. 1977, *Yale Sparse Matrix Package*, Technical Reports 112 and 114 (Yale University Department of Computer Science).[6]
- Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., and van der Vorst, H. (eds.) 2000, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide* Ch. 10 (Philadelphia: S.I.A.M.). Online at URL in <http://www.cs.ucdavis.edu/~bai/ET/contents.html>.[7]
- SPARSKIT, 2007+, at <http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html>.[8]
- Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), Chapters 4 and 10, particularly §10.2–§10.3.[9]
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 8.[10]
- Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill).[11]
- Fletcher, R. 1976, in *Numerical Analysis Dundee 1975*, Lecture Notes in Mathematics, vol. 506, A. Dold and B. Eckmann, eds. (Berlin: Springer), pp. 73–89.[12]
- PCGPAK User's Guide (New Haven: Scientific Computing Associates, Inc.).[13]
- Saad, Y., and Schulz, M. 1986, *SIAM Journal on Scientific and Statistical Computing*, vol. 7, pp. 856–869.[14]
- Ueberhuber, C.W. 1997, *Numerical Computation: Methods, Software, and Analysis*, 2 vols. (Berlin: Springer), Chapter 13.
- Bunch, J.R., and Rose, D.J. (eds.) 1976, *Sparse Matrix Computations* (New York: Academic Press).
- Duff, I.S., and Stewart, G.W. (eds.) 1979, *Sparse Matrix Proceedings 1978* (Philadelphia: S.I.A.M.).

## 2.8 Vandermonde Matrices and Toeplitz Matrices

In §2.4 the case of a tridiagonal matrix was treated specially, because that particular type of linear system admits a solution in only of order  $N$  operations, rather than of order  $N^3$  for the general linear problem. When such particular types exist, it is important to know about them. Your computational savings, should you ever happen to be working on a problem that involves the right kind of particular type, can be enormous.

This section treats two special types of matrices that can be solved in of order  $N^2$  operations, not as good as tridiagonal, but a lot better than the general case. (Other than the operations count, these two types having nothing in common.) Matrices of the first type, termed *Vandermonde matrices*, occur in some problems having to do with the fitting of polynomials, the reconstruction of distributions from their moments, and also other contexts. In this book, for example, a Vandermonde problem crops up in §3.5. Matrices of the second type, termed *Toeplitz matrices*, tend

to occur in problems involving deconvolution and signal processing. In this book, a Toeplitz problem is encountered in §13.7.

These are not the only special types of matrices worth knowing about. The *Hilbert matrices*, whose components are of the form  $a_{ij} = 1/(i + j + 1)$ ,  $i, j = 0, \dots, N - 1$ , can be inverted by an exact integer algorithm and are very difficult to invert in any other way, since they are notoriously ill-conditioned (see [1] for details). The Sherman-Morrison and Woodbury formulas, discussed in §2.7, can sometimes be used to convert new special forms into old ones. Reference [2] gives some other special forms. We have not found these additional forms to arise as frequently as the two that we now discuss.

### 2.8.1 Vandermonde Matrices

A Vandermonde matrix of size  $N \times N$  is completely determined by  $N$  arbitrary numbers  $x_0, x_1, \dots, x_{N-1}$ , in terms of which its  $N^2$  components are the integer powers  $x_i^j$ ,  $i, j = 0, \dots, N - 1$ . Evidently there are two possible such forms, depending on whether we view the  $i$ 's as rows and  $j$ 's as columns, or vice versa. In the former case, we get a linear system of equations that looks like this,

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \cdots & x_{N-1}^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} \quad (2.8.1)$$

Performing the matrix multiplication, you will see that this equation solves for the unknown coefficients  $c_i$  that fit a polynomial to the  $N$  pairs of abscissas and ordinates  $(x_j, y_j)$ . Precisely this problem will arise in §3.5, and the routine given there will solve (2.8.1) by the method that we are about to describe.

The alternative identification of rows and columns leads to the set of equations

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_{N-1} \\ x_0^2 & x_1^2 & \cdots & x_{N-1}^2 \\ \vdots & \vdots & & \vdots \\ x_0^{N-1} & x_1^{N-1} & \cdots & x_{N-1}^{N-1} \end{bmatrix} \cdot \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{N-1} \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ \vdots \\ q_{N-1} \end{bmatrix} \quad (2.8.2)$$

Write this out and you will see that it relates to the *problem of moments*: Given the values of  $N$  points  $x_i$ , find the unknown weights  $w_i$ , assigned so as to match the given values  $q_j$  of the first  $N$  moments. (For more on this problem, consult [3].) The routine given in this section solves (2.8.2).

The method of solution of both (2.8.1) and (2.8.2) is closely related to Lagrange's polynomial interpolation formula, which we will not formally meet until §3.2. Notwithstanding, the following derivation should be comprehensible:

Let  $P_j(x)$  be the polynomial of degree  $N - 1$  defined by

$$P_j(x) = \prod_{\substack{n=0 \\ n \neq j}}^{N-1} \frac{x - x_n}{x_j - x_n} = \sum_{k=0}^{N-1} A_{jk} x^k \quad (2.8.3)$$

Here the meaning of the last equality is to define the components of the matrix  $A_{ij}$  as the coefficients that arise when the product is multiplied out and like terms collected.

The polynomial  $P_j(x)$  is a function of  $x$  generally. But you will notice that it is specifically designed so that it takes on a value of zero at all  $x_i$  with  $i \neq j$  and has a value of unity at  $x = x_j$ . In other words,

$$P_j(x_i) = \delta_{ij} = \sum_{k=0}^{N-1} A_{jk} x_i^k \quad (2.8.4)$$

But (2.8.4) says that  $A_{jk}$  is exactly the inverse of the matrix of components  $x_i^k$ , which appears in (2.8.2), with the subscript as the column index. Therefore the solution of (2.8.2) is just that matrix inverse times the right-hand side,

$$w_j = \sum_{k=0}^{N-1} A_{jk} q_k \quad (2.8.5)$$

As for the transpose problem (2.8.1), we can use the fact that the inverse of the transpose is the transpose of the inverse, so

$$c_j = \sum_{k=0}^{N-1} A_{kj} y_k \quad (2.8.6)$$

The routine in §3.5 implements this.

It remains to find a good way of multiplying out the monomial terms in (2.8.3), in order to get the components of  $A_{jk}$ . This is essentially a bookkeeping problem, and we will let you read the routine itself to see how it can be solved. One trick is to define a master  $P(x)$  by

$$P(x) \equiv \prod_{n=0}^{N-1} (x - x_n) \quad (2.8.7)$$

work out its coefficients, and then obtain the numerators and denominators of the specific  $P_j$ 's via synthetic division by the one supernumerary term. (See §5.1 for more on synthetic division.) Since each such division is only a process of order  $N$ , the total procedure is of order  $N^2$ .

You should be warned that Vandermonde systems are notoriously ill-conditioned, by their very nature. (As an aside anticipating §5.8, the reason is the same as that which makes Chebyshev fitting so impressively accurate: There exist high-order polynomials that are very good uniform fits to zero. Hence roundoff error can introduce rather substantial coefficients of the leading terms of these polynomials.) It is a good idea always to compute Vandermonde problems in double precision or higher.

The routine for (2.8.2) that follows is due to G.B. Rybicki.

```
void vander(VecDoub_I &x, VecDoub_O &w, VecDoub_I &q)
Solves the Vandermonde linear system  $\sum_{i=0}^{N-1} x_i^k w_i = q_k$  ( $k = 0, \dots, N-1$ ). Input consists
of the vectors x[0..n-1] and q[0..n-1]; the vector w[0..n-1] is output.
{
    Int i,j,k,n=q.size();
    Doub b,s,t,xx;
    VecDoub c(n);
    if (n == 1) w[0]=q[0];
    else {
        for (i=0;i<n;i++) c[i]=0.0;           Initialize array.
        c[n-1] = -x[0];                         Coefficients of the master polynomial are found
        for (i=1;i<n;i++) {                     by recursion.
            xx = -x[i];
            for (j=(n-1-i);j<(n-1);j++) c[j] += xx*c[j+1];
            c[n-1] += xx;
        }
        for (i=0;i<n;i++) {                   Each subfactor in turn
            s = 1.0;
            for (j=0;j<i;j++) s *= x[j];
            b = q[i] - s*c[i];
            for (j=i;j<n;j++) c[j] -= b*x[j];
        }
    }
}
```

```

xx=x[i];
t=b=1.0;
s=q[n-1];
for (k=n-1;k>0;k--) {           is synthetically divided,
    b=c[k]+xx*b;
    s += q[k-1]*b;                 matrix-multiplied by the right-hand side,
    t=xx*t+b;
}
w[i]=s/t;                         and supplied with a denominator.
}
}
}

```

## 2.8.2 Toeplitz Matrices

An  $N \times N$  Toeplitz matrix is specified by giving  $2N - 1$  numbers  $R_k$ , where the index  $k$  ranges over  $k = -N + 1, \dots, -1, 0, 1, \dots, N - 1$ . Those numbers are then emplaced as matrix elements constant along the (upper-left to lower-right) diagonals of the matrix:

$$\begin{bmatrix} R_0 & R_{-1} & R_{-2} & \cdots & R_{-(N-2)} & R_{-(N-1)} \\ R_1 & R_0 & R_{-1} & \cdots & R_{-(N-3)} & R_{-(N-2)} \\ R_2 & R_1 & R_0 & \cdots & R_{-(N-4)} & R_{-(N-3)} \\ \cdots & & & \cdots & & \\ R_{N-2} & R_{N-3} & R_{N-4} & \cdots & R_0 & R_{-1} \\ R_{N-1} & R_{N-2} & R_{N-3} & \cdots & R_1 & R_0 \end{bmatrix} \quad (2.8.8)$$

The linear Toeplitz problem can thus be written as

$$\sum_{j=0}^{N-1} R_{i-j} x_j = y_i \quad (i = 0, \dots, N-1) \quad (2.8.9)$$

where the  $x_j$ 's,  $j = 0, \dots, N-1$ , are the unknowns to be solved for.

The Toeplitz matrix is symmetric if  $R_k = R_{-k}$  for all  $k$ . Levinson [4] developed an algorithm for fast solution of the symmetric Toeplitz problem, by a *bordering method*, that is, a recursive procedure that solves the  $(M+1)$ -dimensional Toeplitz problem

$$\sum_{j=0}^M R_{i-j} x_j^{(M)} = y_i \quad (i = 0, \dots, M) \quad (2.8.10)$$

in turn for  $M = 0, 1, \dots$  until  $M = N - 1$ , the desired result, is finally reached. The vector  $x_j^{(M)}$  is the result at the  $M$ th stage and becomes the desired answer only when  $N - 1$  is reached.

Levinson's method is well documented in standard texts (e.g., [5]). The useful fact that the method generalizes to the *nonsymmetric* case seems to be less well known. At some risk of excessive detail, we therefore give a derivation here, due to G.B. Rybicki.

In following a recursion from step  $M$  to step  $M + 1$  we find that our developing solution  $x^{(M)}$  changes in this way:

$$\sum_{j=0}^M R_{i-j} x_j^{(M)} = y_i \quad i = 0, \dots, M \quad (2.8.11)$$

becomes

$$\sum_{j=0}^M R_{i-j} x_j^{(M+1)} + R_{i-(M+1)} x_{M+1}^{(M+1)} = y_i \quad i = 0, \dots, M+1 \quad (2.8.12)$$

By eliminating  $y_i$  we find

$$\sum_{j=0}^M R_{i-j} \left( \frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \right) = R_{i-(M+1)} \quad i = 0, \dots, M \quad (2.8.13)$$

or by letting  $i \rightarrow M - i$  and  $j \rightarrow M - j$ ,

$$\sum_{j=0}^M R_{j-i} G_j^{(M)} = R_{-(i+1)} \quad (2.8.14)$$

where

$$G_j^{(M)} \equiv \frac{x_{M-j}^{(M)} - x_{M-j}^{(M+1)}}{x_{M+1}^{(M+1)}} \quad (2.8.15)$$

To put this another way,

$$x_{M-j}^{(M+1)} = x_{M-j}^{(M)} - x_{M+1}^{(M+1)} G_j^{(M)} \quad j = 0, \dots, M \quad (2.8.16)$$

Thus, if we can use recursion to find the order  $M$  quantities  $x^{(M)}$  and  $G^{(M)}$  and the single order  $M + 1$  quantity  $x_{M+1}^{(M+1)}$ , then all of the other  $x_j^{(M+1)}$ 's will follow. Fortunately, the quantity  $x_{M+1}^{(M+1)}$  follows from equation (2.8.12) with  $i = M + 1$ ,

$$\sum_{j=0}^M R_{M+1-j} x_j^{(M+1)} + R_0 x_{M+1}^{(M+1)} = y_{M+1} \quad (2.8.17)$$

For the unknown order  $M + 1$  quantities  $x_j^{(M+1)}$  we can substitute the previous order quantities in  $G$  since

$$G_{M-j}^{(M)} = \frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \quad (2.8.18)$$

The result of this operation is

$$x_{M+1}^{(M+1)} = \frac{\sum_{j=0}^M R_{M+1-j} x_j^{(M)} - y_{M+1}}{\sum_{j=0}^M R_{M+1-j} G_{M-j}^{(M)} - R_0} \quad (2.8.19)$$

The only remaining problem is to develop a recursion relation for  $G$ . Before we do that, however, we should point out that there are actually two distinct sets of solutions to the original linear problem for a nonsymmetric matrix, namely right-hand solutions (which we have been discussing) and left-hand solutions  $z_i$ . The formalism for the left-hand solutions differs only in that we deal with the equations

$$\sum_{j=0}^M R_{j-i} z_j^{(M)} = y_i \quad i = 0, \dots, M \quad (2.8.20)$$

Then, the same sequence of operations on this set leads to

$$\sum_{j=0}^M R_{i-j} H_j^{(M)} = R_{i+1} \quad (2.8.21)$$

where

$$H_j^{(M)} \equiv \frac{z_{M-j}^{(M)} - z_{M-j}^{(M+1)}}{z_{M+1}^{(M+1)}} \quad (2.8.22)$$

(compare with 2.8.14 – 2.8.15). The reason for mentioning the left-hand solutions now is that, by equation (2.8.21), the  $H_j$ 's satisfy exactly the same equation as the  $x_j$ 's except for the substitution  $y_i \rightarrow R_{i+1}$  on the right-hand side. Therefore we can quickly deduce from equation (2.8.19) that

$$H_{M+1}^{(M+1)} = \frac{\sum_{j=0}^M R_{M+1-j} H_j^{(M)} - R_{M+2}}{\sum_{j=0}^M R_{M+1-j} G_{M-j}^{(M)} - R_0} \quad (2.8.23)$$

By the same token,  $G$  satisfies the same equation as  $z$ , except for the substitution  $y_i \rightarrow R_{-(i+1)}$ . This gives

$$G_{M+1}^{(M+1)} = \frac{\sum_{j=0}^M R_{j-M-1} G_j^{(M)} - R_{-M-2}}{\sum_{j=0}^M R_{j-M-1} H_{M-j}^{(M)} - R_0} \quad (2.8.24)$$

The same “morphism” also turns equation (2.8.16), and its partner for  $z$ , into the final equations

$$\begin{aligned} G_j^{(M+1)} &= G_j^{(M)} - G_{M+1}^{(M+1)} H_{M-j}^{(M)} \\ H_j^{(M+1)} &= H_j^{(M)} - H_{M+1}^{(M+1)} G_{M-j}^{(M)} \end{aligned} \quad (2.8.25)$$

Now, starting with the initial values

$$x_0^{(0)} = y_0/R_0 \quad G_0^{(0)} = R_{-1}/R_0 \quad H_0^{(0)} = R_1/R_0 \quad (2.8.26)$$

we can recurse away. At each stage  $M$  we use equations (2.8.23) and (2.8.24) to find  $H_{M+1}^{(M+1)}$ ,  $G_{M+1}^{(M+1)}$ , and then equation (2.8.25) to find the other components of  $H^{(M+1)}$ ,  $G^{(M+1)}$ . From there the vectors  $x^{(M+1)}$  and/or  $z^{(M+1)}$  are easily calculated.

The program below does this. It incorporates the second equation in (2.8.25) in the form

$$H_{M-j}^{(M+1)} = H_{M-j}^{(M)} - H_{M+1}^{(M+1)} G_j^{(M)} \quad (2.8.27)$$

so that the computation can be done “in place.”

Notice that the above algorithm fails if  $R_0 = 0$ . In fact, because the bordering method does not allow pivoting, the algorithm will fail if any of the diagonal principal minors of the original Toeplitz matrix vanish. (Compare with discussion of the tridiagonal algorithm in §2.4.) If the algorithm fails, your matrix is not necessarily singular — you might just have to solve your problem by a slower and more general algorithm such as  $LU$  decomposition with pivoting.

The routine that implements equations (2.8.23) – (2.8.27) is also due to Rybicki. Note that the routine's  $\mathbf{r}[n-1+j]$  is equal to  $R_j$  above, so that subscripts on the  $\mathbf{r}$  array vary from 0 to  $2N - 2$ .

```
toeplz.h void toeplz(VecDoub_I &r, VecDoub_O &x, VecDoub_I &y)
Solves the Toeplitz system  $\sum_{j=0}^{N-1} R_{(N-1+i-j)} x_j = y_i$  ( $i = 0, \dots, N-1$ ). The Toeplitz
matrix need not be symmetric.  $\mathbf{y}[0..n-1]$  and  $\mathbf{r}[0..2*n-2]$  are input arrays;  $\mathbf{x}[0..n-1]$  is the
output array.
{
    Int j,k,m,m1,m2,n1,n=y.size();
    Doub pp,pt1,pt2,qq,qt1,qt2,sd,sgd,sgn,shn,sxn;
    n1=n-1;
```

```

if (r[n1] == 0.0) throw("toeplz-1 singular principal minor");
x[0]=y[0]/r[n1];                                Initialize for the recursion.
if (n1 == 0) return;
VecDoub g(n1),h(n1);
g[0]=r[n1-1]/r[n1];
h[0]=r[n1+1]/r[n1];
for (m=0;m<n;m++) {                           Main loop over the recursion.
    m1=m+1;
    sxn = -y[m1];                            Compute numerator and denominator for  $x$  from eq.
    sd = -r[n1];                             (2.8.19),
    for (j=0;j<m1;j++) {
        sxn += r[n1+m1-j]*x[j];
        sd += r[n1+m1-j]*g[m-j];
    }
    if (sd == 0.0) throw("toeplz-2 singular principal minor");
    x[m1]=sxn/sd;                          whence  $x$ .
    for (j=0;j<m1;j++) {                  Eq. (2.8.16).
        x[j] -= x[m1]*g[m-j];
    if (m1 == n1) return;
    sgn = -r[n1-m1-1];                    Compute numerator and denominator for  $G$  and  $H$ ,
    shn = -r[n1+m1+1];                   eqs. (2.8.24) and (2.8.23),
    sgd = -r[n1];
    for (j=0;j<m1;j++) {
        sgn += r[n1+j-m1]*g[j];
        shn += r[n1+m1-j]*h[j];
        sgd += r[n1+j-m1]*h[m-j];
    }
    if (sgd == 0.0) throw("toeplz-3 singular principal minor");
    g[m1]=sgn/sgd;                      whence  $G$  and  $H$ .
    h[m1]=shn/sd;
    k=m;
    m2=(m+2) >> 1;
    pp=g[m1];
    qq=h[m1];
    for (j=0;j<m2;j++) {
        pt1=g[j];
        pt2=g[k];
        qt1=h[j];
        qt2=h[k];
        g[j]=pt1-pp*qt2;
        g[k]=pt2-pp*qt1;
        h[j]=qt1-qq*pt2;
        h[k--]=qt2-qq*pt1;
    }
}                                         Back for another recurrence.
throw("toeplz - should not arrive here!");
}

```

If you are in the business of solving *very* large Toeplitz systems, you should find out about so-called “new, fast” algorithms, which require only on the order of  $N(\log N)^2$  operations, compared to  $N^2$  for Levinson’s method. These methods are too complicated to include here. Papers by Bunch [6] and de Hoog [7] will give entry to the literature.

#### CITED REFERENCES AND FURTHER READING:

- Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), Chapter 5 [also treats some other special forms].
- Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), §19.[1]
- Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).[2]

- von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), pp. 394ff.[3]
- Levinson, N., Appendix B of N. Wiener, 1949, *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (New York: Wiley).[4]
- Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall), pp. 163ff.[5]
- Bunch, J.R. 1985, "Stability of Methods for Solving Toeplitz Systems of Equations," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, pp. 349–364.[6]
- de Hoog, F. 1987, "A New Algorithm for Solving Toeplitz Systems of Equations," *Linear Algebra and Its Applications*, vol. 88/89, pp. 123–138.[7]

## 2.9 Cholesky Decomposition

If a square matrix  $\mathbf{A}$  happens to be symmetric and positive-definite, then it has a special, more efficient, triangular decomposition. *Symmetric* means that  $a_{ij} = a_{ji}$  for  $i, j = 0, \dots, N - 1$ , while *positive-definite* means that

$$\mathbf{v} \cdot \mathbf{A} \cdot \mathbf{v} > 0 \quad \text{for all vectors } \mathbf{v} \quad (2.9.1)$$

(In Chapter 11 we will see that positive-definite has the equivalent interpretation that  $\mathbf{A}$  has all positive eigenvalues.) While symmetric, positive-definite matrices are rather special, they occur quite frequently in some applications, so their special factorization, called *Cholesky decomposition*, is good to know about. When you can use it, Cholesky decomposition is about a factor of two faster than alternative methods for solving linear equations.

Instead of seeking arbitrary lower and upper triangular factors  $\mathbf{L}$  and  $\mathbf{U}$ , Cholesky decomposition constructs a lower triangular matrix  $\mathbf{L}$  whose transpose  $\mathbf{L}^T$  can itself serve as the upper triangular part. In other words we replace equation (2.3.1) by

$$\mathbf{L} \cdot \mathbf{L}^T = \mathbf{A} \quad (2.9.2)$$

This factorization is sometimes referred to as “taking the square root” of the matrix  $\mathbf{A}$ , though, because of the transpose, it is not literally that. The components of  $\mathbf{L}^T$  are of course related to those of  $\mathbf{L}$  by

$$L_{ij}^T = L_{ji} \quad (2.9.3)$$

Writing out equation (2.9.2) in components, one readily obtains the analogs of equations (2.3.12) – (2.3.13),

$$L_{ii} = \left( a_{ii} - \sum_{k=0}^{i-1} L_{ik}^2 \right)^{1/2} \quad (2.9.4)$$

and

$$L_{ji} = \frac{1}{L_{ii}} \left( a_{ij} - \sum_{k=0}^{i-1} L_{ik} L_{jk} \right) \quad j = i + 1, i + 2, \dots, N - 1 \quad (2.9.5)$$

If you apply equations (2.9.4) and (2.9.5) in the order  $i = 0, 1, \dots, N - 1$ , you will see that the  $L$ 's that occur on the right-hand side are already determined by the time they are needed. Also, only components  $a_{ij}$  with  $j \geq i$  are referenced. (Since  $\mathbf{A}$  is symmetric, these have complete information.) If storage is at a premium, it is possible to have the factor  $\mathbf{L}$  overwrite the subdiagonal (lower triangular but not including the diagonal) part of  $\mathbf{A}$ , preserving the input upper triangular values of  $\mathbf{A}$ ; one extra vector of length  $N$  is then needed to store the diagonal part of  $\mathbf{L}$ . The operations count is  $N^3/6$  executions of the inner loop (consisting of one multiply and one subtract), with also  $N$  square roots. As already mentioned, this is about a factor 2 better than  $LU$  decomposition of  $\mathbf{A}$  (where its symmetry would be ignored).

You might wonder about pivoting. The pleasant answer is that Cholesky decomposition is extremely stable numerically, without any pivoting at all. Failure of the decomposition simply indicates that the matrix  $\mathbf{A}$  (or, with roundoff error, another very nearby matrix) is not positive-definite. In fact, this is an efficient way to test whether a symmetric matrix is positive-definite. (In this application, you may want to replace the `throw` in the code below with some less drastic signaling method.)

By now you should be familiar with, if not bored by, our conventions for objects implementing decomposition methods, so we list the object `Cholesky` as a single big mouthful. The methods `elmult` and `elsolve` perform manipulations using the matrix  $\mathbf{L}$ . The first multiplies  $\mathbf{L} \cdot \mathbf{y} = \mathbf{c}$  for a given  $\mathbf{y}$ , returning  $\mathbf{c}$ . The second solves this same equation, given  $\mathbf{c}$  and returning  $\mathbf{y}$ . These manipulations are useful in contexts such as multivariate Gaussians (§7.4 and §16.5) and in the analysis of covariance matrices (§15.6).

```
struct Cholesky{cholesky.h
Object for Cholesky decomposition of a matrix A, and related functions.
    Int n;
    MatDoub el; Stores the decomposition.
    Cholesky(MatDoub_I &a) : n(a.nrows()), el(a) {
        Constructor. Given a positive-definite symmetric matrix a[0..n-1][0..n-1], construct
        and store its Cholesky decomposition,  $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T$ .
        Int i,j,k;
        VecDoub tmp;
        Doub sum;
        if (el.ncols() != n) throw("need square matrix");
        for (i=0;i<n;i++) {
            for (j=i;j<n;j++) {
                for (sum=el[i][j],k=i-1;k>=0;k--) sum -= el[i][k]*el[j][k];
                if (i == j) {
                    if (sum <= 0.0) A, with rounding errors, is not positive-definite.
                        throw("Cholesky failed");
                    el[i][i]=sqrt(sum);
                } else el[j][i]=sum/el[i][i];
            }
        }
        for (i=0;i<n;i++) for (j=0;j<i;j++) el[j][i] = 0.;
    }
    void solve(VecDoub_I &b, VecDoub_O &x) {
        Solve the set of n linear equations  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ , where a is a positive-definite symmetric matrix
        whose Cholesky decomposition has been stored. b[0..n-1] is input as the right-hand side
        vector. The solution vector is returned in x[0..n-1].
        Int i,k;
        Doub sum;
        if (b.size() != n || x.size() != n) throw("bad lengths in Cholesky");
        for (i=0;i<n;i++) { Solve  $\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$ , storing  $\mathbf{y}$  in  $\mathbf{x}$ .
            for (sum=b[i],k=i-1;k>=0;k--) sum -= el[i][k]*x[k];
            x[i]=sum/el[i][i];
        }
        for (i=n-1;i>=0;i--) { Solve  $\mathbf{L}^T \cdot \mathbf{x} = \mathbf{y}$ .
            for (sum=x[i],k=i+1;k<n;k++) sum -= el[k][i]*x[k];
            x[i]=sum/el[i][i];
        }
    }
    void elmult(VecDoub_I &y, VecDoub_O &b) {
        Multiply  $\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$ , where  $\mathbf{L}$  is the lower triangular matrix in the stored Cholesky decom-
        position. y[0..n-1] is input. The result is returned in b[0..n-1].
        Int i,j;
        if (b.size() != n || y.size() != n) throw("bad lengths");
        for (i=0;i<n;i++) {
            b[i] = 0.;
            for (j=0;j<=i;j++) b[i] += el[i][j]*y[j];
        }
    }
}
```

```

void elsolve(VecDoub_I &b, VecDoub_O &y) {
Solve  $\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$ , where  $\mathbf{L}$  is the lower triangular matrix in the stored Cholesky decomposition.  $\mathbf{b}[0..n-1]$  is input as the right-hand side vector. The solution vector is returned in  $\mathbf{y}[0..n-1]$ .
    Int i,j;
    Doub sum;
    if (b.size() != n || y.size() != n) throw("bad lengths");
    for (i=0;i<n;i++) {
        for (sum=b[i],j=0; j<i; j++) sum -= el[i][j]*y[j];
        y[i] = sum/el[i][i];
    }
}
void inverse(MatDoub_O &ainv) {
Set ainv[0..n-1][0..n-1] to the matrix inverse of  $\mathbf{A}$ , the matrix whose Cholesky decomposition has been stored.
    Int i,j,k;
    Doub sum;
    ainv.resize(n,n);
    for (i=0;i<n;i++) for (j=0;j<=i;j++){
        sum = (i==j? 1. : 0.);
        for (k=i-1;k>=j;k--) sum -= el[i][k]*ainv[j][k];
        ainv[j][i] = sum/el[i][i];
    }
    for (i=n-1;i>=0;i--) for (j=0;j<=i;j++){
        sum = (i<j? 0. : ainv[j][i]);
        for (k=i+1;k<n;k++) sum -= el[k][i]*ainv[j][k];
        ainv[i][j] = ainv[j][i] = sum/el[i][i];
    }
}
Doub logdet() {
Return the logarithm of the determinant of  $\mathbf{A}$ , the matrix whose Cholesky decomposition has been stored.
    Doub sum = 0.;
    for (Int i=0; i<n; i++) sum += log(el[i][i]);
    return 2.*sum;
}
};

```

**CITED REFERENCES AND FURTHER READING:**

- Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer), Chapter I/1.
- Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), §4.9.2.
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §5.3.5.
- Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §4.2.

## 2.10 QR Decomposition

There is another matrix factorization that is sometimes very useful, the so-called *QR decomposition*,

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \quad (2.10.1)$$

Here  $\mathbf{R}$  is upper triangular, while  $\mathbf{Q}$  is orthogonal, that is,

$$\mathbf{Q}^T \cdot \mathbf{Q} = \mathbf{I} \quad (2.10.2)$$

where  $\mathbf{Q}^T$  is the transpose matrix of  $\mathbf{Q}$ . Although the decomposition exists for a general rectangular matrix, we shall restrict our treatment to the case when all the matrices are square, with dimensions  $N \times N$ .

Like the other matrix factorizations we have met ( $LU$ , SVD, Cholesky),  $QR$  decomposition can be used to solve systems of linear equations. To solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \quad (2.10.3)$$

first form  $\mathbf{Q}^T \cdot \mathbf{b}$  and then solve

$$\mathbf{R} \cdot \mathbf{x} = \mathbf{Q}^T \cdot \mathbf{b} \quad (2.10.4)$$

by backsubstitution. Since  $QR$  decomposition involves about twice as many operations as  $LU$  decomposition, it is not used for typical systems of linear equations. However, we will meet special cases where  $QR$  is the method of choice.

The standard algorithm for the  $QR$  decomposition involves successive Householder transformations (to be discussed later in §11.3). We write a Householder matrix in the form  $\mathbf{1} - \mathbf{u} \otimes \mathbf{u}/c$ , where  $c = \frac{1}{2}\mathbf{u} \cdot \mathbf{u}$ . An appropriate Householder matrix applied to a given matrix can zero all elements in a column of the matrix situated below a chosen element. Thus we arrange for the first Householder matrix  $\mathbf{Q}_0$  to zero all elements in column 0 of  $\mathbf{A}$  below the zeroth element. Similarly,  $\mathbf{Q}_1$  zeroes all elements in column 1 below element 1, and so on up to  $\mathbf{Q}_{n-2}$ . Thus

$$\mathbf{R} = \mathbf{Q}_{n-2} \cdots \mathbf{Q}_0 \cdot \mathbf{A} \quad (2.10.5)$$

Since the Householder matrices are orthogonal,

$$\mathbf{Q} = (\mathbf{Q}_{n-2} \cdots \mathbf{Q}_0)^{-1} = \mathbf{Q}_0 \cdots \mathbf{Q}_{n-2} \quad (2.10.6)$$

In many applications  $\mathbf{Q}$  is not needed explicitly, and it is sufficient to store only the factored form (2.10.6). (We do, however, store  $\mathbf{Q}$ , or rather its transpose, in the code below.) Pivoting is not usually necessary unless the matrix  $\mathbf{A}$  is very close to singular. A general  $QR$  algorithm for rectangular matrices including pivoting is given in [1]. For square matrices and without pivoting, an implementation is as follows:

```
struct QRdcmp {
    Object for QR decomposition of a matrix A, and related functions.
    Int n;
    MatDoub qt, r;                                Stored QT and R.
    Bool sing;                                     Indicates whether A is singular.
    QRdcmp(MatDoub_I &a);                        Constructor from A.
    void solve(VecDoub_I &b, VecDoub_0 &x);      Solve A · x = b for x.
    void qtmult(VecDoub_I &b, VecDoub_0 &x);     Multiply QT · b = x.
    void rsolve(VecDoub_I &b, VecDoub_0 &x);      Solve R · x = b for x.
    void update(VecDoub_I &u, VecDoub_I &v);       See next subsection.
    void rotate(const Int i, const Doub a, const Doub b); Used by update.
};
```

qrdfcmp.h

As usual, the constructor performs the actual decomposition:

```
QRdcmp::QRdcmp(MatDoub_I &a)
    : n(a.nrows()), qt(n,n), r(a), sing(false) {
Construct the QR decomposition of a[0..n-1][0..n-1]. The upper triangular matrix R and
the transpose of the orthogonal matrix Q are stored. sing is set to true if a singularity is
encountered during the decomposition, but the decomposition is still completed in this case;
otherwise it is set to false.
    Int i,j,k;
    VecDoub c(n), d(n);
    Doub scale,sigma,sum,tau;
```

qrdfcmp.h

```

for (k=0;k<n-1;k++) {
    scale=0.0;
    for (i=k;i<n;i++) scale=MAX(scale,abs(r[i][k]));
    if (scale == 0.0) {                                Singular case.
        sing=true;
        c[k]=d[k]=0.0;
    } else {                                         Form  $\mathbf{Q}_k$  and  $\mathbf{Q}_k \cdot \mathbf{A}$ .
        for (i=k;i<n;i++) r[i][k] /= scale;
        for (sum=0.0,i=k;i<n;i++) sum += SQR(r[i][k]);
        sigma=SIGN(sqrt(sum),r[k][k]);
        r[k][k] += sigma;
        c[k]=sigma*r[k][k];
        d[k] = -scale*sigma;
        for (j=k+1;j<n;j++) {
            for (sum=0.0,i=k;i<n;i++) sum += r[i][k]*r[i][j];
            tau=sum/c[k];
            for (i=k;i<n;i++) r[i][j] -= tau*r[i][k];
        }
    }
}
d[n-1]=r[n-1][n-1];
if (d[n-1] == 0.0) sing=true;
for (i=0;i<n;i++) {                                Form  $\mathbf{Q}^T$  explicitly.
    for (j=0;j<n;j++) qt[i][j]=0.0;
    qt[i][i]=1.0;
}
for (k=0;k<n-1;k++) {
    if (c[k] != 0.0) {
        for (j=0;j<n;j++) {
            sum=0.0;
            for (i=k;i<n;i++)
                sum += r[i][k]*qt[i][j];
            sum /= c[k];
            for (i=k;i<n;i++)
                qt[i][j] -= sum*r[i][k];
        }
    }
}
for (i=0;i<n;i++) {                                Form  $\mathbf{R}$  explicitly.
    r[i][i]=d[i];
    for (j=0;j<i;j++) r[i][j]=0.0;
}
}

```

The next set of member functions is used to solve linear systems. In many applications only the part (2.10.4) of the algorithm is needed, so we put in separate routines the multiplication  $\mathbf{Q}^T \cdot \mathbf{b}$  and the backsubstitution on  $\mathbf{R}$ .

qrdfcmp.h

```
void QRdfcmp::solve(VecDoub_I &b, VecDoub_0 &x) {
Solve the set of n linear equations  $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ . b[0..n-1] is input as the right-hand side vector,
and x[0..n-1] is returned as the solution vector.
```

```
    qtmult(b,x);          Form  $\mathbf{Q}^T \cdot \mathbf{b}$ .
    rsolve(x,x);          Solve  $\mathbf{R} \cdot \mathbf{x} = \mathbf{Q}^T \cdot \mathbf{b}$ .
}
```

```
void QRdfcmp::qtmult(VecDoub_I &b, VecDoub_0 &x) {
```

Multiply  $\mathbf{Q}^T \cdot \mathbf{b}$  and put the result in x. Since  $\mathbf{Q}$  is orthogonal, this is equivalent to solving  $\mathbf{Q} \cdot \mathbf{x} = \mathbf{b}$  for x.

```
    Int i,j;
    Doub sum;
    for (i=0;i<n;i++) {
        sum = 0.;
```

```

        for (j=0;j<n;j++) sum += qt[i][j]*b[j];
        x[i] = sum;
    }

void QRdcmp::rsolve(VecDoub_I &b, VecDoub_O &x) {
Solve the triangular set of n linear equations  $\mathbf{R} \cdot \mathbf{x} = \mathbf{b}$ . b[0..n-1] is input as the right-hand side vector, and x[0..n-1] is returned as the solution vector.
    Int i,j;
    Doub sum;
    if (sing) throw("attempting solve in a singular QR");
    for (i=n-1;i>=0;i--) {
        sum=b[i];
        for (j=i+1;j<n;j++) sum -= r[i][j]*x[j];
        x[i]=sum/r[i][i];
    }
}

```

See [2] for details on how to use  $QR$  decomposition for constructing orthogonal bases, and for solving least-squares problems. (We prefer to use SVD, §2.6, for these purposes, because of its greater diagnostic capability in pathological cases.)

### 2.10.1 Updating a QR decomposition

Some numerical algorithms involve solving a succession of linear systems each of which differs only slightly from its predecessor. Instead of doing  $O(N^3)$  operations each time to solve the equations from scratch, one can often update a matrix factorization in  $O(N^2)$  operations and use the new factorization to solve the next set of linear equations. The  $LU$  decomposition is complicated to update because of pivoting. However,  $QR$  turns out to be quite simple for a very common kind of update,

$$\mathbf{A} \rightarrow \mathbf{A} + \mathbf{s} \otimes \mathbf{t} \quad (2.10.7)$$

(compare equation 2.7.1). In practice it is more convenient to work with the equivalent form

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \quad \rightarrow \quad \mathbf{A}' = \mathbf{Q}' \cdot \mathbf{R}' = \mathbf{Q} \cdot (\mathbf{R} + \mathbf{u} \otimes \mathbf{v}) \quad (2.10.8)$$

One can go back and forth between equations (2.10.7) and (2.10.8) using the fact that  $\mathbf{Q}$  is orthogonal, giving

$$\mathbf{t} = \mathbf{v} \quad \text{and either} \quad \mathbf{s} = \mathbf{Q} \cdot \mathbf{u} \quad \text{or} \quad \mathbf{u} = \mathbf{Q}^T \cdot \mathbf{s} \quad (2.10.9)$$

The algorithm [2] has two phases. In the first we apply  $N - 1$  Jacobi rotations (§11.1) to reduce  $\mathbf{R} + \mathbf{u} \otimes \mathbf{v}$  to upper Hessenberg form. Another  $N - 1$  Jacobi rotations transform this upper Hessenberg matrix to the new upper triangular matrix  $\mathbf{R}'$ . The matrix  $\mathbf{Q}'$  is simply the product of  $\mathbf{Q}$  with the  $2(N - 1)$  Jacobi rotations. In applications we usually want  $\mathbf{Q}^T$ , so the algorithm is arranged to work with this matrix (which is stored in the QRdcmp object) instead of with  $\mathbf{Q}$ .

```

void QRdcmp::update(VecDoub_I &u, VecDoub_I &v) {
Starting from the stored QR decomposition  $\mathbf{A} = \mathbf{Q} \cdot \mathbf{R}$ , update it to be the QR decomposition
of the matrix  $\mathbf{Q} \cdot (\mathbf{R} + \mathbf{u} \otimes \mathbf{v})$ . Input quantities are u[0..n-1], and v[0..n-1].
    Int i,k;
    VecDoub w(u);
    for (k=n-1;k>=0;k--)
        if (w[k] != 0.0) break;
    if (k < 0) k=0;
    for (i=k-1;i>=0;i--) {
        rotate(i,w[i],-w[i+1]);
        if (w[i] == 0.0)
            w[i]=abs(w[i+1]);
    }
}

```

qrdfmp.h

```

    else if (abs(w[i]) > abs(w[i+1]))
        w[i]=abs(w[i])*sqrt(1.0+SQR(w[i+1]/w[i]));
    else w[i]=abs(w[i+1])*sqrt(1.0+SQR(w[i]/w[i+1]));
}
for (i=0;i<n;i++) r[0][i] += w[0]*v[i];
for (i=0;i<k;i++)                                     Transform upper Hessenberg matrix to upper tri-
    rotate(i,r[i][i],-r[i+1][i]);                      angular.
for (i=0;i<n;i++)
    if (r[i][i] == 0.0) sing=true;
}

void QRdcmp::rotate(const Int i, const Doub a, const Doub b)
Utility used by update. Given matrices r[0..n-1][0..n-1] and qt[0..n-1][0..n-1], carry
out a Jacobi rotation on rows i and i + 1 of each matrix. a and b are the parameters of the
rotation:  $\cos \theta = a/\sqrt{a^2 + b^2}$ ,  $\sin \theta = b/\sqrt{a^2 + b^2}$ .
{
    Int j;
    Doub c,fact,s,w,y;
    if (a == 0.0) {                                     Avoid unnecessary overflow or underflow.
        c=0.0;
        s=(b >= 0.0 ? 1.0 : -1.0);
    } else if (abs(a) > abs(b)) {
        fact=b/a;
        c=SIGN(1.0/sqrt(1.0+(fact*fact)),a);
        s=fact*c;
    } else {
        fact=a/b;
        s=SIGN(1.0/sqrt(1.0+(fact*fact)),b);
        c=fact*s;
    }
    for (j=i;j<n;j++)                               Premultiply r by Jacobi rotation.
        y=r[i][j];
        w=r[i+1][j];
        r[i][j]=c*y-s*w;
        r[i+1][j]=s*y+c*w;
    }
    for (j=0;j<n;j++)                               Premultiply qt by Jacobi rotation.
        y=qt[i][j];
        w=qt[i+1][j];
        qt[i][j]=c*y-s*w;
        qt[i+1][j]=s*y+c*w;
    }
}

```

We will make use of *QR* decomposition, and its updating, in §9.7.

#### CITED REFERENCES AND FURTHER READING:

- Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer), Chapter I/8.[1]
- Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §5.2, §5.3, §12.5.[2]

## 2.11 Is Matrix Inversion an $N^3$ Process?

We close this chapter with a little entertainment, a bit of algorithmic prestidigitation that probes more deeply into the subject of matrix inversion. We start with a seemingly simple question:

How many individual multiplications does it take to perform the matrix multiplication of two  $2 \times 2$  matrices,

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \cdot \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} \quad (2.11.1)$$

Eight, right? Here they are written explicitly:

$$\begin{aligned} c_{00} &= a_{00} \times b_{00} + a_{01} \times b_{10} \\ c_{01} &= a_{00} \times b_{01} + a_{01} \times b_{11} \\ c_{10} &= a_{10} \times b_{00} + a_{11} \times b_{10} \\ c_{11} &= a_{10} \times b_{01} + a_{11} \times b_{11} \end{aligned} \quad (2.11.2)$$

Do you think that one can write formulas for the  $c$ 's that involve only *seven* multiplications? (Try it yourself, before reading on.)

Such a set of formulas was, in fact, discovered by Strassen [1]. The formulas are

$$\begin{aligned} Q_0 &\equiv (a_{00} + a_{11}) \times (b_{00} + b_{11}) \\ Q_1 &\equiv (a_{10} + a_{11}) \times b_{00} \\ Q_2 &\equiv a_{00} \times (b_{01} - b_{11}) \\ Q_3 &\equiv a_{11} \times (-b_{00} + b_{10}) \\ Q_4 &\equiv (a_{00} + a_{01}) \times b_{11} \\ Q_5 &\equiv (-a_{00} + a_{10}) \times (b_{00} + b_{01}) \\ Q_6 &\equiv (a_{01} - a_{11}) \times (b_{10} + b_{11}) \end{aligned} \quad (2.11.3)$$

in terms of which

$$\begin{aligned} c_{00} &= Q_0 + Q_3 - Q_4 + Q_6 \\ c_{10} &= Q_1 + Q_3 \\ c_{01} &= Q_2 + Q_4 \\ c_{11} &= Q_0 + Q_2 - Q_1 + Q_5 \end{aligned} \quad (2.11.4)$$

What's the use of this? There is one fewer multiplication than in equation (2.11.2), but *many more* additions and subtractions. It is not clear that anything has been gained. But notice that in (2.11.3) the  $a$ 's and  $b$ 's are never commuted. Therefore (2.11.3) and (2.11.4) are valid when the  $a$ 's and  $b$ 's are themselves matrices. The problem of multiplying two very large matrices (of order  $N = 2^m$  for some integer  $m$ ) can now be broken down recursively by partitioning the matrices into quarters, sixteenths, etc. And note the key point: The savings is not just a factor "7/8"; it is that factor at *each* hierarchical level of the recursion. In total it reduces the process of matrix multiplication to order  $N^{\log_2 7}$  instead of  $N^3$ .

What about all the extra additions in (2.11.3) – (2.11.4)? Don't they outweigh the advantage of the fewer multiplications? For large  $N$ , it turns out that there are six times as many additions as multiplications implied by (2.11.3) – (2.11.4). But, if  $N$  is very large, this constant factor is no match for the change in the *exponent* from  $N^3$  to  $N^{\log_2 7}$ .

With this “fast” matrix multiplication, Strassen also obtained a surprising result for matrix inversion [1]. Suppose that the matrices

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} \quad (2.11.5)$$

are inverses of each other. Then the  $c$ ’s can be obtained from the  $a$ ’s by the following operations (compare equations 2.7.11 and 2.7.25):

$$\begin{aligned} R_0 &= \text{Inverse}(a_{00}) \\ R_1 &= a_{10} \times R_0 \\ R_2 &= R_0 \times a_{01} \\ R_3 &= a_{10} \times R_2 \\ R_4 &= R_3 - a_{11} \\ R_5 &= \text{Inverse}(R_4) \\ c_{01} &= R_2 \times R_5 \\ c_{10} &= R_5 \times R_1 \\ R_6 &= R_2 \times c_{10} \\ c_{00} &= R_0 - R_6 \\ c_{11} &= -R_5 \end{aligned} \quad (2.11.6)$$

In (2.11.6) the “inverse” operator occurs just twice. It is to be interpreted as the reciprocal if the  $a$ ’s and  $c$ ’s are scalars, but as matrix inversion if the  $a$ ’s and  $c$ ’s are themselves submatrices. Imagine doing the inversion of a very large matrix, of order  $N = 2^m$ , recursively by partitions in half. At each step, halving the order *doubles* the number of inverse operations. But this means that there are only  $N$  divisions in all! So divisions don’t dominate in the recursive use of (2.11.6). Equation (2.11.6) is dominated, in fact, by its 6 multiplications. Since these can be done by an  $N^{\log_2 7}$  algorithm, so can the matrix inversion!

This is fun, but let’s look at practicalities: If you estimate how large  $N$  has to be before the difference between exponent 3 and exponent  $\log_2 7 = 2.807$  is substantial enough to outweigh the bookkeeping overhead, arising from the complicated nature of the recursive Strassen algorithm, you will find that  $LU$  decomposition is in no immediate danger of becoming obsolete. However, the fast matrix multiplication routine itself is beginning to appear in libraries like BLAS, where it is typically used for  $N \gtrsim 100$ .

Strassen’s original result for matrix multiplication has been steadily improved. The fastest currently known algorithm [2] has an asymptotic order of  $N^{2.376}$ , but it is not likely to be practical to implement it.

If you like this kind of fun, then try these: (1) Can you multiply the complex numbers  $(a+ib)$  and  $(c+id)$  in only *three* real multiplications? [Answer: See §5.5.] (2) Can you evaluate a general fourth-degree polynomial in  $x$  for many different values of  $x$  with only *three* multiplications per evaluation? [Answer: See §5.1.]

#### CITED REFERENCES AND FURTHER READING:

Strassen, V. 1969, “Gaussian Elimination Is Not Optimal,” *Numerische Mathematik*, vol. 13, pp. 354–356.[1]

- Coppersmith, D., and Winograd, S. 1990, "Matrix Multiplications via Arithmetic Progressions," *Journal of Symbolic Computation*, vol. 9, pp. 251–280.[2]
- Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley).
- Winograd, S. 1971, "On the Multiplication of 2 by 2 Matrices," *Linear Algebra and Its Applications*, vol. 4, pp. 381–388.
- Pan, V. Ya. 1980, "New Fast Algorithms for Matrix Operations," *SIAM Journal on Computing*, vol. 9, pp. 321–342.
- Pan, V. 1984, *How to Multiply Matrices Faster*, Lecture Notes in Computer Science, vol. 179 (New York: Springer)
- Pan, V. 1984, "How Can We Speed Up Matrix Multiplication?", *SIAM Review*, vol. 26, pp. 393–415.

# Interpolation and Extrapolation

---

## 3.0 Introduction

We sometimes know the value of a function  $f(x)$  at a set of points  $x_0, x_1, \dots, x_{N-1}$  (say, with  $x_0 < \dots < x_{N-1}$ ), but we don't have an analytic expression for  $f(x)$  that lets us calculate its value at an arbitrary point. For example, the  $f(x_i)$ 's might result from some physical measurement or from long numerical calculation that cannot be cast into a simple functional form. Often the  $x_i$ 's are equally spaced, but not necessarily.

The task now is to estimate  $f(x)$  for arbitrary  $x$  by, in some sense, drawing a smooth curve through (and perhaps beyond) the  $x_i$ . If the desired  $x$  is in between the largest and smallest of the  $x_i$ 's, the problem is called *interpolation*; if  $x$  is outside that range, it is called *extrapolation*, which is considerably more hazardous (as many former investment analysts can attest).

Interpolation and extrapolation schemes must model the function, between or beyond the known points, by some plausible functional form. The form should be sufficiently general so as to be able to approximate large classes of functions that might arise in practice. By far most common among the functional forms used are polynomials (§3.2). Rational functions (quotients of polynomials) also turn out to be extremely useful (§3.4). Trigonometric functions, sines and cosines, give rise to *trigonometric interpolation* and related Fourier methods, which we defer to Chapters 12 and 13.

There is an extensive mathematical literature devoted to theorems about what sort of functions can be well approximated by which interpolating functions. These theorems are, alas, almost completely useless in day-to-day work: If we know enough about our function to apply a theorem of any power, we are usually not in the pitiful state of having to interpolate on a table of its values!

Interpolation is related to, but distinct from, *function approximation*. That task consists of finding an approximate (but easily computable) function to use in place of a more complicated one. In the case of interpolation, you are given the function  $f$  at points *not of your own choosing*. For the case of function approximation, you are allowed to compute the function  $f$  at *any* desired points for the purpose of developing

your approximation. We deal with function approximation in Chapter 5.

One can easily find pathological functions that make a mockery of any interpolation scheme. Consider, for example, the function

$$f(x) = 3x^2 + \frac{1}{\pi^4} \ln [(\pi - x)^2] + 1 \quad (3.0.1)$$

which is well-behaved everywhere except at  $x = \pi$ , very mildly singular at  $x = \pi$ , and otherwise takes on all positive and negative values. Any interpolation based on the values  $x = 3.13, 3.14, 3.15, 3.16$ , will assuredly get a very wrong answer for the value  $x = 3.1416$ , even though a graph plotting those five points looks really quite smooth! (Try it.)

Because pathologies can lurk anywhere, it is highly desirable that an interpolation and extrapolation routine should provide an estimate of its own error. Such an error estimate can never be foolproof, of course. We could have a function that, for reasons known only to its maker, takes off wildly and unexpectedly between two tabulated points. Interpolation always presumes some degree of smoothness for the function interpolated, but within this framework of presumption, deviations from smoothness can be detected.

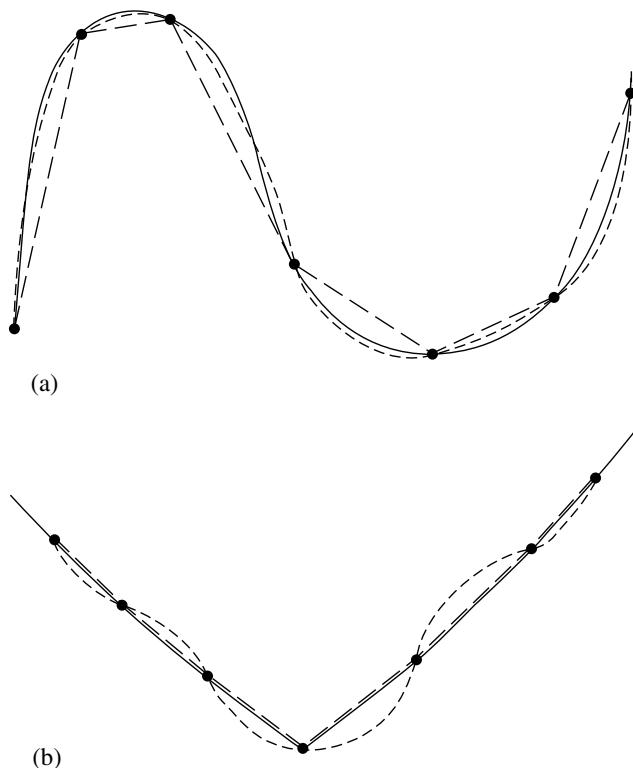
Conceptually, the interpolation process has two stages: (1) Fit (once) an interpolating function to the data points provided. (2) Evaluate (as many times as you wish) that interpolating function at a target point  $x$ .

However, this two-stage method is usually not the best way to proceed in practice. Typically it is computationally less efficient, and more susceptible to roundoff error, than methods that construct a functional estimate  $f(x)$  directly from the  $N$  tabulated values every time one is desired. Many practical schemes start at a nearby point  $f(x_i)$ , and then add a sequence of (hopefully) decreasing corrections, as information from other nearby  $f(x_i)$ 's is incorporated. The procedure typically takes  $O(M^2)$  operations, where  $M \ll N$  is the number of local points used. If everything is well behaved, the last correction will be the smallest, and it can be used as an informal (though not rigorous) bound on the error. In schemes like this, we might also say that there are two stages, but now they are: (1) Find the right starting position in the table ( $x_i$  or  $i$ ). (2) Perform the interpolation using  $M$  nearby values (for example, centered on  $x_i$ ).

In the case of polynomial interpolation, it sometimes does happen that the coefficients of the interpolating polynomial are of interest, even though their use in evaluating the interpolating function should be frowned on. We deal with this possibility in §3.5.

Local interpolation, using  $M$  nearest-neighbor points, gives interpolated values  $f(x)$  that do not, in general, have continuous first or higher derivatives. That happens because, as  $x$  crosses the tabulated values  $x_i$ , the interpolation scheme switches which tabulated points are the “local” ones. (If such a switch is allowed to occur anywhere else, then there will be a discontinuity in the interpolated function itself at that point. Bad idea!)

In situations where continuity of derivatives is a concern, one must use the “stiffer” interpolation provided by a so-called *spline* function. A spline is a polynomial between each pair of table points, but one whose coefficients are determined “slightly” nonlocally. The nonlocality is designed to guarantee global smoothness in the interpolated function up to some order of derivative. Cubic splines (§3.3) are the



**Figure 3.0.1.** (a) A smooth function (solid line) is more accurately interpolated by a high-order polynomial (shown schematically as dotted line) than by a low-order polynomial (shown as a piecewise linear dashed line). (b) A function with sharp corners or rapidly changing higher derivatives is *less* accurately approximated by a high-order polynomial (dotted line), which is too “stiff,” than by a low-order polynomial (dashed lines). Even some smooth functions, such as exponentials or rational functions, can be badly approximated by high-order polynomials.

most popular. They produce an interpolated function that is continuous through the second derivative. Splines tend to be stabler than polynomials, with less possibility of wild oscillation between the tabulated points.

The number  $M$  of points used in an interpolation scheme, minus 1, is called the *order* of the interpolation. Increasing the order does not necessarily increase the accuracy, especially in polynomial interpolation. If the added points are distant from the point of interest  $x$ , the resulting higher-order polynomial, with its additional constrained points, tends to oscillate wildly between the tabulated values. This oscillation may have no relation at all to the behavior of the “true” function (see Figure 3.0.1). Of course, adding points *close* to the desired point usually does help, but a finer mesh implies a larger table of values, which is not always available.

For polynomial interpolation, it turns out that the *worst* possible arrangement of the  $x_i$ 's is for them to be equally spaced. Unfortunately, this is by far the most common way that tabulated data are gathered or presented. High-order polynomial interpolation on equally spaced data is *ill-conditioned*: small changes in the data can give large differences in the oscillations between the points. The disease is particularly bad if you are interpolating on values of an analytic function that has poles in

the complex plane lying inside a certain oval region whose major axis is the  $M$ -point interval. But even if you have a function with no nearby poles, roundoff error can, in effect, create nearby poles and cause big interpolation errors. In §5.8 we will see that these issues go away if you are allowed to choose an optimal set of  $x_i$ 's. But when you are handed a table of function values, that option is not available.

As the order is increased, it is typical for interpolation error to decrease at first, but only up to a certain point. Larger orders result in the error exploding.

For the reasons mentioned, it is a good idea to be cautious about high-order interpolation. We can enthusiastically endorse polynomial interpolation with 3 or 4 points; we are perhaps tolerant of 5 or 6; but we rarely go higher than that unless there is quite rigorous monitoring of estimated errors. Most of the interpolation methods in this chapter are applied *piecewise* using only  $M$  points at a time, so that the order is a fixed value  $M - 1$ , no matter how large  $N$  is. As mentioned, *splines* (§3.3) are a special case where the function and various derivatives are required to be continuous from one interval to the next, but the order is nevertheless held fixed at a small value (usually 3).

In §3.4 we discuss *rational function interpolation*. In many, but not all, cases, rational function interpolation is more robust, allowing higher orders to give higher accuracy. The standard algorithm, however, allows poles on the real axis or nearby in the complex plane. (This is not necessarily bad: You may be trying to approximate a function with such poles.) A newer method, *barycentric rational interpolation* (§3.4.1) suppresses all nearby poles. This is the only method in this chapter for which we might actually encourage experimentation with high order (say,  $> 6$ ). Barycentric rational interpolation competes very favorably with splines: its error is often smaller, and the resulting approximation is infinitely smooth (unlike splines).

The interpolation methods below are also methods for extrapolation. An important application, in Chapter 17, is their use in the integration of ordinary differential equations. There, considerable care is taken with the monitoring of errors. Otherwise, the dangers of extrapolation cannot be overemphasized: An interpolating function, which is *perforce* an extrapolating function, will typically go berserk when the argument  $x$  is outside the range of tabulated values by more (and often significantly less) than the typical spacing of tabulated points.

Interpolation can be done in more than one dimension, e.g., for a function  $f(x, y, z)$ . Multidimensional interpolation is often accomplished by a sequence of one-dimensional interpolations, but there are also other techniques applicable to scattered data. We discuss multidimensional methods in §3.6 – §3.8.

#### CITED REFERENCES AND FURTHER READING:

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, §25.2.
- Ueberhuber, C.W. 1997, *Numerical Computation: Methods, Software, and Analysis*, vol. 1 (Berlin: Springer), Chapter 9.
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 2.
- Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), Chapter 3.
- Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 5.

- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), Chapter 3.
- Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), Chapter 6.

## 3.1 Preliminaries: Searching an Ordered Table

We want to define an interpolation object that knows everything about interpolation except one thing — how to actually interpolate! Then we can plug mathematically different interpolation methods into the object to get different objects sharing a common user interface. A key task common to all objects in this framework is finding your place in the table of  $x_i$ 's, given some particular value  $x$  at which the function evaluation is desired. It is worth some effort to do this efficiently; otherwise you can easily spend more time searching the table than doing the actual interpolation.

Our highest-level object for one-dimensional interpolation is an abstract base class containing just one function intended to be called by the user: `interp(x)` returns the interpolated function value at  $x$ . The base class “promises,” by declaring a virtual function `rawinterp(jlo,x)`, that every derived interpolation class will provide a method for local interpolation when given an appropriate local starting point in the table, an offset  $jlo$ . Interfacing between `interp` and `rawinterp` must thus be a method for calculating  $jlo$  from  $x$ , that is, for searching the table. In fact, we will use two such methods.

`interp_1d.h`

```
struct Base_interp
Abstract base class used by all interpolation routines in this chapter. Only the routine interp is called directly by the user.
```

```
{
    Int n, mm, jsav, cor, dj;
    const Doub *xx, *yy;
    Base_interp(VecDoub_I &x, const Doub *y, Int m)
Constructor: Set up for interpolating on a table of x's and y's of length m. Normally called by a derived class, not by the user.
    : n(x.size()), mm(m), jsav(0), cor(0), xx(&x[0]), yy(y) {
        dj = MIN(1,(int)pow((Doub)n,0.25));
    }
```

```
Doub interp(Doub x) {
Given a value x, return an interpolated value, using data pointed to by xx and yy.
    Int jlo = cor ? hunt(x) : locate(x);
    return rawinterp(jlo,x);
}
```

```
Int locate(const Doub x);           See definitions below.
Int hunt(const Doub x);
```

```
Doub virtual rawinterp(Int jlo, Doub x) = 0;
Derived classes provide this as the actual interpolation method.
```

```
};
```

Formally, the problem is this: Given an array of abscissas  $x_j$ ,  $j = 0, \dots, N - 1$ , with the abscissas either monotonically increasing or monotonically decreasing, and given an integer  $M \leq N$ , and a number  $x$ , find an integer  $j_{lo}$  such that  $x$  is centered

among the  $M$  abscissas  $x_{j_0}, \dots, x_{j_0+M-1}$ . By centered we mean that  $x$  lies between  $x_m$  and  $x_{m+1}$  insofar as possible, where

$$m = j_0 + \left\lfloor \frac{M-2}{2} \right\rfloor \quad (3.1.1)$$

By “insofar as possible” we mean that  $j_0$  should never be less than zero, nor should  $j_0 + M - 1$  be greater than  $N - 1$ .

In most cases, when all is said and done, it is hard to do better than *bisection*, which will find the right place in the table in about  $\log_2 N$  tries.

#### `Int Base_interp::locate(const Doub x)`

Given a value  $x$ , return a value  $j$  such that  $x$  is (insofar as possible) centered in the subrange  $xx[j..j+mm-1]$ , where  $xx$  is the stored pointer. The values in  $xx$  must be monotonic, either increasing or decreasing. The returned value is not less than 0, nor greater than  $n-1$ .

{

```
    Int ju,jm,jl;
    if (n < 2 || mm < 2 || mm > n) throw("locate size error");
    Bool ascnd=(xx[n-1] >= xx[0]);      True if ascending order of table, false otherwise.
    jl=0;
    ju=n-1;
    while (ju-jl > 1) {
        jm = (ju+jl) >> 1;
        if (x >= xx[jm] == ascnd)
            jl=jm;
        else
            ju=jm;
    }
    cor = abs(jl-jsav) > dj ? 0 : 1;   Decide whether to use hunt or locate next time.
    jsav = jl;
    return MAX(0,MIN(n-mm,jl-((mm-2)>>1)));
}
```

interp\_1d.h

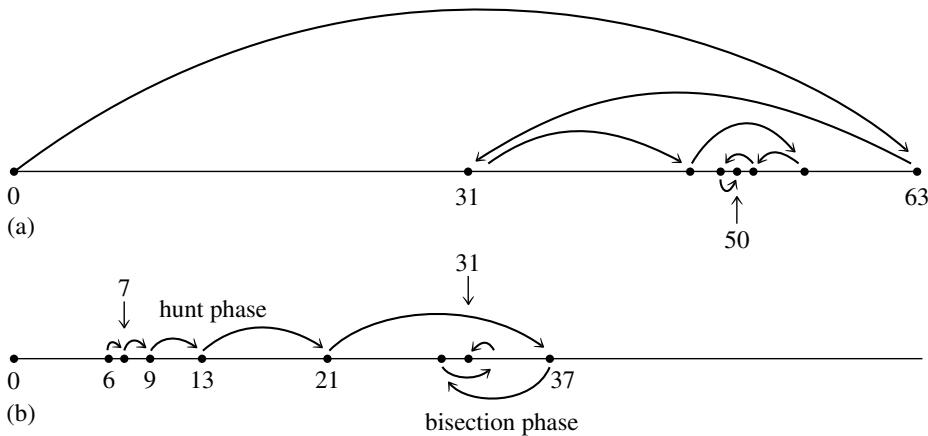
The above `locate` routine accesses the array of values `xx[]` via a pointer stored by the base class. This rather primitive method of access, avoiding the use of a higher-level vector class like `VecDoub`, is here preferable for two reasons: (1) It’s usually faster; and (2) for two-dimensional interpolation, we will later need to point directly into a row of a matrix. The peril of this design choice is that it assumes that consecutive values of a vector are stored consecutively, and similarly for consecutive values of a single row of a matrix. See discussion in §1.4.2.

### 3.1.1 Search with Correlated Values

Experience shows that in many, perhaps even most, applications, interpolation routines are called with nearly identical abscissas on consecutive searches. For example, you may be generating a function that is used on the right-hand side of a differential equation: Most differential equation integrators, as we shall see in Chapter 17, call for right-hand side evaluations at points that hop back and forth a bit, but whose trend moves slowly in the direction of the integration.

In such cases it is wasteful to do a full bisection, *ab initio*, on each call. Much more desirable is to give our base class a tiny bit of intelligence: If it sees two calls that are “close,” it anticipates that the next call will also be. Of course, there must not be too big a penalty if it anticipates wrongly.

The `hunt` method starts with a guessed position in the table. It first “hunts,” either up or down, in increments of 1, then 2, then 4, etc., until the desired value is bracketed. It then bisects in the bracketed interval. At worst, this routine is about a



**Figure 3.1.1.** Finding a table entry by bisection. Shown here is the sequence of steps that converge to element 50 in a table of length 64. (b) The routine `hunt` searches from a previous known position in the table by increasing steps and then converges by bisection. Shown here is a particularly unfavorable example, converging to element 31 from element 6. A favorable example would be convergence to an element near 6, such as 8, which would require just three “hops.”

factor of 2 slower than `locate` above (if the hunt phase expands to include the whole table). At best, it can be a factor of  $\log_2 n$  faster than `locate`, if the desired point is usually quite close to the input guess. Figure 3.1.1 compares the two routines.

#### interp\_1d.h

##### `Int Base_interp::hunt(const Doub x)`

Given a value  $x$ , return a value  $j$  such that  $x$  is (insofar as possible) centered in the subrange  $xx[j..j+mm-1]$ , where  $xx$  is the stored pointer. The values in  $xx$  must be monotonic, either increasing or decreasing. The returned value is not less than 0, nor greater than  $n-1$ .

```

{
    Int jl=jsav, jm, ju, inc=1;
    if (n < 2 || mm < 2 || mm > n) throw("hunt size error");
    Bool ascnd=(xx[n-1] >= xx[0]);      True if ascending order of table, false otherwise.
    if (jl < 0 || jl > n-1) {            Input guess not useful. Go immediately to bisection.
        jl=0;
        ju=n-1;
    } else {
        if (x >= xx[jl] == ascnd) {      Hunt up:
            for (;;) {
                ju = jl + inc;
                if (ju >= n-1) { ju = n-1; break;}          Off end of table.
                else if (x < xx[ju] == ascnd) break;        Found bracket.
                else {                                     Not done, so double the increment and try again.
                    jl = ju;
                    inc += inc;
                }
            }
        } else {                            Hunt down:
            ju = jl;
            for (;;) {
                jl = jl - inc;
                if (jl <= 0) { jl = 0; break;}          Off end of table.
                else if (x >= xx[jl] == ascnd) break;        Found bracket.
                else {                                     Not done, so double the increment and try again.
                    ju = jl;
                    inc += inc;
                }
            }
        }
    }
}
```

```

        }
    }
}
while (ju-jl > 1)           Hunt is done, so begin the final bisection phase:
    jm = (ju+jl) >> 1;
    if (x >= xx[jm] == ascnd)
        jl=jm;
    else
        ju=jm;
}
cor = abs(jl-jsav) > dj ? 0 : 1;      Decide whether to use hunt or locate next
jsav = jl;                           time.
return MAX(0,MIN(n-mm,jl-((mm-2)>>1)));
}

```

The methods `locate` and `hunt` each update the boolean variable `cor` in the base class, indicating whether consecutive calls seem correlated. That variable is then used by `interp` to decide whether to use `locate` or `hunt` on the next call. This is all invisible to the user, of course.

### 3.1.2 Example: Linear Interpolation

You may think that, at this point, we have wandered far from the subject of interpolation methods. To show that we are actually on track, here is a class that efficiently implements piecewise linear interpolation.

```
struct Linear_interp : Base_interp
Piecewise linear interpolation object. Construct with x and y vectors, then call interp for
interpolated values.
{
    Linear_interp(VecDoub_I &xv, VecDoub_I &yy)
        : Base_interp(xv,&yy[0],2) {}
    Doub rawinterp(Int j, Doub x) {
        if (xx[j]==xx[j+1]) return yy[j];      Table is defective, but we can recover.
        else return yy[j] + ((x-xx[j])/xx[j+1]-xx[j]))*(yy[j+1]-yy[j]);
    }
};
```

interp\_linear.h

You construct a linear interpolation object by declaring an instance with your filled vectors of abscissas  $x_i$  and function values  $y_i = f(x_i)$ ,

```
Int n=....;
VecDoub xx(n), yy(n);
...
Linear_interp myfunc(xx,yy);
```

Behind the scenes, the base class constructor is called with  $M = 2$  because linear interpolation uses just the two points bracketing a value. Also, pointers to the data are saved. (You must ensure that the vectors `xx` and `yy` don't go out of scope while `myfunc` is in use.)

When you want an interpolated value, it's as simple as

```
Doub x,y;
...
y = myfunc.interp(x);
```

If you have several functions that you want to interpolate, you declare a separate instance of `Linear_interp` for each one.

We will now use the same interface for more advanced interpolation methods.

#### CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §6.2.1.

## 3.2 Polynomial Interpolation and Extrapolation

Through any two points there is a unique line. Through any three points there is a unique quadratic. *Et cetera.* The interpolating polynomial of degree  $M - 1$  through the  $M$  points  $y_0 = f(x_0), y_1 = f(x_1), \dots, y_{M-1} = f(x_{M-1})$  is given explicitly by Lagrange's classical formula,

$$\begin{aligned} P(x) &= \frac{(x - x_1)(x - x_2)\dots(x - x_{M-1})}{(x_0 - x_1)(x_0 - x_2)\dots(x_0 - x_{M-1})} y_0 \\ &\quad + \frac{(x - x_0)(x - x_2)\dots(x - x_{M-1})}{(x_1 - x_0)(x_1 - x_2)\dots(x_1 - x_{M-1})} y_1 + \dots \\ &\quad + \frac{(x - x_0)(x - x_1)\dots(x - x_{M-2})}{(x_{M-1} - x_0)(x_{M-1} - x_1)\dots(x_{M-1} - x_{M-2})} y_{M-1} \end{aligned} \quad (3.2.1)$$

There are  $M$  terms, each a polynomial of degree  $M - 1$  and each constructed to be zero at all of the  $x_i$ 's except one, at which it is constructed to be  $y_i$ .

It is not terribly wrong to implement the Lagrange formula straightforwardly, but it is not terribly right either. The resulting algorithm gives no error estimate, and it is also somewhat awkward to program. A much better algorithm (for constructing the same, unique, interpolating polynomial) is *Neville's algorithm*, closely related to and sometimes confused with *Aitken's algorithm*, the latter now considered obsolete.

Let  $P_0$  be the value at  $x$  of the unique polynomial of degree zero (i.e., a constant) passing through the point  $(x_0, y_0)$ ; so  $P_0 = y_0$ . Likewise define  $P_1, P_2, \dots, P_{M-1}$ . Now let  $P_{01}$  be the value at  $x$  of the unique polynomial of degree one passing through both  $(x_0, y_0)$  and  $(x_1, y_1)$ . Likewise  $P_{12}, P_{23}, \dots, P_{(M-2)(M-1)}$ . Similarly, for higher-order polynomials, up to  $P_{012\dots(M-1)}$ , which is the value of the unique interpolating polynomial through all  $M$  points, i.e., the desired answer. The various  $P$ 's form a "tableau" with "ancestors" on the left leading to a single "descendant" at the extreme right. For example, with  $M = 4$ ,

$$\begin{array}{lll} x_0 : & y_0 = P_0 & \\ & & P_{01} \\ x_1 : & y_1 = P_1 & P_{012} \\ & & P_{12} & P_{0123} \\ x_2 : & y_2 = P_2 & P_{123} \\ & & P_{23} \\ x_3 : & y_3 = P_3 & \end{array} \quad (3.2.2)$$

Neville's algorithm is a recursive way of filling in the numbers in the tableau a column at a time, from left to right. It is based on the relationship between a

“daughter”  $P$  and its two “parents,”

$$P_{i(i+1)\dots(i+m)} = \frac{(x - x_{i+m}) P_{i(i+1)\dots(i+m-1)} + (x_i - x) P_{(i+1)(i+2)\dots(i+m)}}{x_i - x_{i+m}} \quad (3.2.3)$$

This recurrence works because the two parents already agree at points  $x_{i+1} \dots x_{i+m-1}$ .

An improvement on the recurrence (3.2.3) is to keep track of the small *differences* between parents and daughters, namely to define (for  $m = 1, 2, \dots, M-1$ ),

$$\begin{aligned} C_{m,i} &\equiv P_{i\dots(i+m)} - P_{i\dots(i+m-1)} \\ D_{m,i} &\equiv P_{i\dots(i+m)} - P_{(i+1)\dots(i+m)}. \end{aligned} \quad (3.2.4)$$

Then one can easily derive from (3.2.3) the relations

$$\begin{aligned} D_{m+1,i} &= \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \\ C_{m+1,i} &= \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \end{aligned} \quad (3.2.5)$$

At each level  $m$ , the  $C$ ’s and  $D$ ’s are the corrections that make the interpolation one order higher. The final answer  $P_{0\dots(M-1)}$  is equal to the sum of *any*  $y_i$  plus a set of  $C$ ’s and/or  $D$ ’s that form a path through the family tree to the rightmost daughter.

Here is the class implementing polynomial interpolation or extrapolation. All of its “support infrastructure” is in the base class `Base_interp`. It needs only to provide a `rawinterp` method that contains Neville’s algorithm.

```
struct Poly_interp : Base_interp
Polynomial interpolation object. Construct with x and y vectors, and the number M of points
to be used locally (polynomial order plus one), then call interp for interpolated values.
{
    Doub dy;
    Poly_interp(VecDoub_I &xv, VecDoub_I &yv, Int m)
        : Base_interp(xv,&yv[0],m), dy(0.) {}
    Doub rawinterp(Int jl, Doub x);
};

Doub Poly_interp::rawinterp(Int jl, Doub x)
Given a value x, and using pointers to data xx and yy, this routine returns an interpolated
value y, and stores an error estimate dy. The returned value is obtained by mm-point polynomial
interpolation on the subrange xx[jl..jl+mm-1].
{
    Int i,m,ns=0;
    Doub y,den,dif,dift,ho,hp,w;
    const Doub *xa = &xx[jl], *ya = &yy[jl];
    VecDoub c(mm),d(mm);
    dif=abs(xxa[0]);
    for (i=0;i<mm;i++) {           Here we find the index ns of the closest table entry,
        if ((dift=abs(x-xa[i])) < dif) {
            ns=i;
            dif=dift;
        }
        c[i]=ya[i];                  and initialize the tableau of c's and d's.
        d[i]=ya[i];
    }
    y=ya[ns--];
    for (m=1;m<mm;m++) {          This is the initial approximation to y.
        For each column of the tableau,
```

`interp_id.h`

```

for (i=0;i<mm-m;i++) {      we loop over the current c's and d's and update
    ho=xa[i]-x;
    hp=xa[i+m]-x;
    w=c[i+1]-d[i];
    if ((den=ho-hp) == 0.0) throw("Poly_interp error");
    This error can occur only if two input xa's are (to within roundoff) identical.
    den=w/den;
    d[i]=hp*den;           Here the c's and d's are updated.
    c[i]=ho*den;
}
y += (dy=(2*(ns+1) < (mm-m) ? c[ns+1] : d[ns--]));
After each column in the tableau is completed, we decide which correction, c or d, we
want to add to our accumulating value of y, i.e., which path to take through the tableau
— forking up or down. We do this in such a way as to take the most “straight line”
route through the tableau to its apex, updating ns accordingly to keep track of where
we are. This route keeps the partial approximations centered (insofar as possible) on
the target x. The last dy added is thus the error indication.
}
return y;
}

```

The user interface to Poly\_interp is virtually the same as for Linear\_interp (end of §3.1), except that an additional argument in the constructor sets  $M$ , the number of points used (the order plus one). A cubic interpolator looks like this:

```

Int n=...;
VecDoub xx(n), yy(n);
...
Poly_interp myfunc(xx,yy,4);

```

Poly\_interp stores an error estimate dy for the most recent call to its interp function:

```

Doub x,y,err;
...
y = myfunc.interp(x);
err = myfunc.dy;

```

#### CITED REFERENCES AND FURTHER READING:

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, §25.2.
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §2.1.
- Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.

### 3.3 Cubic Spline Interpolation

Given a tabulated function  $y_i = y(x_i)$ ,  $i = 0 \dots N - 1$ , focus attention on one particular interval, between  $x_j$  and  $x_{j+1}$ . Linear interpolation in that interval gives the interpolation formula

$$y = Ay_j + By_{j+1} \quad (3.3.1)$$

where

$$A \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \quad B \equiv 1 - A = \frac{x - x_j}{x_{j+1} - x_j} \quad (3.3.2)$$

Equations (3.3.1) and (3.3.2) are a special case of the general Lagrange interpolation formula (3.2.1).

Since it is (piecewise) linear, equation (3.3.1) has zero second derivative in the interior of each interval and an undefined, or infinite, second derivative at the abscissas  $x_j$ . The goal of cubic spline interpolation is to get an interpolation formula that is smooth in the first derivative and continuous in the second derivative, both within an interval and at its boundaries.

Suppose, contrary to fact, that in addition to the tabulated values of  $y_i$ , we also have tabulated values for the function's second derivatives,  $y''$ , that is, a set of numbers  $y''_i$ . Then, within each interval, we can add to the right-hand side of equation (3.3.1) a cubic polynomial whose second derivative varies linearly from a value  $y''_j$  on the left to a value  $y''_{j+1}$  on the right. Doing so, we will have the desired continuous second derivative. If we also construct the cubic polynomial to have zero values at  $x_j$  and  $x_{j+1}$ , then adding it in will not spoil the agreement with the tabulated functional values  $y_j$  and  $y_{j+1}$  at the endpoints  $x_j$  and  $x_{j+1}$ .

A little side calculation shows that there is only one way to arrange this construction, namely replacing (3.3.1) by

$$y = Ay_j + By_{j+1} + Cy''_j + Dy''_{j+1} \quad (3.3.3)$$

where  $A$  and  $B$  are defined in (3.3.2) and

$$C \equiv \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2 \quad D \equiv \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2 \quad (3.3.4)$$

Notice that the dependence on the independent variable  $x$  in equations (3.3.3) and (3.3.4) is entirely through the linear  $x$ -dependence of  $A$  and  $B$ , and (through  $A$  and  $B$ ) the cubic  $x$ -dependence of  $C$  and  $D$ .

We can readily check that  $y''$  is in fact the second derivative of the new interpolating polynomial. We take derivatives of equation (3.3.3) with respect to  $x$ , using the definitions of  $A$ ,  $B$ ,  $C$ , and  $D$  to compute  $dA/dx$ ,  $dB/dx$ ,  $dC/dx$ , and  $dD/dx$ . The result is

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y''_j + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y''_{j+1} \quad (3.3.5)$$

for the first derivative, and

$$\frac{d^2y}{dx^2} = Ay''_j + By''_{j+1} \quad (3.3.6)$$

for the second derivative. Since  $A = 1$  at  $x_j$ ,  $A = 0$  at  $x_{j+1}$ , while  $B$  is just the other way around, (3.3.6) shows that  $y''$  is just the tabulated second derivative, and also that the second derivative will be continuous across, e.g., the boundary between the two intervals  $(x_{j-1}, x_j)$  and  $(x_j, x_{j+1})$ .

The only problem now is that we supposed the  $y''_i$ 's to be known, when, actually, they are not. However, we have not yet required that the *first* derivative, computed from equation (3.3.5), be continuous across the boundary between two intervals. The

key idea of a cubic spline is to require this continuity and to use it to get equations for the second derivatives  $y''_i$ .

The required equations are obtained by setting equation (3.3.5) evaluated for  $x = x_j$  in the interval  $(x_{j-1}, x_j)$  equal to the same equation evaluated for  $x = x_j$  but in the interval  $(x_j, x_{j+1})$ . With some rearrangement, this gives (for  $j = 1, \dots, N-2$ )

$$\frac{x_j - x_{j-1}}{6} y''_{j-1} + \frac{x_{j+1} - x_{j-1}}{3} y''_j + \frac{x_{j+1} - x_j}{6} y''_{j+1} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}} \quad (3.3.7)$$

These are  $N-2$  linear equations in the  $N$  unknowns  $y''_i, i = 0, \dots, N-1$ . Therefore there is a two-parameter family of possible solutions.

For a unique solution, we need to specify two further conditions, typically taken as boundary conditions at  $x_0$  and  $x_{N-1}$ . The most common ways of doing this are either

- set one or both of  $y''_0$  and  $y''_{N-1}$  equal to zero, giving the so-called *natural cubic spline*, which has zero second derivative on one or both of its boundaries, or
- set either of  $y''_0$  and  $y''_{N-1}$  to values calculated from equation (3.3.5) so as to make the first derivative of the interpolating function have a specified value on either or both boundaries.

Although the boundary condition for natural splines is commonly used, another possibility is to estimate the first derivatives at the endpoints from the first and last few tabulated points. For details of how to do this, see the end of §3.7. Best, of course, is if you can compute the endpoint first derivatives analytically.

One reason that cubic splines are especially practical is that the set of equations (3.3.7), along with the two additional boundary conditions, are not only linear, but also *tridiagonal*. Each  $y''_j$  is coupled only to its nearest neighbors at  $j \pm 1$ . Therefore, the equations can be solved in  $O(N)$  operations by the tridiagonal algorithm (§2.4). That algorithm is concise enough to build right into the function called by the constructor.

The object for cubic spline interpolation looks like this:

interp\_1d.h

```
struct Spline_interp : Base_interp
Cubic spline interpolation object. Construct with x and y vectors, and (optionally) values of
the first derivative at the endpoints, then call interp for interpolated values.
{
    VecDoub y2;

    Spline_interp(VecDoub_I &xv, VecDoub_I &yv, Doub yp1=1.e99, Doub ypn=1.e99)
        : Base_interp(xv,&yv[0],2), y2(xv.size())
        {sety2(&xv[0],&yv[0],yp1,ypn);}

    Spline_interp(VecDoub_I &xv, const Doub *yv, Doub yp1=1.e99, Doub ypn=1.e99)
        : Base_interp(xv,yv,2), y2(xv.size())
        {sety2(&xv[0],yv,yp1,ypn);}

    void sety2(const Doub *xv, const Doub *yv, Doub yp1, Doub ypn);
    Doub rawinterp(Int jl, Doub xv);
};
```

For now, you can ignore the second constructor; it will be used later for two-dimensional spline interpolation.

The user interface differs from previous ones only in the addition of two constructor arguments, used to set the values of the first derivatives at the endpoints,  $y'_0$  and  $y'_{N-1}$ . These are coded with default values that signal that you want a natural spline, so they can be omitted in most situations. Both constructors invoke `sety2` to do the actual work of computing, and storing, the second derivatives.

**void Spline\_interp::sety2(const Doub \*xv, const Doub \*yv, Doub yp1, Doub ypn)**  
This routine stores an array  $y2[0..n-1]$  with second derivatives of the interpolating function at the tabulated points pointed to by  $xv$ , using function values pointed to by  $yv$ . If  $yp1$  and/or  $ypn$  are equal to  $1 \times 10^{99}$  or larger, the routine is signaled to set the corresponding boundary condition for a natural spline, with zero second derivative on that boundary; otherwise, they are the values of the first derivatives at the endpoints.

```

    Int i,k;
    Doub p,qn,sig,un;
    Int n=y2.size();
    VecDoub u(n-1);
    if (yp1 > 0.99e99)
        y2[0]=u[0]=0.0;
    else {
        y2[0] = -0.5;
        u[0]=(3.0/(xv[1]-xv[0]))*((yv[1]-yv[0])/(xv[1]-xv[0])-yp1);
    }
    for (i=1;i<n-1;i++) {
        sig=(xv[i]-xv[i-1])/(xv[i+1]-xv[i-1]);
        p=sig*y2[i-1]+2.0;
        y2[i]=(sig-1.0)/p;
        u[i]=(yv[i+1]-yv[i])/(xv[i+1]-xv[i]) - (yv[i]-yv[i-1])/(xv[i]-xv[i-1]);
        u[i]=(6.0*u[i]/(xv[i+1]-xv[i-1])-sig*u[i-1])/p;
    }
    if (ypn > 0.99e99)
        qn=un=0.0;
    else {
        qn=0.5;
        un=(3.0/(xv[n-1]-xv[n-2]))*(ypn-(yv[n-1]-yv[n-2])/(xv[n-1]-xv[n-2]));
    }
    y2[n-1]=(un-qn*u[n-2])/(qn*y2[n-2]+1.0);
    for (k=n-2;k>=0;k--)
        y2[k]=y2[k]*y2[k+1]+u[k];
}

```

The lower boundary condition is set either to be “natural” or else to have a specified first derivative.

This is the decomposition loop of the tridiagonal algorithm.  $y_2$  and  $u$  are used for temporary storage of the decomposed factors.

The upper boundary condition is set either to be “natural” or else to have a specified first derivative.

This is the backsubstitution loop of the tridiagonal algorithm.

## interp\_1d.h

Note that, unlike the previous object `Poly_interp`, `Spline_interp` stores data that depend on the contents of your array of  $y_i$ 's at its time of creation — a whole vector `y2`. Although we didn't point it out, the previous interpolation object actually allowed the misuse of altering the contents of their `x` and `y` arrays on the fly (as long as the lengths didn't change). If you do that with `Spline_interp`, you'll get definitely wrong answers!

The required `rawinterp` method, never called directly by the users, uses the stored `y2` and implements equation (3.3.3):

```
Doub Spline_interp::rawinterp(Int jl, Doub x)
```

Given a value  $x$ , and using pointers to data  $xx$  and  $yy$ , and the stored vector of second derivatives  $y2$ , this routine returns the cubic spline interpolated value  $y$ .

```
{  
    Int klo=jl,khi=jl+1;  
    Doub y,h,b,a;  
    h=xx[khi]-xx[klo];  
    if (h == 0.0) throw("Bad input to routine splint"); The xa's must be dis-  
    a=(xx[khi]-x)/h; tinct
```

interp\_1d.h

```

b=(x-xx[klo])/h;                                Cubic spline polynomial is now evaluated.
y=a*yy[klo]+b*yy[khi]+((a*a*a-a)*y2[klo]
+(b*b*b-b)*y2[khi])*(h*h)/6.0;
return y;
}

```

Typical use looks like this:

```

Int n=...;
VecDoub xx(n), yy(n);
...
Spline_interp myfunc(xx,yy);

```

and then, as often as you like,

```

Doub x,y;
...
y = myfunc.interp(x);

```

Note that no error estimate is available.

#### CITED REFERENCES AND FURTHER READING:

- De Boor, C. 1978, *A Practical Guide to Splines* (New York: Springer).
- Ueberhuber, C.W. 1997, *Numerical Computation: Methods, Software, and Analysis*, vol. 1 (Berlin: Springer), Chapter 9.
- Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §4.4 – §4.5.
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §2.4.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §3.8.

## 3.4 Rational Function Interpolation and Extrapolation

Some functions are not well approximated by polynomials but *are* well approximated by rational functions, that is quotients of polynomials. We denote by  $R_{i(i+1)\dots(i+m)}$  a rational function passing through the  $m + 1$  points  $(x_i, y_i), \dots, (x_{i+m}, y_{i+m})$ . More explicitly, suppose

$$R_{i(i+1)\dots(i+m)} = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1x + \dots + p_\mu x^\mu}{q_0 + q_1x + \dots + q_\nu x^\nu} \quad (3.4.1)$$

Since there are  $\mu + \nu + 1$  unknown  $p$ 's and  $q$ 's ( $q_0$  being arbitrary), we must have

$$m + 1 = \mu + \nu + 1 \quad (3.4.2)$$

In specifying a rational function interpolating function, you must give the desired order of both the numerator and the denominator.

Rational functions are sometimes superior to polynomials, roughly speaking, because of their ability to model functions with poles, that is, zeros of the denominator of equation (3.4.1). These poles might occur for real values of  $x$ , if the function

to be interpolated itself has poles. More often, the function  $f(x)$  is finite for all finite *real*  $x$  but has an analytic continuation with poles in the complex  $x$ -plane. Such poles can themselves ruin a polynomial approximation, even one restricted to real values of  $x$ , just as they can ruin the convergence of an infinite power series in  $x$ . If you draw a circle in the complex plane around your  $m$  tabulated points, then you should not expect polynomial interpolation to be good unless the nearest pole is rather far outside the circle. A rational function approximation, by contrast, will stay “good” as long as it has enough powers of  $x$  in its denominator to account for (cancel) any nearby poles.

For the interpolation problem, a rational function is constructed so as to go through a chosen set of tabulated functional values. However, we should also mention in passing that rational function approximations can be used in analytic work. One sometimes constructs a rational function approximation by the criterion that the rational function of equation (3.4.1) itself have a power series expansion that agrees with the first  $m + 1$  terms of the power series expansion of the desired function  $f(x)$ . This is called *Padé approximation* and is discussed in §5.12.

Bulirsch and Stoer found an algorithm of the Neville type that performs rational function extrapolation on tabulated data. A tableau like that of equation (3.2.2) is constructed column by column, leading to a result and an error estimate. The Bulirsch-Stoer algorithm produces the so-called *diagonal* rational function, with the degrees of the numerator and denominator equal (if  $m$  is even) or with the degree of the denominator larger by one (if  $m$  is odd; cf. equation 3.4.2 above). For the derivation of the algorithm, refer to [1]. The algorithm is summarized by a recurrence relation exactly analogous to equation (3.2.3) for polynomial approximation:

$$\begin{aligned} R_{i(i+1)\dots(i+m)} &= R_{(i+1)\dots(i+m)} \\ &+ \frac{R_{(i+1)\dots(i+m)} - R_{i\dots(i+m-1)}}{\left(\frac{x-x_i}{x-x_{i+m}}\right)\left(1 - \frac{R_{(i+1)\dots(i+m)} - R_{i\dots(i+m-1)}}{R_{(i+1)\dots(i+m)} - R_{(i+1)\dots(i+m-1)}}\right) - 1} \end{aligned} \quad (3.4.3)$$

This recurrence generates the rational functions through  $m + 1$  points from the ones through  $m$  and (the term  $R_{(i+1)\dots(i+m-1)}$  in equation 3.4.3)  $m - 1$  points. It is started with

$$R_i = y_i \quad (3.4.4)$$

and with

$$R \equiv [R_{i(i+1)\dots(i+m)} \quad \text{with} \quad m = -1] = 0 \quad (3.4.5)$$

Now, exactly as in equations (3.2.4) and (3.2.5) above, we can convert the recurrence (3.4.3) to one involving only the small differences

$$\begin{aligned} C_{m,i} &\equiv R_{i\dots(i+m)} - R_{i\dots(i+m-1)} \\ D_{m,i} &\equiv R_{i\dots(i+m)} - R_{(i+1)\dots(i+m)} \end{aligned} \quad (3.4.6)$$

Note that these satisfy the relation

$$C_{m+1,i} - D_{m+1,i} = C_{m,i+1} - D_{m,i} \quad (3.4.7)$$

which is useful in proving the recurrences

$$\begin{aligned} D_{m+1,i} &= \frac{C_{m,i+1}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right)D_{m,i} - C_{m,i+1}} \\ C_{m+1,i} &= \frac{\left(\frac{x-x_i}{x-x_{i+m+1}}\right)D_{m,i}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right)D_{m,i} - C_{m,i+1}} \end{aligned} \quad (3.4.8)$$

The class for rational function interpolation is identical to that for polynomial interpolation in every way, except, of course, for the different method implemented in `rawinterp`. See the end of §3.2 for usage. Plausible values for  $M$  are in the range 4 to 7.

`interp_1d.h`

`struct Rat_interp : Base_interp`

Diagonal rational function interpolation object. Construct with `x` and `y` vectors, and the number `m` of points to be used locally, then call `interp` for interpolated values.

```
{
    Doub dy;
    Rat_interp(VecDoub_I &xv, VecDoub_I &yy, Int m)
        : Base_interp(xv,&yy[0],m), dy(0.) {}
    Doub rawinterp(Int jl, Doub x);
};
```

`Doub Rat_interp::rawinterp(Int jl, Doub x)`

Given a value `x`, and using pointers to data `xx` and `yy`, this routine returns an interpolated value `y`, and stores an error estimate `dy`. The returned value is obtained by `mm`-point diagonal rational function interpolation on the subrange `xx[jl..jl+mm-1]`.

```
{
    const Doub TINY=1.0e-99;           A small number.
    Int m,i,ns=0;
    Doub y,w,t,hh,h,dd;
    const Doub *xa = &xx[jl], *ya = &yy[jl];
    VecDoub c(mm),d(mm);
    hh=abs(x-xa[0]);
    for (i=0;i<mm;i++) {
        h=abs(x-xa[i]);
        if (h == 0.0) {
            dy=0.0;
            return ya[i];
        } else if (h < hh) {
            ns=i;
            hh=h;
        }
        c[i]=ya[i];
        d[i]=ya[i]+TINY;
    }                                     The TINY part is needed to prevent a rare zero-over-zero
    y=ya[ns--];                         condition.
    for (m=1;m<mm;m++) {
        for (i=0;i<mm-m;i++) {
            w=c[i+1]-d[i];
            h=xa[i+m]-x;          h will never be zero, since this was tested in the initial-
            t=(xa[i]-x)*d[i]/h;      izing loop.
            dd=t-c[i+1];
            if (dd == 0.0) throw("Error in routine ratint");
            This error condition indicates that the interpolating function has a pole at the
            requested value of x.
            dd=w/dd;
            d[i]=c[i+1]*dd;
            c[i]=t*dd;
```

```

    }
    y += (dy=(2*(ns+1) < (mm-m) ? c[ns+1] : d[ns--]));
}
return y;
}

```

### 3.4.1 Barycentric Rational Interpolation

Suppose one tries to use the above algorithm to construct a global approximation on the entire table of values using all the given nodes  $x_0, x_1, \dots, x_{N-1}$ . One potential drawback is that the approximation can have poles inside the interpolation interval where the denominator in (3.4.1) vanishes, even if the original function has no poles there. An even greater (related) hazard is that we have allowed the order of the approximation to grow to  $N - 1$ , probably much too large.

An alternative algorithm can be derived [2] that has no poles anywhere on the real axis, and that allows the actual order of the approximation to be specified to be any integer  $d < N$ . The trick is to make the degree of both the numerator and the denominator in equation (3.4.1) be  $N - 1$ . This requires that the  $p$ 's and the  $q$ 's not be independent, so that equation (3.4.2) no longer holds.

The algorithm utilizes the *barycentric form* of the rational interpolant

$$R(x) = \frac{\sum_{i=0}^{N-1} \frac{w_i}{x - x_i} y_i}{\sum_{i=0}^{N-1} \frac{w_i}{x - x_i}} \quad (3.4.9)$$

One can show that by a suitable choice of the weights  $w_i$ , *every* rational interpolant can be written in this form, and that, as a special case, so can polynomial interpolants [3]. It turns out that this form has many nice numerical properties. Barycentric rational interpolation competes very favorably with splines: its error is often smaller, and the resulting approximation is infinitely smooth (unlike splines).

Suppose we want our rational interpolant to have approximation order  $d$ , i.e., if the spacing of the points is  $O(h)$ , the error is  $O(h^{d+1})$  as  $h \rightarrow 0$ . Then the formula for the weights is

$$w_k = \sum_{\substack{i=k-d \\ 0 \leq i < N-d}}^k (-1)^k \prod_{\substack{j=i \\ j \neq k}}^{i+d} \frac{1}{x_k - x_j} \quad (3.4.10)$$

For example,

$$\begin{aligned} w_k &= (-1)^k, & d &= 0 \\ w_k &= (-1)^{k-1} \left[ \frac{1}{x_k - x_{k-1}} + \frac{1}{x_{k+1} - x_k} \right], & d &= 1 \end{aligned} \quad (3.4.11)$$

In the last equation, you omit the terms in  $w_0$  and  $w_{N-1}$  that refer to out-of-range values of  $x_k$ .

Here is a routine that implements barycentric rational interpolation. Given a set of  $N$  nodes and a desired order  $d$ , with  $d < N$ , it first computes the weights  $w_k$ . Then subsequent calls to `interp` evaluate the interpolant using equation (3.4.9). Note that the parameter `j1` of `rawinterp` is not used, since the algorithm is designed to construct an approximation on the entire interval at once.

The workload to construct the weights is of order  $O(Nd)$  operations. For small  $d$ , this is not too different from splines. Note, however, that the workload for each subsequent interpolated value is  $O(N)$ , not  $O(d)$  as for splines.

`interp_1d.h`

```
struct BaryRat_interp : Base_interp
```

Barycentric rational interpolation object. After constructing the object, call `interp` for interpolated values. Note that no error estimate `dy` is calculated.

```
{
```

```
    VecDoub w;
    Int d;
    BaryRat_interp(VecDoub_I &xv, VecDoub_I &yv, Int dd);
    Doub rawinterp(Int jl, Doub x);
    Doub interp(Doub x);
}
```

```
BaryRat_interp::BaryRat_interp(VecDoub_I &xv, VecDoub_I &yv, Int dd)
    : Base_interp(xv,&yv[0],xv.size()), w(n), d(dd)
```

Constructor arguments are `x` and `y` vectors of length `n`, and order `d` of desired approximation.

```
{
```

```
    if (n<=d) throw("d too large for number of points in BaryRat_interp");
    for (Int k=0;k<n;k++) {
```

Compute weights from equation (3.4.10).

```
        Int imin=MAX(k-d,0);
        Int imax = k >= n-d ? n-d-1 : k;
        Doub temp = imin & 1 ? -1.0 : 1.0;
        Doub sum=0.0;
        for (Int i=imin;i<=imax;i++) {
            Int jmax=MIN(i+d,n-1);
            Doub term=1.0;
            for (Int j=i;j<=jmax;j++) {
                if (j==k) continue;
                term *= (xx[k]-xx[j]);
            }
            term=temp/term;
            temp=-temp;
            sum += term;
        }
        w[k]=sum;
    }
```

```
}
```

```
Doub BaryRat_interp::rawinterp(Int jl, Doub x)
```

Use equation (3.4.9) to compute the barycentric rational interpolant. Note that `jl` is not used since the approximation is global; it is included only for compatibility with `Base_interp`.

```
{
```

```
    Doub num=0,den=0;
    for (Int i=0;i<n;i++) {
        Doub h=x-xx[i];
        if (h == 0.0) {
            return yy[i];
        } else {
            Doub temp=w[i]/h;
            num += temp*yy[i];
            den += temp;
        }
    }
    return num/den;
}
```

```
Doub BaryRat_interp::interp(Doub x) {
```

No need to invoke `hunt` or `locate` since the interpolation is global, so override `interp` to simply call `rawinterp` directly with a dummy value of `jl`.

```
    return rawinterp(1,x);
}
```

It is wise to start with small values of  $d$  before trying larger values.

**CITED REFERENCES AND FURTHER READING:**

- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §2.2.[1]
- Floater, M.S., and Hormann, K. 2006+, "Barycentric Rational Interpolation with No Poles and High Rates of Approximation," at <http://www.in.tu-clausthal.de/fileadmin/homes/techreports/ifi0606hormann.pdf>.[2]
- Berrut, J.-P., and Trefethen, L.N. 2004, "Barycentric Lagrange Interpolation," *SIAM Review*, vol. 46, pp. 501–517.[3]
- Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.2.
- Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 3.

## 3.5 Coefficients of the Interpolating Polynomial

Occasionally you may wish to know not the value of the interpolating polynomial that passes through a (small!) number of points, but the coefficients of that polynomial. A valid use of the coefficients might be, for example, to compute simultaneous interpolated values of the function and of several of its derivatives (see §5.1), or to convolve a segment of the tabulated function with some other function, where the moments of that other function (i.e., its convolution with powers of  $x$ ) are known analytically.

Please be certain, however, that the coefficients are what you need. Generally the coefficients of the interpolating polynomial can be determined much less accurately than its value at a desired abscissa. Therefore, it is not a good idea to determine the coefficients only for use in calculating interpolating values. Values thus calculated will not pass exactly through the tabulated points, for example, while values computed by the routines in §3.1 – §3.3 will pass exactly through such points.

Also, you should not mistake the interpolating polynomial (and its coefficients) for its cousin, the *best-fit* polynomial through a data set. Fitting is a *smoothing* process, since the number of fitted coefficients is typically much less than the number of data points. Therefore, fitted coefficients can be accurately and stably determined even in the presence of statistical errors in the tabulated values. (See §14.9.) Interpolation, where the number of coefficients and number of tabulated points are equal, takes the tabulated values as perfect. If they in fact contain statistical errors, these can be magnified into oscillations of the interpolating polynomial in between the tabulated points.

As before, we take the tabulated points to be  $y_i \equiv y(x_i)$ . If the interpolating polynomial is written as

$$y = c_0 + c_1x + c_2x^2 + \cdots + c_{N-1}x^{N-1} \quad (3.5.1)$$

then the  $c_i$ 's are required to satisfy the linear equation

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \cdots & x_{N-1}^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} \quad (3.5.2)$$

This is a *Vandermonde matrix*, as described in §2.8. One could in principle solve equation (3.5.2) by standard techniques for linear equations generally (§2.3); however, the special method that was derived in §2.8 is more efficient by a large factor, of order  $N$ , so it is much better.

Remember that Vandermonde systems can be quite ill-conditioned. In such a case, *no* numerical method is going to give a very accurate answer. Such cases do not, please note, imply any difficulty in finding interpolated *values* by the methods of §3.2, but only difficulty in finding *coefficients*.

Like the routine in §2.8, the following is due to G.B. Rybicki.

```
polcoef.h void polcoe(VecDoub_I &x, VecDoub_I &y, VecDoub_0 &cof)
Given arrays x[0..n-1] and y[0..n-1] containing a tabulated function  $y_i = f(x_i)$ , this routine
returns an array of coefficients cof[0..n-1], such that  $y_i = \sum_{j=0}^{n-1} \text{cof}_j x_i^j$ .
{
    Int k,j,i,n=x.size();
    Doub phi,ff,b;
    VecDoub s(n);
    for (i=0;i<n;i++) s[i]=cof[i]=0.0;
    s[n-1]=-x[0];
    for (i=1;i<n;i++) {
        for (j=n-1-i;j<n-1;j++)
            s[j] -= x[i]*s[j+1];
        s[n-1] -= x[i];
    }
    for (j=0;j<n;j++) {
        phi=n;
        for (k=n-1;k>0;k--)
            phi=k*s[k]+x[j]*phi;
        ff=y[j]/phi;
        b=1.0;
        for (k=n-1;k>=0;k--) {
            cof[k] += b*ff;
            b=s[k]+x[j]*b;
        }
    }
}
```

Coefficients  $s_i$  of the master polynomial  $P(x)$  are found by recurrence.

The quantity  $\phi = \prod_{j \neq k} (x_j - x_k)$  is found as a derivative of  $P(x_j)$ .

Coefficients of polynomials in each term of the Lagrange formula are found by synthetic division of  $P(x)$  by  $(x - x_j)$ . The solution  $c_k$  is accumulated.

### 3.5.1 Another Method

Another technique is to make use of the function value interpolation routine already given (`polint`; §3.2). If we interpolate (or extrapolate) to find the value of the interpolating polynomial at  $x = 0$ , then this value will evidently be  $c_0$ . Now we can subtract  $c_0$  from the  $y_i$ 's and divide each by its corresponding  $x_i$ . Throwing out one point (the one with smallest  $x_i$  is a good candidate), we can repeat the procedure to find  $c_1$ , and so on.

It is not instantly obvious that this procedure is stable, but we have generally found it to be somewhat *more* stable than the routine immediately preceding. This method is of order  $N^3$ , while the preceding one was of order  $N^2$ . You will find, however, that neither works very well for large  $N$ , because of the intrinsic ill-condition of the Vandermonde problem. In single precision,  $N$  up to 8 or 10 is satisfactory; about double this in double precision.

```

void polcof(VecDoub_I &xa, VecDoub_I &ya, VecDoub_0 &cof)
Given arrays xa[0..n-1] and ya[0..n-1] containing a tabulated function yai = f(xai), this
routine returns an array of coefficients cof [0..n-1], such that yai =  $\sum_{j=0}^{n-1} \text{cof}_j \text{xa}_i^j$ .
{
    Int k,j,i,n=xa.size();
    Doub xmin;
    VecDoub x(n),y(n);
    for (j=0;j<n;j++) {
        x[j]=xa[j];
        y[j]=ya[j];
    }
    for (j=0;j<n;j++) {
        VecDoub x_t(n-j),y_t(n-j);
        for (k=0;k<n-j;k++) {
            x_t[k]=x[k];
            y_t[k]=y[k];
        }
        Poly_interp interp(x,y,n-j);
        cof[j] = interp.rawinterp(0,0.);           Fill a temporary vector whose size
        xmin=1.0e99;                                decreases as each coefficient is
        k = -1;                                     found.
        for (i=0;i<n-j;i++) {
            if (abs(x[i]) < xmin) {                Extrapolate to x = 0.
                xmin=abs(x[i]);
                k=i;
            }
            if (x[i] != 0.0)                      Find the remaining xi of smallest
                y[i]=(y[i]-cof[j])/x[i];          absolute value
        }
        for (i=k+1;i<n-j;i++) {                  (meanwhile reducing all the terms)
            y[i-1]=y[i];
            x[i-1]=x[i];
        }
    }
}

```

polcof.h

If the point  $x = 0$  is not in (or at least close to) the range of the tabulated  $x_i$ 's, then the coefficients of the interpolating polynomial will in general become very large. However, the real “information content” of the coefficients is in small differences from the “translation-induced” large values. This is one cause of ill-conditioning, resulting in loss of significance and poorly determined coefficients. In this case, you should consider redefining the origin of the problem, to put  $x = 0$  in a sensible place.

Another pathology is that, if too high a degree of interpolation is attempted on a smooth function, the interpolating polynomial will attempt to use its high-degree coefficients, in combinations with large and almost precisely canceling combinations, to match the tabulated values down to the last possible epsilon of accuracy. This effect is the same as the intrinsic tendency of the interpolating polynomial values to oscillate (wildly) between its constrained points and would be present even if the machine's floating precision were infinitely good. The above routines polcof and polcof have slightly different sensitivities to the pathologies that can occur.

Are you still quite certain that using the *coefficients* is a good idea?

#### CITED REFERENCES AND FURTHER READING:

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), §5.2.

## 3.6 Interpolation on a Grid in Multidimensions

In multidimensional interpolation, we seek an estimate of a function of more than one independent variable,  $y(x_1, x_2, \dots, x_n)$ . The Great Divide is, Are we given a complete set of tabulated values on an  $n$ -dimensional grid? Or, do we know function values only on some scattered set of points in the  $n$ -dimensional space? In one dimension, the question never arose, because any set of  $x_i$ 's, once sorted into ascending order, could be viewed as a valid one-dimensional grid (regular spacing not being a requirement).

As the number of dimensions  $n$  gets large, maintaining a full grid becomes rapidly impractical, because of the explosion in the number of gridpoints. Methods that work with scattered data, to be considered in §3.7, then become the methods of choice. Don't, however, make the mistake of thinking that such methods are intrinsically more accurate than grid methods. In general they are less accurate. Like the proverbial three-legged dog, they are remarkable because they work at all, not because they work, necessarily, well!

Both kinds of methods are practical in two dimensions, and some other kinds as well. For example, *finite element methods*, of which *triangulation* is the most common, find ways to impose some kind of geometrically regular structure on scattered points, and then use that structure for interpolation. We will treat two-dimensional interpolation by triangulation in detail in §21.6; that section should be considered as a continuation of the discussion here.

In the remainder of this section, we consider only the case of interpolating on a grid, and we implement in code only the (most common) case of two dimensions. All of the methods given generalize to three dimensions in an obvious way. When we implement methods for scattered data, in §3.7, the treatment will be for general  $n$ .

In two dimensions, we imagine that we are given a matrix of functional values  $y_{ij}$ , with  $i = 0, \dots, M - 1$  and  $j = 0, \dots, N - 1$ . We are also given an array of  $x_1$  values  $x_{1i}$ , and an array of  $x_2$  values  $x_{2j}$ , with  $i$  and  $j$  as just stated. The relation of these input quantities to an underlying function  $y(x_1, x_2)$  is just

$$y_{ij} = y(x_{1i}, x_{2j}) \quad (3.6.1)$$

We want to estimate, by interpolation, the function  $y$  at some untabulated point  $(x_1, x_2)$ .

An important concept is that of the *grid square* in which the point  $(x_1, x_2)$  falls, that is, the four tabulated points that surround the desired interior point. For convenience, we will number these points from 0 to 3, counterclockwise starting from the lower left. More precisely, if

$$\begin{aligned} x_{1i} &\leq x_1 \leq x_{1(i+1)} \\ x_{2j} &\leq x_2 \leq x_{2(j+1)} \end{aligned} \quad (3.6.2)$$

defines values of  $i$  and  $j$ , then

$$\begin{aligned} y_0 &\equiv y_{ij} \\ y_1 &\equiv y_{(i+1)j} \\ y_2 &\equiv y_{(i+1)(j+1)} \\ y_3 &\equiv y_{i(j+1)} \end{aligned} \quad (3.6.3)$$

The simplest interpolation in two dimensions is *bilinear interpolation* on the grid square. Its formulas are

$$\begin{aligned} t &\equiv (x_1 - x_{1i}) / (x_{1(i+1)} - x_{1i}) \\ u &\equiv (x_2 - x_{2j}) / (x_{2(j+1)} - x_{2j}) \end{aligned} \quad (3.6.4)$$

(so that  $t$  and  $u$  each lie between 0 and 1) and

$$y(x_1, x_2) = (1-t)(1-u)y_0 + t(1-u)y_1 + tuy_2 + (1-t)uy_3 \quad (3.6.5)$$

Bilinear interpolation is frequently “close enough for government work.” As the interpolating point wanders from grid square to grid square, the interpolated function value changes continuously. However, the gradient of the interpolated function changes discontinuously at the boundaries of each grid square.

We can easily implement an object for bilinear interpolation by grabbing pieces of “machinery” from our one-dimensional interpolation classes:

```
struct Bilin_interp {
```

Object for bilinear interpolation on a matrix. Construct with a vector of  $x_1$  values, a vector of  $x_2$  values, and a matrix of tabulated function values  $y_{ij}$ . Then call `interp` for interpolated values.

```
    Int m,n;
    const MatDoub &y;
    Linear_interp x1terp, x2terp;

    Bilin_interp(VecDoub_I &x1v, VecDoub_I &x2v, MatDoub_I &ym)
        : m(x1v.size()), n(x2v.size()), y(ym),
        x1terp(x1v, x1v), x2terp(x2v, x2v) {}           Construct dummy 1-dim interpola-
                                                       tions for their locate and hunt
    Doub interp(Doub x1p, Doub x2p) {                  functions.
        Int i,j;
        Doub yy, t, u;
        i = x1terp.cor ? x1terp.hunt(x1p) : x1terp.locate(x1p);
        j = x2terp.cor ? x2terp.hunt(x2p) : x2terp.locate(x2p);
        Find the grid square.
        t = (x1p-x1terp.xx[i])/(x1terp.xx[i+1]-x1terp.xx[i]);  Interpolate.
        u = (x2p-x2terp.xx[j])/(x2terp.xx[j+1]-x2terp.xx[j]);
        yy = (1.-t)*(1.-u)*y[i][j] + t*(1.-u)*y[i+1][j]
            + (1.-t)*u*y[i][j+1] + t*u*y[i+1][j+1];
        return yy;
    }
};
```

`interp_2d.h`

Here we declare two instances of `Linear_interp`, one for each direction, and use them merely to do the bookkeeping on the arrays  $x_{1i}$  and  $x_{2j}$  — in particular, to provide the “intelligent” table-searching mechanisms that we have come to rely on. (The second occurrence of  $x1v$  and  $x2v$  in the constructors is just a placeholder; there are not really any one-dimensional “ $y$ ” arrays.)

Usage of `Bilin_interp` is just what you’d expect:

```
Int m=..., n=...
MatDoub yy(m,n);
VecDoub x1(m), x2(n);
...
Bilin_interp myfunc(x1,x2,yy);
```

followed (any number of times) by

```
Doub x1,x2,y;
...
y = myfunc.interp(x1,x2);
```

Bilinear interpolation is a good place to start, in two dimensions, unless you positively know that you need something fancier.

There are two distinctly different directions that one can take in going beyond bilinear interpolation to higher-order methods: One can use higher order to obtain increased accuracy for the interpolated function (for sufficiently smooth functions!), without necessarily trying to fix up the continuity of the gradient and higher derivatives. Or, one can make use of higher order to enforce smoothness of some of these derivatives as the interpolating point crosses grid-square boundaries. We will now consider each of these two directions in turn.

### 3.6.1 Higher Order for Accuracy

The basic idea is to break up the problem into a succession of one-dimensional interpolations. If we want to do  $m-1$  order interpolation in the  $x_1$  direction, and  $n-1$  order in the  $x_2$  direction, we first locate an  $m \times n$  sub-block of the tabulated function matrix that contains our desired point  $(x_1, x_2)$ . We then do  $m$  one-dimensional interpolations in the  $x_2$  direction, i.e., on the rows of the sub-block, to get function values at the points  $(x_{1i}, x_2)$ , with  $m$  values of  $i$ . Finally, we do a last interpolation in the  $x_1$  direction to get the answer.

Again using the previous one-dimensional machinery, this can all be coded very concisely as

`interp_2d.h`

```
struct Poly2D_interp {
```

Object for two-dimensional polynomial interpolation on a matrix. Construct with a vector of  $x_1$  values, a vector of  $x_2$  values, a matrix of tabulated function values  $y_{ij}$ , and integers to specify the number of points to use locally in each direction. Then call `interp` for interpolated values.

```
    Int m,n,mm,nn;
    const MatDoub &y;
    VecDoub yv;
    Poly_interp x1terp, x2terp;

    Poly2D_interp(VecDoub_I &x1v, VecDoub_I &x2v, MatDoub_I &ym,
        Int mp, Int np) : m(x1v.size()), n(x2v.size()),
        mm(mp), nn(np), y(ym), yv(m),
        x1terp(x1v,yv,mm), x2terp(x2v,x2v,nn) {} Dummy 1-dim interpolations for their
        locate and hunt functions.

    Doub interp(Doub x1p, Doub x2p) {
        Int i,j,k;
        i = x1terp.cor ? x1terp.hunt(x1p) : x1terp.locate(x1p);
        j = x2terp.cor ? x2terp.hunt(x2p) : x2terp.locate(x2p);
        Find grid block.
        for (k=i;k<i+mm;k++) {                                mm interpolations in the x2 direction.
            x2terp.yy = &y[k][0];
            yv[k] = x2terp.rawinterp(j,x2p);
        }
        return x1terp.rawinterp(i,x1p);                         A final interpolation in the x1 direc-
    };
```

The user interface is the same as for `Bilin_interp`, except that the constructor has two additional arguments that specify the number of points (order plus one) to be used locally in, respectively, the  $x_1$  and  $x_2$  interpolations. Typical values will be in the range 3 to 7.

Code stylists won't like some of the details in Poly2D\_interp (see discussion in §3.1 immediately following Base\_interp). As we loop over the rows of the sub-block, we reach into the guts of x2terp and repoint its yy array to a row of our  $y$  matrix. Further, we alter the contents of the array yy, for which x1terp has stored a pointer, on the fly. None of this is particularly dangerous as long as we control the implementations in both Base\_interp and Poly2D\_interp; and it makes for a very efficient implementation. You should view these two classes as not just (implicitly) `friend` classes, but as *really intimate* friends.

### 3.6.2 Higher Order for Smoothness: Bicubic Spline

A favorite technique for obtaining smoothness in two-dimensional interpolation is the *bicubic spline*. To set up a bicubic spline, you (one time) construct  $M$  one-dimensional splines across the rows of the two-dimensional matrix of function values. Then, for each desired interpolated value you proceed as follows: (1) Perform  $M$  spline interpolations to get a vector of values  $y(x_{1i}, x_2)$ ,  $i = 0, \dots, M - 1$ . (2) Construct a one-dimensional spline through those values. (3) Finally, spline-interpolate to the desired value  $y(x_1, x_2)$ .

If this sounds like a lot of work, well, yes, it is. The one-time setup work scales as the table size  $M \times N$ , while the work per interpolated value scales roughly as  $M \log M + N$ , both with pretty hefty constants in front. This is the price that you pay for the desirable characteristics of splines that derive from their nonlocality. For tables with modest  $M$  and  $N$ , less than a few hundred, say, the cost is usually tolerable. If it's not, then fall back to the previous local methods.

Again a very concise implementation is possible:

```
struct Spline2D_interp {
    Object for two-dimensional cubic spline interpolation on a matrix. Construct with a vector of  $x_1$  values, a vector of  $x_2$  values, and a matrix of tabulated function values  $y_{ij}$ . Then call interp for interpolated values.
    Int m,n;
    const MatDoub &y;
    const VecDoub &x1;
    VecDoub yv;
    NRvector<Spline_interp*> srp;

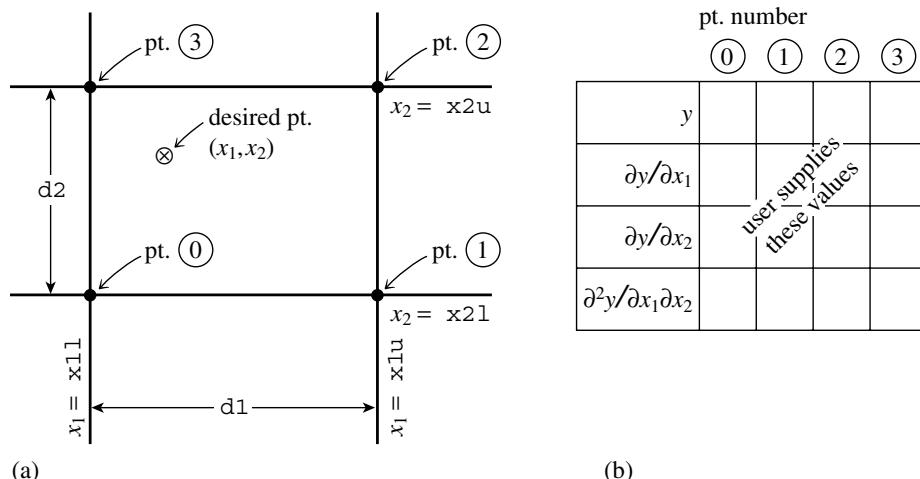
    Spline2D_interp(VecDoub_I &x1v, VecDoub_I &x2v, MatDoub_I &ym)
        : m(x1v.size()), n(x2v.size()), y(ym), yv(m), x1(x1v), srp(m) {
        for (Int i=0;i<m;i++) srp[i] = new Spline_interp(x2v,&y[i][0]);
        Save an array of pointers to 1-dim row splines.
    }

    ~Spline2D_interp(){
        for (Int i=0;i<m;i++) delete srp[i];      We need a destructor to clean up.
    }

    Doub interp(Doub x1p, Doub x2p) {
        for (Int i=0;i<m;i++) yv[i] = (*srp[i]).interp(x2p);
        Interpolate on each row.
        Spline_interp scol(x1,yv);                  Construct the column spline,
        return scol.interp(x1p);                    and evaluate it.
    }
};
```

interp\_2d.h

The reason for that ugly vector of pointers to `Spline_interp` objects is that we need to initialize each row spline separately, with data from the appropriate row. The user interface is the same as `Bilin_interp`, above.



**Figure 3.6.1.** (a) Labeling of points used in the two-dimensional interpolation routines `bcout` and `bcucof`. (b) For each of the four points in (a), the user supplies one function value, two first derivatives, and one cross-derivative, a total of 16 numbers.

### 3.6.3 Higher Order for Smoothness: Bicubic Interpolation

Bicubic interpolation gives the same degree of smoothness as bicubic spline interpolation, but it has the advantage of being a local method. Thus, after you set it up, a function interpolation costs only a constant, plus  $\log M + \log N$ , to find your place in the table. Unfortunately, this advantage comes with a lot of complexity in coding. Here, we will give only some building blocks for the method, not a complete user interface.

Bicubic splines are in fact a special case of bicubic interpolation. In the general case, however, we leave the values of all derivatives at the grid points as freely specifiable. You, the user, can specify them *any way you want*. In other words, you specify at each grid point not just the function  $y(x_1, x_2)$ , but also the gradients  $\partial y / \partial x_1 \equiv y_{,1}$ ,  $\partial y / \partial x_2 \equiv y_{,2}$  and the cross derivative  $\partial^2 y / \partial x_1 \partial x_2 \equiv y_{,12}$  (see Figure 3.6.1). Then an interpolating function that is *cubic* in the scaled coordinates  $t$  and  $u$  (equation 3.6.4) can be found, with the following properties: (i) The values of the function and the specified derivatives are reproduced exactly on the grid points, and (ii) the values of the function and the specified derivatives change continuously as the interpolating point crosses from one grid square to another.

It is important to understand that nothing in the equations of bicubic interpolation requires you to specify the extra derivatives *correctly*! The smoothness properties are tautologically “forced,” and have nothing to do with the “accuracy” of the specified derivatives. It is a separate problem for you to decide how to obtain the values that are specified. The better you do, the more *accurate* the interpolation will be. But it will be *smooth* no matter what you do.

Best of all is to know the derivatives analytically, or to be able to compute them accurately by numerical means, at the grid points. Next best is to determine them by numerical differencing from the functional values already tabulated on the grid. The relevant code would be something like this (using centered differencing):

```

y1a[j][k]=(ya[j+1][k]-ya[j-1][k])/(x1a[j+1]-x1a[j-1]);
y2a[j][k]=(ya[j][k+1]-ya[j][k-1])/(x2a[k+1]-x2a[k-1]);
y12a[j][k]=(ya[j+1][k+1]-ya[j+1][k-1]-ya[j-1][k+1]+ya[j-1][k-1])
/((x1a[j+1]-x1a[j-1])*(x2a[k+1]-x2a[k-1]));

```

To do a bicubic interpolation within a grid square, given the function  $y$  and the derivatives  $y_1, y_2, y_{12}$  at each of the four corners of the square, there are two steps: First obtain the 16 quantities  $c_{ij}, i, j = 0, \dots, 3$  using the routine `bccuof` below. (The formulas that obtain the  $c$ 's from the function and derivative values are just a complicated linear transformation, with coefficients that, having been determined once in the mists of numerical history, can be tabulated and forgotten.) Next, substitute the  $c$ 's into any or all of the following bicubic formulas for function and derivatives, as desired:

$$\begin{aligned}
y(x_1, x_2) &= \sum_{i=0}^3 \sum_{j=0}^3 c_{ij} t^i u^j \\
y_{,1}(x_1, x_2) &= \sum_{i=0}^3 \sum_{j=0}^3 i c_{ij} t^{i-1} u^j (dt/dx_1) \\
y_{,2}(x_1, x_2) &= \sum_{i=0}^3 \sum_{j=0}^3 j c_{ij} t^i u^{j-1} (du/dx_2) \\
y_{,12}(x_1, x_2) &= \sum_{i=0}^3 \sum_{j=0}^3 i j c_{ij} t^{i-1} u^{j-1} (dt/dx_1)(du/dx_2)
\end{aligned} \tag{3.6.6}$$

where  $t$  and  $u$  are again given by equation (3.6.4).

```

void bccuof(VecDoub_I &y, VecDoub_I &y1, VecDoub_I &y2, VecDoub_I &y12,           interp_2d.h
            const Doub d1, const Doub d2, MatDoub_0 &c) {
Given arrays y[0..3], y1[0..3], y2[0..3], and y12[0..3], containing the function, gradients,
and cross-derivative at the four grid points of a rectangular grid cell (numbered counterclockwise
from the lower left), and given d1 and d2, the length of the grid cell in the 1 and 2 directions, this
routine returns the table c[0..3][0..3] that is used by routine bcuint for bicubic interpolation.

static Int wt_d[16*16]=
{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
 -3, 0, 0, 3, 0, 0, 0, 0, -2, 0, 0, -1, 0, 0, 0, 0,
 2, 0, 0, -2, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0,
 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
 0, 0, 0, 0, -3, 0, 0, 3, 0, 0, 0, 0, -2, 0, 0, -1,
 0, 0, 0, 0, 2, 0, 0, -2, 0, 0, 0, 0, 1, 0, 0, 1,
 -3, 3, 0, 0, -2, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, -3, 3, 0, 0, -2, -1, 0, 0,
 9,-9, 9,-9, 6, 3,-3,-6, 6,-6,-3, 3, 4, 2, 1, 2,
 -6, 6,-6, 6,-4,-2, 2, 4,-3, 3, 3,-3,-2,-1,-1,-2,
 2,-2, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 2,-2, 0, 0, 1, 1, 0, 0,
 -6, 6,-6, 6,-3,-3, 3, 3,-4, 4, 2,-2,-2,-2,-1,-1,
 4,-4, 4,-4, 2, 2,-2,-2, 2,-2,-2, 2, 1, 1, 1, 1};

Int l,k,j,i;
Doub xx,d1d2=d1*d2;
VecDoub c1(16),x(16);
static MatInt wt(16,16,wt_d);
for (i=0;i<4;i++) {           Pack a temporary vector x.

```

```

x[i]=y[i];
x[i+4]=y1[i]*d1;
x[i+8]=y2[i]*d2;
x[i+12]=y12[i]*d1d2;
}
for (i=0;i<16;i++) {           Matrix-multiply by the stored table.
    xx=0.0;
    for (k=0;k<16;k++) xx += wt[i][k]*x[k];
    cl[i]=xx;
}
l=0;
for (i=0;i<4;i++)           Unpack the result into the output table.
    for (j=0;j<4;j++) c[i][j]=cl[l++];
}

```

The implementation of equation (3.6.6), which performs a bicubic interpolation, gives back the interpolated function value and the two gradient values, and uses the above routine `bcucof`, is simply:

```

interp_2d.h void bcuint(VecDoub_I &y, VecDoub_I &y1, VecDoub_I &y2, VecDoub_I &y12,
                        const Doub x1l, const Doub x1u, const Doub x2l, const Doub x2u,
                        const Doub x1, const Doub x2, Doub &ansy, Doub &ansy1, Doub &ansy2) {
Bicubic interpolation within a grid square. Input quantities are y,y1,y2,y12 (as described in
bcucof); x1l and x1u, the lower and upper coordinates of the grid square in the 1 direction;
x2l and x2u likewise for the 2 direction; and x1,x2, the coordinates of the desired point for
the interpolation. The interpolated function value is returned as ansy, and the interpolated
gradient values as ansy1 and ansy2. This routine calls bcucof.
    Int i;
    Doub t,u,d1=x1u-x1l,d2=x2u-x2l;
    MatDoub c(4,4);
    bcucof(y,y1,y2,y12,d1,d2,c);           Get the c's.
    if (x1u == x1l || x2u == x2l)
        throw("Bad input in routine bcuint");
    t=(x1-x1l)/d1;                         Equation (3.6.4).
    u=(x2-x2l)/d2;
    ansy=ansy2=ansy1=0.0;
    for (i=3;i>0;i--) {                   Equation (3.6.6).
        ansy=t*ansy+((c[i][3]*u+c[i][2])*u+c[i][1])*u+c[i][0];
        ansy2=t*ansy2+(3.0*c[i][3]*u+2.0*c[i][2])*u+c[i][1];
        ansy1=u*ansy1+(3.0*c[3][i]*t+2.0*c[2][i])*t+c[1][i];
    }
    ansy1 /= d1;
    ansy2 /= d2;
}

```

You can combine the best features of bicubic interpolation and bicubic splines by using splines to compute values for the necessary derivatives at the grid points, storing these values, and then using bicubic interpolation, with an efficient table-searching method, for the actual function interpolations. Unfortunately this is beyond our scope here.

#### CITED REFERENCES AND FURTHER READING:

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, §25.2.
- Kinahan, B.F., and Harm, R. 1975, "Chemical Composition and the Hertzsprung Gap," *Astrophysical Journal*, vol. 200, pp. 330–335.

- 
- Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §5.2.7.
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §7.7.

## 3.7 Interpolation on Scattered Data in Multidimensions

We now leave behind, if with some trepidation, the orderly world of regular grids. Courage is required. We are given an arbitrarily scattered set of  $N$  data points  $(\mathbf{x}_i, y_i)$ ,  $i = 0, \dots, N-1$  in  $n$ -dimensional space. Here  $\mathbf{x}_i$  denotes an  $n$ -dimensional vector of independent variables,  $(x_{1i}, x_{2i}, \dots, x_{ni})$ , and  $y_i$  is the value of the function at that point.

In this section we discuss two of the most widely used *general* methods for this problem, *radial basis function (RBF)* interpolation, and *kriging*. Both of these methods are expensive. By that we mean that they require  $O(N^3)$  operations to initially digest a set of data points, followed by  $O(N)$  operations for each interpolated value. Kriging is also able to supply an error estimate — but at the rather high cost of  $O(N^2)$  per value. Shepard interpolation, discussed below, is a variant of RBF that at least avoids the  $O(N^3)$  initial work; otherwise these workloads effectively limit the usefulness of these general methods to values of  $N \lesssim 10^4$ . It is therefore worthwhile for you to consider whether you have any other options. Two of these are

- If  $n$  is not too large (meaning, usually,  $n = 2$ ), and if the data points are fairly dense, then consider triangulation, discussed in §21.6. Triangulation is an example of a finite element method. Such methods construct some semblance of geometric regularity and then exploit that construction to advantage. Mesh generation is a closely related subject.
- If your accuracy goals will tolerate it, consider moving each data point to the nearest point on a regular Cartesian grid and then using Laplace interpolation (§3.8) to fill in the rest of the grid points. After that, you can interpolate on the grid by the methods of §3.6. You will need to compromise between making the grid very fine (to minimize the error introduced when you move the points) and the compute time workload of the Laplace method.

If neither of these options seem attractive, and you can't think of another one that is, then try one or both of the two methods that we now discuss. RBF interpolation is probably the more widely used of the two, but kriging is our personal favorite. Which works better will depend on the details of your problem.

The related, but easier, problem of *curve interpolation* in multidimensions is discussed at the end of this section.

### 3.7.1 Radial Basis Function Interpolation

The idea behind RBF interpolation is very simple: Imagine that every known point  $j$  “influences” its surroundings the same way in all directions, according to

some assumed functional form  $\phi(r)$  — the radial basis function — that is a function only of radial distance  $r = |\mathbf{x} - \mathbf{x}_j|$  from the point. Let us try to approximate the interpolating function everywhere by a linear combination of the  $\phi$ 's, centered on all the known points,

$$y(\mathbf{x}) = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x} - \mathbf{x}_i|) \quad (3.7.1)$$

where the  $w_i$ 's are some unknown set of weights. How do we find these weights? Well, we haven't used the function values  $y_i$  yet. The weights are determined by requiring that the interpolation be exact at all the known data points. That is equivalent to solving a set of  $N$  linear equations in  $N$  unknowns for the  $w_i$ 's:

$$y_j = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x}_j - \mathbf{x}_i|) \quad (3.7.2)$$

For many functional forms  $\phi$ , it can be proved, under various general assumptions, that this set of equations is nondegenerate and can be readily solved by, e.g., *LU* decomposition (§2.3). References [1,2] provide entry to the literature.

A variant on RBF interpolation is *normalized radial basis function (NRBF)* interpolation, in which we require the sum of the basis functions to be unity or, equivalently, replace equations (3.7.1) and (3.7.2) by

$$y(\mathbf{x}) = \frac{\sum_{i=0}^{N-1} w_i \phi(|\mathbf{x} - \mathbf{x}_i|)}{\sum_{i=0}^{N-1} \phi(|\mathbf{x} - \mathbf{x}_i|)} \quad (3.7.3)$$

and

$$y_j \sum_{i=0}^{N-1} \phi(|\mathbf{x}_j - \mathbf{x}_i|) = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x}_j - \mathbf{x}_i|) \quad (3.7.4)$$

Equations (3.7.3) and (3.7.4) arise more naturally from a Bayesian statistical perspective [3]. However, there is no evidence that either the NRBF method is consistently superior to the RBF method, or vice versa. It is easy to implement both methods in the same code, leaving the choice to the user.

As we already mentioned, for  $N$  data points the one-time work to solve for the weights by *LU* decomposition is  $O(N^3)$ . After that, the cost is  $O(N)$  for each interpolation. Thus  $N \sim 10^3$  is a rough dividing line (at 2007 desktop speeds) between “easy” and “difficult.” If your  $N$  is larger, however, don’t despair: There are *fast multipole methods*, beyond our scope here, with much more favorable scaling [1,4,5]. Another, much lower-tech, option is to use *Shepard interpolation* discussed later in this section.

Here are a couple of objects that implement everything discussed thus far. `RBF_fn` is a virtual base class whose derived classes will embody different functional forms for  $\text{rbf}(r) \equiv \phi(r)$ . `RBF_interp`, via its constructor, digests your data and solves the equations for the weights. The data points  $\mathbf{x}_i$  are input as an  $N \times n$  matrix, and the code works for any dimension  $n$ . A boolean argument `nrbf` inputs whether NRBF is to be used instead of RBF. You call `interp` to get an interpolated function value at a new point  $\mathbf{x}$ .

```

struct RBF_fn {
Abstract base class template for any particular radial basis function. See specific examples
below.
    virtual Doub rbf(Doub r) = 0;
};

struct RBF_interp {
Object for radial basis function interpolation using n points in dim dimensions. Call constructor
once, then interp as many times as desired.
    Int dim, n;
    const MatDoub &pts;
    const VecDoub &vals;
    VecDoub w;
    RBF_fn &fn;
    Bool norm;

    RBF_interp(MatDoub_I &ptss, VecDoub_I &valss, RBF_fn &func, Bool nrbf=false)
    : dim(ptss.ncols()), n(ptss.nrows()) , pts(ptss), vals(valss),
    w(n), fn(func), norm(nrbf) {
Constructor. The  $n \times dim$  matrix ptss inputs the data points, the vector valss the function
values. func contains the chosen radial basis function, derived from the class RBF_fn. The
default value of nrbf gives RBF interpolation; set it to 1 for NRBF.
    Int i,j;
    Doub sum;
    MatDoub rbf(n,n);
    VecDoub rhs(n);
    for (i=0;i<n;i++) { Fill the matrix  $\phi(|\mathbf{r}_i - \mathbf{r}_j|)$  and the r.h.s. vector.
        sum = 0.;
        for (j=0;j<n;j++) {
            sum += (rbf[i][j] = fn.rbf(rad(&pts[i][0],&pts[j][0])));
        }
        if (norm) rhs[i] = sum*vals[i];
        else rhs[i] = vals[i];
    }
    LUDcmp lu(rbf); Solve the set of linear equations.
    lu.solve(rhs,w);
}

Doub interp(VecDoub_I &pt) {
Return the interpolated function value at a dim-dimensional point pt.
    Doub fval, sum=0., sumw=0.;
    if (pt.size() != dim) throw("RBF_interp bad pt size");
    for (Int i=0;i<n;i++) { Sum over all tabulated points.
        fval = fn.rbf(rad(&pt[0],&pts[i][0]));
        sumw += w[i]*fval;
        sum += fval;
    }
    return norm ? sumw/sum : sumw;
}

Doub rad(const Doub *p1, const Doub *p2) {
Euclidean distance.
    Doub sum = 0.;
    for (Int i=0;i<dim;i++) sum += SQR(p1[i]-p2[i]);
    return sqrt(sum);
}

```

interp\_rbf.h

### 3.7.2 Radial Basis Functions in General Use

The most often used radial basis function is the *multiquadric* first used by Hardy, circa 1970. The functional form is

$$\phi(r) = (r^2 + r_0^2)^{1/2} \quad (3.7.5)$$

where  $r_0$  is a scale factor that you get to choose. Multiquadratics are said to be less sensitive to the choice of  $r_0$  than some other functional forms.

In general, both for multiquadratics and for other functions, below,  $r_0$  should be larger than the typical separation of points but smaller than the “outer scale” or feature size of the function that you are interpolating. There can be several orders of magnitude difference between the interpolation accuracy with a good choice for  $r_0$ , versus a poor choice, so it is definitely worth some experimentation. One way to experiment is to construct an RBF interpolator omitting one data point at a time and measuring the interpolation error at the omitted point.

The *inverse multiquadric*

$$\phi(r) = (r^2 + r_0^2)^{-1/2} \quad (3.7.6)$$

gives results that are comparable to the multiquadric, sometimes better.

It might seem odd that a function and its inverse (actually, reciprocal) work about equally well. The explanation is that what really matters is smoothness, and certain properties of the function’s Fourier transform that are not very different between the multiquadric and its reciprocal. The fact that one increases monotonically and the other decreases turns out to be almost irrelevant. However, if you want the extrapolated function to go to zero far from all the data (where an accurate value is impossible anyway), then the inverse multiquadric is a good choice.

The *thin-plate spline* radial basis function is

$$\phi(r) = r^2 \log(r/r_0) \quad (3.7.7)$$

with the limiting value  $\phi(0) = 0$  assumed. This function has some physical justification in the energy minimization problem associated with warping a thin elastic plate. There is no indication that it is generally better than either of the above forms, however.

The *Gaussian* radial basis function is just what you’d expect,

$$\phi(r) = \exp(-\frac{1}{2}r^2/r_0^2) \quad (3.7.8)$$

The interpolation accuracy using Gaussian basis functions can be very sensitive to  $r_0$ , and they are often avoided for this reason. However, for smooth functions and with an optimal  $r_0$ , very high accuracy can be achieved. The Gaussian also will extrapolate any function to zero far from the data, and it gets to zero quickly.

Other functions are also in use, for example those of Wendland [6]. There is a large literature in which the above choices for basis functions are tested against specific functional forms or experimental data sets [1,2,7]. Few, if any, general recommendations emerge. We suggest that you try the alternatives in the order listed above, starting with multiquadratics, and that you not omit experimenting with different choices of the scale parameters  $r_0$ .

The functions discussed are implemented in code as:

```

struct RBF_multiquadric : RBF_fn {
  Instantiate this and send to RBF_interp to get multiquadric interpolation.
  Doub r02;
  RBF_multiquadric(Doub scale=1.) : r02(SQR(scale)) {}
  Constructor argument is the scale factor. See text.
  Doub rbf(Doub r) { return sqrt(SQR(r)+r02); }
};

struct RBF_thinplate : RBF_fn {
  Same as above, but for thin-plate spline.
  Doub r0;
  RBF_thinplate(Doub scale=1.) : r0(scale) {}
  Doub rbf(Doub r) { return r <= 0. ? 0. : SQR(r)*log(r/r0); }
};

struct RBF_gauss : RBF_fn {
  Same as above, but for Gaussian.
  Doub r0;
  RBF_gauss(Doub scale=1.) : r0(scale) {}
  Doub rbf(Doub r) { return exp(-0.5*SQR(r/r0)); }
};

struct RBF_inversemultiquadric : RBF_fn {
  Same as above, but for inverse multiquadric.
  Doub r02;
  RBF_inversemultiquadric(Doub scale=1.) : r02(SQR(scale)) {}
  Doub rbf(Doub r) { return 1./sqrt(SQR(r)+r02); }
};

```

Typical use of the objects in this section should look something like this:

```

Int npts=...,ndim=...
Doub r0=...
MatDoub pts(npts,ndim);
VecDoub y(npts);
...
RBF_multiquadric multiquadric(r0);
RBF_interp myfunc(pts,y,multiquadric,0);

```

followed by any number of interpolation calls,

```

VecDoub pt(ndim);
Doub val;
...
val = myfunc.interp(pt);

```

### 3.7.3 Shepard Interpolation

An interesting special case of normalized radial basis function interpolation (equations 3.7.3 and 3.7.4) occurs if the function  $\phi(r)$  goes to infinity as  $r \rightarrow 0$ , and is finite (e.g., decreasing) for  $r > 0$ . In that case it is easy to see that the weights  $w_i$  are just equal to the respective function values  $y_i$ , and the interpolation formula is simply

$$y(\mathbf{x}) = \frac{\sum_{i=0}^{N-1} y_i \phi(|\mathbf{x} - \mathbf{x}_i|)}{\sum_{i=0}^{N-1} \phi(|\mathbf{x} - \mathbf{x}_i|)} \quad (3.7.9)$$

(with appropriate provision for the limiting case where  $\mathbf{x}$  is equal to one of the  $\mathbf{x}_i$ 's). Note that no solution of linear equations is required. The one-time work is negligible, while the work for each interpolation is  $O(N)$ , tolerable even for very large  $N$ .

Shepard proposed the simple power-law function

$$\phi(r) = r^{-p} \quad (3.7.10)$$

with (typically)  $1 < p \lesssim 3$ , as well as some more complicated functions with different exponents in an inner and outer region (see [8]). You can see that what is going on is basically interpolation by a nearness-weighted average, with nearby points contributing more strongly than distant ones.

Shepard interpolation is rarely as accurate as the well-tuned application of one of the other radial basis functions, above. On the other hand, it is simple, fast, and often just the thing for quick and dirty applications. It, and variants, are thus widely used.

An implementing object is

`interp_rbf.h`

```
struct Shep_interp {
```

Object for Shepard interpolation using `n` points in `dim` dimensions. Call constructor once, then `interp` as many times as desired.

```
    Int dim, n;
    const MatDoub &pts;
    const VecDoub &vals;
    Doub pneg;
```

```
    Shep_interp(MatDoub_I &ptss, VecDoub_I &valss, Doub p=2.)
        : dim(ptss.ncols()), n(ptss.nrows()), pts(ptss),
          vals(valss), pneg(-p) {}
```

Constructor. The  $n \times dim$  matrix `ptss` inputs the data points, the vector `valss` the function values. Set `p` to the desired exponent. The default value is typical.

```
Doub interp(VecDoub_I &pt) {
    Return the interpolated function value at a dim-dimensional point pt.
    Doub r, w, sum=0., sumw=0.;
    if (pt.size() != dim) throw("RBF_interp bad pt size");
    for (Int i=0;i<n;i++) {
        if ((r=rad(&pt[0],&pts[i][0])) == 0.) return vals[i];
        sum += (w = pow(r,pneg));
        sumw += w*vals[i];
    }
    return sumw/sum;
}
```

```
Doub rad(const Doub *p1, const Doub *p2) {
    Euclidean distance.
    Doub sum = 0.;
    for (Int i=0;i<dim;i++) sum += SQR(p1[i]-p2[i]);
    return sqrt(sum);
};
```

### 3.7.4 Interpolation by Kriging

Kriging is a technique named for South African mining engineer D.G. Krige. It is basically a form of linear prediction (§13.6), also known in different communities as *Gauss-Markov estimation* or *Gaussian process regression*.

Kriging can be either an interpolation method or a fitting method. The distinction between the two is whether the fitted/interpolated function goes exactly through all the input data points (interpolation), or whether it allows measurement errors to be specified and then “smooths” to get a statistically better predictor that does not

generally go through the data points (does not “honor the data”). In this section we consider only the former case, that is, interpolation. We will return to the latter case in §15.9.

At this point in the book, it is beyond our scope to derive the equations for kriging. You can turn to §13.6 to get a flavor, and look to references [9,10,11] for details. To use kriging, you must be able to estimate the mean square variation of your function  $y(\mathbf{x})$  as a function of offset distance  $\mathbf{r}$ , a so-called *variogram*,

$$v(\mathbf{r}) \sim \frac{1}{2} \left\langle [y(\mathbf{x} + \mathbf{r}) - y(\mathbf{x})]^2 \right\rangle \quad (3.7.11)$$

where the average is over all  $\mathbf{x}$  with fixed  $\mathbf{r}$ . If this seems daunting, don’t worry. For interpolation, even very crude variogram estimates work fine, and we will give below a routine to estimate  $v(\mathbf{r})$  from your input data points  $\mathbf{x}_i$  and  $y_i = y(\mathbf{x}_i)$ ,  $i = 0, \dots, N - 1$ , automatically. One usually takes  $v(\mathbf{r})$  to be a function only of the magnitude  $r = |\mathbf{r}|$  and writes it as  $v(r)$ .

Let  $v_{ij}$  denote  $v(|\mathbf{x}_i - \mathbf{x}_j|)$ , where  $i$  and  $j$  are input points, and let  $v_{*j}$  denote  $v(|\mathbf{x}_* - \mathbf{x}_j|)$ ,  $\mathbf{x}_*$  being a point at which we want an interpolated value  $y(\mathbf{x}_*)$ . Now define two vectors of length  $N + 1$ ,

$$\begin{aligned} \mathbf{Y} &= (y_0, y_1, \dots, y_{N-1}, 0) \\ \mathbf{V}_* &= (v_{*1}, v_{*2}, \dots, v_{*,N-1}, 1) \end{aligned} \quad (3.7.12)$$

and an  $(N + 1) \times (N + 1)$  symmetric matrix,

$$\mathbf{V} = \begin{pmatrix} v_{00} & v_{01} & \dots & v_{0,N-1} & 1 \\ v_{10} & v_{11} & \dots & v_{1,N-1} & 1 \\ & & \dots & & \dots \\ v_{N-1,0} & v_{N-1,1} & \dots & v_{N-1,N-1} & 1 \\ 1 & 1 & \dots & 1 & 0 \end{pmatrix} \quad (3.7.13)$$

Then the kriging interpolation estimate  $\hat{y}_* \approx y(\mathbf{x}_*)$  is given by

$$\hat{y}_* = \mathbf{V}_* \cdot \mathbf{V}^{-1} \cdot \mathbf{Y} \quad (3.7.14)$$

and its variance is given by

$$\text{Var}(\hat{y}_*) = \mathbf{V}_* \cdot \mathbf{V}^{-1} \cdot \mathbf{V}_* \quad (3.7.15)$$

Notice that if we compute, once, the *LU* decomposition of  $\mathbf{V}$ , and then backsubstitute, once, to get the vector  $\mathbf{V}^{-1} \cdot \mathbf{Y}$ , then the individual interpolations cost only  $O(N)$ : Compute the vector  $\mathbf{V}_*$  and take a vector dot product. On the other hand, every computation of a variance, equation (3.7.15), requires an  $O(N^2)$  backsubstitution.

As an aside (if you have looked ahead to §13.6) the purpose of the extra row and column in  $\mathbf{V}$ , and extra last components in  $\mathbf{V}_*$  and  $\mathbf{Y}$ , is to automatically calculate, and correct for, an appropriately weighted average of the data, and thus to make equation (3.7.14) an unbiased estimator.

Here is an implementation of equations (3.7.12) – (3.7.15). The constructor does the one-time work, while the two overloaded `interp` methods calculate either an interpolated value or else a value and a standard deviation (square root of the variance). You should leave the optional argument `err` set to the default value of `NULL` until you read §15.9.

```

krig.h
template<class T>
struct Krig {
Object for interpolation by kriging, using npt points in ndim dimensions. Call constructor once,
then interp as many times as desired.
    const MatDoub &x;
    const T &vgram;
    Int ndim, npt;
    Doub lastval, lasterr;
    VecDoub y,dstar,vstar,yvi;           Most recently computed value and (if com-
    MatDoub v;                         puted) error.
    LUdcmp *vi;

Krig(MatDoub_I &xx, VecDoub_I &yy, T &vgram, const Doub *err=NULL)
: x(xx),vgram(vgram),npt(xx.nrows()),ndim(xx.ncols()),dstar(npt+1),
vstar(npt+1),v(npt+1,npt+1),y(npt+1),yvi(npt+1) {
Constructor. The npt × ndim matrix xx inputs the data points, the vector yy the function
values. vgram is the variogram function or functor. The argument err is not used for
interpolation; see §15.9.
    Int i,j;
    for (i=0;i<npt;i++) {             Fill Y and V.
        y[i] = yy[i];
        for (j=i;j<npt;j++) {
            v[i][j] = v[j][i] = vgram(rdist(&x[i][0],&x[j][0]));
        }
        v[i][npt] = v[npt][i] = 1.;
    }
    v[npt][npt] = y[npt] = 0.;
    if (err) for (i=0;i<npt;i++) v[i][i] -= SQR(err[i]);    §15.9.
    vi = new LUdcmp(v);
    vi->solve(y,yvi);
}
~Krig() { delete vi; }

Doub interp(VecDoub_I &xstar) {
Return an interpolated value at the point xstar.
    Int i;
    for (i=0;i<npt;i++) vstar[i] = vgram(rdist(&xstar[0],&x[i][0]));
    vstar[npt] = 1.;
    lastval = 0.;
    for (i=0;i<=npt;i++) lastval += yvi[i]*vstar[i];
    return lastval;
}

Doub interp(VecDoub_I &xstar, Doub &esterr) {
Return an interpolated value at the point xstar, and return its estimated error as esterr.

    lastval = interp(xstar);
    vi->solve(vstar,dstar);
    lasterr = 0;
    for (Int i=0;i<=npt;i++) lasterr += dstar[i]*vstar[i];
    esterr = lasterr = sqrt(MAX(0.,lasterr));
    return lastval;
}

Doub rdist(const Doub *x1, const Doub *x2) {
Utility used internally. Cartesian distance between two points.
    Doub d=0.;
    for (Int i=0;i<ndim;i++) d += SQR(x1[i]-x2[i]);
    return sqrt(d);
}
};

The constructor argument vgram, the variogram function, can be either a func-
```

tion or functor (§1.3.3). For interpolation, you can use a `Powvargram` object that fits a simple model

$$v(r) = \alpha r^\beta \quad (3.7.16)$$

where  $\beta$  is considered fixed and  $\alpha$  is fitted by unweighted least squares over all pairs of data points  $i$  and  $j$ . We'll get more sophisticated about variograms in §15.9; but for interpolation, excellent results can be obtained with this simple choice. The value of  $\beta$  should be in the range  $1 \leq \beta < 2$ . A good general choice is 1.5, but for functions with a strong linear trend, you may want to experiment with values as large as 1.99. (The value 2 gives a degenerate matrix and meaningless results.) The optional argument `nug` will be explained in §15.9.

```
struct Powvargram { krig.h
    Functor for variogram  $v(r) = \alpha r^\beta$ , where  $\beta$  is specified,  $\alpha$  is fitted from the data.
    Doub alph, bet, nugsq;

    Powvargram(MatDoub_I &x, VecDoub_I &y, const Doub beta=1.5, const Doub nug=0.)
        : bet(beta), nugsq(nug*nug) {
        Constructor. The  $npt \times ndim$  matrix  $x$  inputs the data points, the vector  $y$  the function
        values,  $\beta$  the value of  $\beta$ . For interpolation, the default value of  $\beta$  is usually adequate.
        For the (rare) use of nug see §15.9.
        Int i,j,k,npt=x.nrows(),ndim=x.ncols();
        Doub rb,num=0.,denom=0.;
        for (i=0;i<npt;i++) for (j=i+1;j<npt;j++) {
            rb = 0.;
            for (k=0;k<ndim;k++) rb += SQR(x[i][k]-x[j][k]);
            rb = pow(rb,0.5*beta);
            num += rb*(0.5*SQR(y[i]-y[j]) - nugsq);
            denom += SQR(rb);
        }
        alph = num/denom;
    }

    Doub operator() (const Doub r) const {return nugsq+alph*pow(r,bet);}
};
```

Sample code for interpolating on a set of data points is

```
MatDoub x(npts,ndim);
VecDoub y(npts), xstar(ndim);
...
Powvargram vgram(x,y);
Krig<Powvargram> krig(x,y,vgram);
```

followed by any number of interpolations of the form

```
ystar = krig.interp(xstar);
```

Be aware that while the interpolated values are quite insensitive to the variogram model, the estimated errors are rather sensitive to it. You should thus consider the error estimates as being order of magnitude only. Since they are also relatively expensive to compute, their value in this application is not great. They will be much more useful in §15.9, when our model includes measurement errors.

### 3.7.5 Curve Interpolation in Multidimensions

A different kind of interpolation, worth a brief mention here, is when you have an ordered set of  $N$  tabulated points in  $n$  dimensions that lie on a one-dimensional curve,  $\mathbf{x}_0, \dots, \mathbf{x}_{N-1}$ , and you want to interpolate other values along the curve. Two

cases worth distinguishing are: (i) The curve is an open curve, so that  $\mathbf{x}_0$  and  $\mathbf{x}_{N-1}$  represent endpoints. (ii) The curve is a closed curve, so that there is an implied curve segment connecting  $\mathbf{x}_{N-1}$  back to  $\mathbf{x}_0$ .

A straightforward solution, using methods already at hand, is first to approximate distance along the curve by the sum of chord lengths between the tabulated points, and then to construct spline interpolations for each of the coordinates,  $0, \dots, n - 1$ , as a function of that parameter. Since the derivative of any single coordinate with respect to arc length can be no greater than 1, it is guaranteed that the spline interpolations will be well-behaved.

Probably 90% of applications require nothing more complicated than the above. If you are in the unhappy 10%, then you will need to learn about *Bézier curves*, *B-splines*, and *interpolating splines* more generally [12,13,14]. For the happy majority, an implementation is

interp\_curve.h

```
struct Curve_interp {
    Object for interpolating a curve specified by n points in dim dimensions.
    Int dim, n, in;
    Bool cls;                                Set if a closed curve.
    MatDoub pts;
    VecDoub s;
    VecDoub ans;
    NRvector<Spline_interp*> srp;

    Curve_interp(MatDoub &ptsin, Bool close=0)
        : n(ptsin.nrows()), dim(ptsin.ncols()), in(close ? 2*n : n),
          cls(close), pts(dim,in), s(in), ans(dim), srp(dim) {
        Constructor. The n × dim matrix ptsin inputs the data points. Input close as 0 for
        an open curve, 1 for a closed curve. (For a closed curve, the last data point should not
        duplicate the first — the algorithm will connect them.)

        Int i,ii,im,j,ofs;
        Doub ss,soff,db,de;
        ofs = close ? n/2 : 0;
        s[0] = 0.;
        for (i=0;i<in;i++) {
            ii = (i-ofs+n) % n;
            im = (ii-1+n) % n;
            for (j=0;j<dim;j++) pts[j][i] = ptsin[ii][j];      Store transpose.
            if (i>0) {                                         Accumulate arc length.
                s[i] = s[i-1] + rad(&ptsin[ii][0],&ptsin[im][0]);
                if (s[i] == s[i-1]) throw("error in Curve_interp");
                Consecutive points may not be identical. For a closed curve, the last data
                point should not duplicate the first.
            }
        }
        ss = close ? s[ofs+n]-s[ofs] : s[n-1]-s[0];  Rescale parameter so that the
        soff = s[ofs];                                interval [0,1] is the whole curve (or one period).
        for (i=0;i<in;i++) s[i] = (s[i]-soff)/ss;   Construct the splines using endpoint derivatives.
        for (j=0;j<dim;j++) {
            db = in < 4 ? 1.e99 : fprime(&s[0],&pts[j][0],1);
            de = in < 4 ? 1.e99 : fprime(&s[in-1],&pts[j][in-1],-1);
            srp[j] = new Spline_interp(s,&pts[j][0],db,de);
        }
    }
    ~Curve_interp() {for (Int j=0;j<dim;j++) delete srp[j);}

    VecDoub &interp(Doub t) {
        Interpolate a point on the stored curve. The point is parameterized by t, in the range [0,1].
        For open curves, values of t outside this range will return extrapolations (dangerous!). For
        closed curves, t is periodic with period 1.
    }
}
```

```

    if (cls) t = t - floor(t);
    for (Int j=0;j<dim;j++) ans[j] = (*srp[j]).interp(t);
    return ans;
}

Doub fprime(Doub *x, Doub *y, Int pm) {
Utility for estimating the derivatives at the endpoints. x and y point to the abscissa and
ordinate of the endpoint. If pm is +1, points to the right will be used (left endpoint); if it
is -1, points to the left will be used (right endpoint). See text, below.
    Doub s1 = x[0]-x[pm*1], s2 = x[0]-x[pm*2], s3 = x[0]-x[pm*3],
        s12 = s1-s2, s13 = s1-s3, s23 = s2-s3;
    return -(s1*s2/(s13*s23*s3))*y[pm*3]+(s1*s3/(s12*s2*s23))*y[pm*2]
        -(s2*s3/(s1*s12*s13))*y[pm*1]+(1./s1+1./s2+1./s3)*y[0];
}

Doub rad(const Doub *p1, const Doub *p2) {
Euclidean distance.
    Doub sum = 0.;
    for (Int i=0;i<dim;i++) sum += SQR(p1[i]-p2[i]);
    return sqrt(sum);
}

};


```

The utility routine `fprime` estimates the derivative of a function at a tabulated abscissa  $x_0$  using four consecutive tabulated abscissa-ordinate pairs,  $(x_0, y_0), \dots, (x_3, y_3)$ . The formula for this, readily derived by power-series expansion, is

$$y'_0 = -C_0 y_0 + C_1 y_1 - C_2 y_2 + C_3 y_3 \quad (3.7.17)$$

where

$$\begin{aligned} C_0 &= \frac{1}{s_1} + \frac{1}{s_2} + \frac{1}{s_3} \\ C_1 &= \frac{s_2 s_3}{s_1(s_2 - s_1)(s_3 - s_1)} \\ C_2 &= \frac{s_1 s_3}{(s_2 - s_1)s_2(s_3 - s_2)} \\ C_3 &= \frac{s_1 s_2}{(s_3 - s_1)(s_3 - s_2)s_3} \end{aligned} \quad (3.7.18)$$

with

$$\begin{aligned} s_1 &\equiv x_1 - x_0 \\ s_2 &\equiv x_2 - x_0 \\ s_3 &\equiv x_3 - x_0 \end{aligned} \quad (3.7.19)$$

#### CITED REFERENCES AND FURTHER READING:

- Buhmann, M.D. 2003, *Radial Basis Functions: Theory and Implementations* (Cambridge, UK: Cambridge University Press).[1]
- Powell, M.J.D. 1992, “The Theory of Radial Basis Function Approximation” in *Advances in Numerical Analysis II: Wavelets, Subdivision Algorithms and Radial Functions*, ed. W. A. Light (Oxford: Oxford University Press), pp. 105–210.[2]
- Wikipedia. 2007+, “Radial Basis Functions,” at <http://en.wikipedia.org/>.[3]
- Beatson, R.K. and Greengard, L. 1997, “A Short Course on Fast Multipole Methods”, in *Wavelets, Multilevel Methods and Elliptic PDEs*, eds. M. Ainsworth, J. Levesley, W. Light, and M. Marletta (Oxford: Oxford University Press), pp. 1–37.[4]

- Beatson, R.K. and Newsam, G.N. 1998, "Fast Evaluation of Radial Basis Functions: Moment-Based Methods" in *SIAM Journal on Scientific Computing*, vol. 19, pp. 1428–1449.[5]
- Wendland, H. 2005, *Scattered Data Approximation* (Cambridge, UK: Cambridge University Press).[6]
- Franke, R. 1982, "Scattered Data Interpolation: Tests of Some Methods," *Mathematics of Computation*, vol. 38, pp. 181–200.[7]
- Shepard, D. 1968, "A Two-dimensional Interpolation Function for Irregularly-spaced Data," in *Proceedings of the 1968 23rd ACM National Conference* (New York: ACM Press), pp. 517–524.[8]
- Cressie, N. 1991, *Statistics for Spatial Data* (New York: Wiley).[9]
- Wackernagel, H. 1998, *Multivariate Geostatistics*, 2nd ed. (Berlin: Springer).[10]
- Rybicki, G.B., and Press, W.H. 1992, "Interpolation, Realization, and Reconstruction of Noisy, Irregularly Sampled Data," *Astrophysical Journal*, vol. 398, pp. 169–176.[11]
- Isaaks, E.H., and Srivastava, R.M. 1989, *Applied Geostatistics* (New York: Oxford University Press).
- Deutsch, C.V., and Journel, A.G. 1992, *GSLIB: Geostatistical Software Library and User's Guide* (New York: Oxford University Press).
- Knott, G.D. 1999, *Interpolating Cubic Splines* (Boston: Birkhäuser).[12]
- De Boor, C. 2001, *A Practical Guide to Splines* (Berlin: Springer).[13]
- Prautzsch, H., Boehm, W., and Paluszny, M. 2002, *Bézier and B-Spline Techniques* (Berlin: Springer).[14]

## 3.8 Laplace Interpolation

In this section we look at a *missing data* or *gridding* problem, namely, how to restore missing or unmeasured values on a regular grid. Evidently some kind of interpolation from the not-missing values is required, but how shall we do this in a principled way?

One good method, already in use at the dawn of the computer age [1,2], is *Laplace interpolation*, sometimes called *Laplace/Poisson interpolation*. The general idea is to find an interpolating function  $y$  that satisfies Laplace's equation in  $n$  dimensions,

$$\nabla^2 y = 0 \quad (3.8.1)$$

wherever there is no data, and which satisfies

$$y(\mathbf{x}_i) = y_i \quad (3.8.2)$$

at all measured data points. Generically, such a function does exist. The reason for choosing Laplace's equation (among all possible partial differential equations, say) is that the solution to Laplace's equation selects, in some sense, the smoothest possible interpolant. In particular, its solution minimizes the integrated square of the gradient,

$$\int_{\Omega} |\nabla y|^2 d\Omega \quad (3.8.3)$$

where  $\Omega$  denotes the  $n$ -dimensional domain of interest. This is a very general idea, and it can be applied to irregular meshes as well as to regular grids. Here, however, we consider only the latter.

For purposes of illustration (and because it is the most useful example) we further specialize to the case of two dimensions, and to the case of a Cartesian grid whose  $x_1$  and  $x_2$  values are evenly spaced — like a checkerboard.

In this geometry, the finite difference approximation to Laplace's equation has a particularly simple form, one that echos the *mean value theorem* for continuous solutions of the Laplace equation: The value of the solution at any free gridpoint (i.e., not a point with a measured value) equals the average of its four Cartesian neighbors. (See §20.0.) Indeed, this already sounds a lot like interpolation.

If  $y_0$  denotes the value at a free point, while  $y_u$ ,  $y_d$ ,  $y_l$ , and  $y_r$  denote the values at its up, down, left, and right neighbors, respectively, then the equation satisfied is

$$y_0 - \frac{1}{4}y_u - \frac{1}{4}y_d - \frac{1}{4}y_l - \frac{1}{4}y_r = 0 \quad (3.8.4)$$

For gridpoints with measured values, on the other hand, a different (simple) equation is satisfied,

$$y_0 = y_{0(\text{measured})} \quad (3.8.5)$$

Note that these nonzero right-hand sides are what make an inhomogeneous, and therefore generally solvable, set of linear equations.

We are not quite done, since we must provide special forms for the top, bottom, left, and right boundaries, and for the four corners. Homogeneous choices that embody “natural” boundary conditions (with no preferred function values) are

$$\begin{aligned} y_0 - \frac{1}{2}y_u - \frac{1}{2}y_d &= 0 && (\text{left and right boundaries}) \\ y_0 - \frac{1}{2}y_l - \frac{1}{2}y_r &= 0 && (\text{top and bottom boundaries}) \\ y_0 - \frac{1}{2}y_r - \frac{1}{2}y_d &= 0 && (\text{top-left corner}) \\ y_0 - \frac{1}{2}y_l - \frac{1}{2}y_d &= 0 && (\text{top-right corner}) \\ y_0 - \frac{1}{2}y_r - \frac{1}{2}y_u &= 0 && (\text{bottom-left corner}) \\ y_0 - \frac{1}{2}y_l - \frac{1}{2}y_u &= 0 && (\text{bottom-right corner}) \end{aligned} \quad (3.8.6)$$

Since every gridpoint corresponds to exactly one of the equations in (3.8.4), (3.8.5), or (3.8.6), we have exactly as many equations as there are unknowns. If the grid is  $M$  by  $N$ , then there are  $MN$  of each. This can be quite a large number; but the equations are evidently very sparse. We solve them by defining a derived class from §2.7's Linbcg base class. You can readily identify all the cases of equations (3.8.4) – (3.8.6) in the code for `atimes`, below.

```
struct Laplace_interp : Linbcg {
    Object for interpolating missing data in a matrix by solving Laplace's equation. Call constructor once, then solve one or more times (see text).
```

```
    MatDoub &mat;
    Int ii,jj;
    Int nn,iter;
    VecDoub b,y,mask;
```

```
Laplace_interp(MatDoub_IO &matrix) : mat(matrix), ii(mat.nrows()), jj(mat.ncols()), nn(ii*jj), iter(0), b(nn), y(nn), mask(nn) {
```

Constructor. Values greater than 1.e99 in the input matrix `mat` are deemed to be missing data. The matrix is not altered until `solve` is called.

```
    Int i,j,k;
    Doub vl = 0.;
```

`interp.laplace.h`

```

for (k=0;k<nn;k++) {
    i = k/jj;
    j = k - i*jj;
    if (mat[i][j] < 1.e99) {
        b[k] = y[k] = vl = mat[i][j];
        mask[k] = 1;
    } else {
        b[k] = 0.;
        y[k] = vl;
        mask[k] = 0;
    }
}
}

```

```

void asolve(VecDoub_I &b, VecDoub_O &x, const Int itrnsp);
void atimes(VecDoub_I &x, VecDoub_O &r, const Int itrnsp);

```

See definitions below. These are the real algorithmic content.

```
Doub solve(Doub tol=1.e-6, Int itmax=-1) {
```

Invoke Linbcg::solve with appropriate arguments. The default argument values will usually work, in which case this routine need be called only once. The original matrix mat is refilled with the interpolated solution.

```

Int i,j,k;
Doub err;
if (itmax <= 0) itmax = 2*MAX(ii,jj);
Linbcg::solve(b,y,1,tol,itmax,iter,err);
for (k=0,i=0;i<ii;i++) for (j=0;j<jj;j++) mat[i][j] = y[k++];
return err;
}
};
```

```
void Laplace_interp::asolve(VecDoub_I &b, VecDoub_O &x, const Int itrnsp) {
```

Diagonal preconditioner. (Diagonal elements all unity.)

```

Int i,n=b.size();
for (i=0;i<n;i++) x[i] = b[i];
}
```

```
void Laplace_interp::atimes(VecDoub_I &x, VecDoub_O &r, const Int itrnsp) {
```

Sparse matrix, and matrix transpose, multiply. This routine embodies eqs. (3.8.4), (3.8.5), and (3.8.6).

```

Int i,j,k,n=r.size(),jjt,it;
Doub del;
for (k=0;k<n;k++) r[k] = 0.;
for (k=0;k<n;k++) {
    i = k/jj;
    j = k - i*jj;
    if (mask[k]) {                                Measured point, eq. (3.8.5).
        r[k] += x[k];
    } else if (i>0 && i<ii-1 && j>0 && j<jj-1) {   Interior point, eq. (3.8.4).
        if (itrnsp) {
            r[k] += x[k];
            del = -0.25*x[k];
            r[k-1] += del;
            r[k+1] += del;
            r[k-jj] += del;
            r[k+jj] += del;
        } else {
            r[k] = x[k] - 0.25*(x[k-1]+x[k+1]+x[k+jj]+x[k-jj]);
        }
    } else if (i>0 && i<ii-1) {                  Left or right edge, eq. (3.8.6).
        if (itrnsp) {
            r[k] += x[k];
            del = -0.5*x[k];
            r[k-jj] += del;
        }
    }
}
```

```

        r[k+jj] += del;
    } else {
        r[k] = x[k] - 0.5*(x[k+jj]+x[k-jj]);
    }
} else if (j>0 && j<jj-1) {                                Top or bottom edge, eq. (3.8.6).
    if (itrnsp) {
        r[k] += x[k];
        del = -0.5*x[k];
        r[k-1] += del;
        r[k+1] += del;
    } else {
        r[k] = x[k] - 0.5*(x[k+1]+x[k-1]);
    }
} else {                                                     Corners, eq. (3.8.6).
    jjt = i==0 ? jj : -jj;
    it = j==0 ? 1 : -1;
    if (itrnsp) {
        r[k] += x[k];
        del = -0.5*x[k];
        r[k+jjt] += del;
        r[k+it] += del;
    } else {
        r[k] = x[k] - 0.5*(x[k+jjt]+x[k+it]);
    }
}
}
}
}
}

```

Usage is quite simple. Just fill a matrix with function values where they are known, and with 1.e99 where they are not; send the matrix to the constructor; and call the `solve` routine. The missing values will be interpolated. The default arguments should serve for most cases.

```

Int m=...,n=...
MatDoub mat(m,n);
...
Laplace_interp mylaplace(mat);
mylaplace.solve();

```

Quite decent results are obtained for smooth functions on  $300 \times 300$  matrices in which a random 10% of gridpoints have known function values, with 90% interpolated. However, since compute time scales as  $MN \max(M, N)$  (that is, as the cube), this is not a method to use for much larger matrices, unless you break them up into overlapping tiles. If you experience convergence difficulties, then you should call `solve`, with appropriate nondefault arguments, several times in succession, and look at the returned error estimate after each call returns.

### 3.8.1 Minimum Curvature Methods

Laplace interpolation has a tendency to yield cone-like cusps around any small islands of known data points that are surrounded by a sea of unknowns. The reason is that, in two dimensions, the solution of Laplace's equation near a point source is logarithmically singular. When the known data is spread fairly evenly (if randomly) across the grid, this is not generally a problem. *Minimum curvature methods* deal with the problem at a more fundamental level by being based on the biharmonic equation

$$\nabla(\nabla y) = 0 \quad (3.8.7)$$

instead of Laplace's equation. Solutions of the biharmonic equation minimize the integrated square of the curvature,

$$\int_{\Omega} |\nabla^2 y|^2 d\Omega \quad (3.8.8)$$

Minimum curvature methods are widely used in the earth-science community [3,4].

The references give a variety of other methods that can be used for missing data interpolation and gridding.

#### CITED REFERENCES AND FURTHER READING:

- Noma, A.A. and Misulia, M.G. 1959, "Programming Topographic Maps for Automatic Terrain Model Construction," *Surveying and Mapping*, vol. 19, pp. 355–366.[1]
- Crain, I.K. 1970, "Computer Interpolation and Contouring of Two-dimensional Data: a Review," *Geoexploration*, vol. 8, pp. 71–86.[2]
- Burrough, P.A. 1998, *Principles of Geographical Information Systems*, 2nd ed. (Oxford, UK: Clarendon Press)
- Watson, D.F. 1982, *Contouring: A Guide to the Analysis and Display of Spatial Data* (Oxford, UK: Pergamon).
- Briggs, I.C. 1974, "Machine Contouring Using Minimum Curvature," *Geophysics*, vol. 39, pp. 39–48.[3]
- Smith, W.H.F. and Wessel, P. 1990, "Gridding With Continuous Curvature Splines in Tension," *Geophysics*, vol. 55, pp. 293–305.[4]

# Integration of Functions

## 4.0 Introduction

Numerical integration, which is also called *quadrature*, has a history extending back to the invention of calculus and before. The fact that integrals of elementary functions could not, in general, be computed analytically, while derivatives *could* be, served to give the field a certain panache, and to set it a cut above the arithmetic drudgery of numerical analysis during the whole of the 18th and 19th centuries.

With the invention of automatic computing, quadrature became just one numerical task among many, and not a very interesting one at that. Automatic computing, even the most primitive sort involving desk calculators and rooms full of “computers” (that were, until the 1950s, people rather than machines), opened to feasibility the much richer field of numerical integration of differential equations. Quadrature is merely the simplest special case: The evaluation of the integral

$$I = \int_a^b f(x)dx \quad (4.0.1)$$

is precisely equivalent to solving for the value  $I \equiv y(b)$  the differential equation

$$\frac{dy}{dx} = f(x) \quad (4.0.2)$$

with the boundary condition

$$y(a) = 0 \quad (4.0.3)$$

Chapter 17 of this book deals with the numerical integration of differential equations. In that chapter, much emphasis is given to the concept of “variable” or “adaptive” choices of stepsize. We will not, therefore, develop that material here. If the function that you propose to integrate is sharply concentrated in one or more peaks, or if its shape is not readily characterized by a single length scale, then it is likely that you should cast the problem in the form of (4.0.2) – (4.0.3) and use the methods of Chapter 17. (But take a look at §4.7 first.)

The quadrature methods in this chapter are based, in one way or another, on the obvious device of adding up the value of the integrand at a sequence of abscissas

within the range of integration. The game is to obtain the integral as accurately as possible with the smallest number of function evaluations of the integrand. Just as in the case of interpolation (Chapter 3), one has the freedom to choose methods of various *orders*, with higher order sometimes, but not always, giving higher accuracy. *Romberg integration*, which is discussed in §4.3, is a general formalism for making use of integration methods of a variety of different orders, and we recommend it highly.

Apart from the methods of this chapter and of Chapter 17, there are yet other methods for obtaining integrals. One important class is based on function approximation. We discuss explicitly the integration of functions by Chebyshev approximation (*Clenshaw-Curtis quadrature*) in §5.9. Although not explicitly discussed here, you ought to be able to figure out how to do *cubic spline quadrature* using the output of the routine `spline` in §3.3. (Hint: Integrate equation 3.3.3 over  $x$  analytically. See [1].)

Some integrals related to Fourier transforms can be calculated using the fast Fourier transform (FFT) algorithm. This is discussed in §13.9. A related problem is the evaluation of integrals with long oscillatory tails. This is discussed at the end of §5.3.

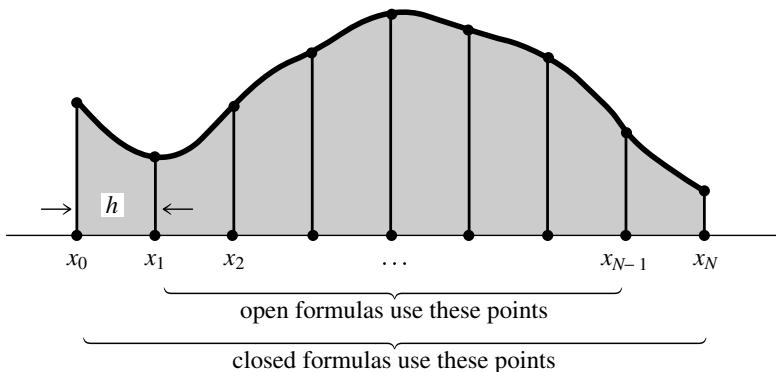
Multidimensional integrals are a whole 'nother multidimensional bag of worms. Section 4.8 is an introductory discussion in this chapter; the important technique of *Monte Carlo integration* is treated in Chapter 7.

#### CITED REFERENCES AND FURTHER READING:

- Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Chapter 2.
- Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), Chapter 7.
- Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), Chapter 4.
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 3.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), Chapter 4.
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §7.4.
- Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 5.
- Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.2, p. 89.[1]
- Davis, P., and Rabinowitz, P. 1984, *Methods of Numerical Integration*, 2nd ed. (Orlando, FL: Academic Press).

## 4.1 Classical Formulas for Equally Spaced Abscissas

Where would any book on numerical analysis be without Mr. Simpson and his “rule”? The classical formulas for integrating a function whose value is known at



**Figure 4.1.1.** Quadrature formulas with equally spaced abscissas compute the integral of a function between  $x_0$  and  $x_N$ . Closed formulas evaluate the function on the boundary points, while open formulas refrain from doing so (useful if the evaluation algorithm breaks down on the boundary points).

equally spaced steps have a certain elegance about them, and they are redolent with historical association. Through them, the modern numerical analyst communes with the spirits of his or her predecessors back across the centuries, as far as the time of Newton, if not farther. Alas, times *do* change; with the exception of two of the most modest formulas (“extended trapezoidal rule,” equation 4.1.11, and “extended midpoint rule,” equation 4.1.19; see §4.2), the classical formulas are almost entirely useless. They are museum pieces, but beautiful ones; we now enter the museum. (You can skip to §4.2 if you are not touristically inclined.)

Some notation: We have a sequence of abscissas, denoted  $x_0, x_1, \dots, x_{N-1}, x_N$ , that are spaced apart by a constant step  $h$ ,

$$x_i = x_0 + ih \quad i = 0, 1, \dots, N \quad (4.1.1)$$

A function  $f(x)$  has known values at the  $x_i$ 's,

$$f(x_i) \equiv f_i \quad (4.1.2)$$

We want to integrate the function  $f(x)$  between a lower limit  $a$  and an upper limit  $b$ , where  $a$  and  $b$  are each equal to one or the other of the  $x_i$ 's. An integration formula that uses the value of the function at the endpoints,  $f(a)$  or  $f(b)$ , is called a *closed* formula. Occasionally, we want to integrate a function whose value at one or both endpoints is difficult to compute (e.g., the computation of  $f$  goes to a limit of zero over zero there, or worse yet has an integrable singularity there). In this case we want an *open* formula, which estimates the integral using only  $x_i$ 's strictly *between*  $a$  and  $b$  (see Figure 4.1.1).

The basic building blocks of the classical formulas are rules for integrating a function over a small number of intervals. As that number increases, we can find rules that are exact for polynomials of increasingly high order. (Keep in mind that higher order does not always imply higher accuracy in real cases.) A sequence of such closed formulas is now given.

### 4.1.1 Closed Newton-Cotes Formulas

*Trapezoidal rule:*

$$\int_{x_0}^{x_1} f(x)dx = h \left[ \frac{1}{2}f_0 + \frac{1}{2}f_1 \right] + O(h^3 f'') \quad (4.1.3)$$

Here the error term  $O()$  signifies that the true answer differs from the estimate by an amount that is the product of some numerical coefficient times  $h^3$  times the value of the function's second derivative somewhere in the interval of integration. The coefficient is knowable, and it can be found in all the standard references on this subject. The point at which the second derivative is to be evaluated is, however, unknowable. If we knew it, we could evaluate the function there and have a higher-order method! Since the product of a knowable and an unknowable is unknowable, we will streamline our formulas and write only  $O()$ , instead of the coefficient.

Equation (4.1.3) is a two-point formula ( $x_0$  and  $x_1$ ). It is exact for polynomials up to and including degree 1, i.e.,  $f(x) = x$ . One anticipates that there is a three-point formula exact up to polynomials of degree 2. This is true; moreover, by a cancellation of coefficients due to left-right symmetry of the formula, the three-point formula is exact for polynomials up to and including degree 3, i.e.,  $f(x) = x^3$ .

*Simpson's rule:*

$$\int_{x_0}^{x_2} f(x)dx = h \left[ \frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{1}{3}f_2 \right] + O(h^5 f^{(4)}) \quad (4.1.4)$$

Here  $f^{(4)}$  means the fourth derivative of the function  $f$  evaluated at an unknown place in the interval. Note also that the formula gives the integral over an interval of size  $2h$ , so the coefficients add up to 2.

There is no lucky cancellation in the four-point formula, so it is also exact for polynomials up to and including degree 3.

*Simpson's  $\frac{3}{8}$  rule:*

$$\int_{x_0}^{x_3} f(x)dx = h \left[ \frac{3}{8}f_0 + \frac{9}{8}f_1 + \frac{9}{8}f_2 + \frac{3}{8}f_3 \right] + O(h^5 f^{(4)}) \quad (4.1.5)$$

The five-point formula again benefits from a cancellation:

*Bode's rule:*

$$\int_{x_0}^{x_4} f(x)dx = h \left[ \frac{14}{45}f_0 + \frac{64}{45}f_1 + \frac{24}{45}f_2 + \frac{64}{45}f_3 + \frac{14}{45}f_4 \right] + O(h^7 f^{(6)}) \quad (4.1.6)$$

This is exact for polynomials up to and including degree 5.

At this point the formulas stop being named after famous personages, so we will not go any further. Consult [1] for additional formulas in the sequence.

### 4.1.2 Extrapolative Formulas for a Single Interval

We are going to depart from historical practice for a moment. Many texts would give, at this point, a sequence of “Newton-Cotes Formulas of Open Type.” Here is

an example:

$$\int_{x_0}^{x_5} f(x)dx = h \left[ \frac{55}{24} f_1 + \frac{5}{24} f_2 + \frac{5}{24} f_3 + \frac{55}{24} f_4 \right] + O(h^5 f^{(4)})$$

Notice that the integral from  $a = x_0$  to  $b = x_5$  is estimated, using only the interior points  $x_1, x_2, x_3, x_4$ . In our opinion, formulas of this type are not useful for the reasons that (i) they cannot usefully be strung together to get “extended” rules, as we are about to do with the closed formulas, and (ii) for all other possible uses they are dominated by the Gaussian integration formulas, which we will introduce in §4.6.

Instead of the Newton-Cotes open formulas, let us set out the formulas for estimating the integral in the single interval from  $x_0$  to  $x_1$ , using values of the function  $f$  at  $x_1, x_2, \dots$ . These will be useful building blocks later for the “extended” open formulas.

$$\int_{x_0}^{x_1} f(x)dx = h[f_1] + O(h^2 f') \quad (4.1.7)$$

$$\int_{x_0}^{x_1} f(x)dx = h \left[ \frac{3}{2} f_1 - \frac{1}{2} f_2 \right] + O(h^3 f'') \quad (4.1.8)$$

$$\int_{x_0}^{x_1} f(x)dx = h \left[ \frac{23}{12} f_1 - \frac{16}{12} f_2 + \frac{5}{12} f_3 \right] + O(h^4 f^{(3)}) \quad (4.1.9)$$

$$\int_{x_0}^{x_1} f(x)dx = h \left[ \frac{55}{24} f_1 - \frac{59}{24} f_2 + \frac{37}{24} f_3 - \frac{9}{24} f_4 \right] + O(h^5 f^{(4)}) \quad (4.1.10)$$

Perhaps a word here would be in order about how formulas like the above can be derived. There are elegant ways, but the most straightforward is to write down the basic form of the formula, replacing the numerical coefficients with unknowns, say  $p, q, r, s$ . Without loss of generality take  $x_0 = 0$  and  $x_1 = 1$ , so  $h = 1$ . Substitute in turn for  $f(x)$  (and for  $f_1, f_2, f_3, f_4$ ) the functions  $f(x) = 1$ ,  $f(x) = x$ ,  $f(x) = x^2$ , and  $f(x) = x^3$ . Doing the integral in each case reduces the left-hand side to a number and the right-hand side to a linear equation for the unknowns  $p, q, r, s$ . Solving the four equations produced in this way gives the coefficients.

### 4.1.3 Extended Formulas (Closed)

If we use equation (4.1.3)  $N - 1$  times to do the integration in the intervals  $(x_0, x_1), (x_1, x_2), \dots, (x_{N-2}, x_{N-1})$  and then add the results, we obtain an “extended” or “composite” formula for the integral from  $x_0$  to  $x_{N-1}$ .

*Extended trapezoidal rule:*

$$\begin{aligned} \int_{x_0}^{x_{N-1}} f(x)dx &= h \left[ \frac{1}{2} f_0 + f_1 + f_2 + \right. \\ &\quad \left. \dots + f_{N-2} + \frac{1}{2} f_{N-1} \right] + O\left(\frac{(b-a)^3 f''}{N^2}\right) \end{aligned} \quad (4.1.11)$$

Here we have written the error estimate in terms of the interval  $b - a$  and the number of points  $N$  instead of in terms of  $h$ . This is clearer, since one is usually holding  $a$  and

$b$  fixed and wanting to know, e.g., how much the error will be decreased by taking twice as many steps (in this case, it is by a factor of 4). In subsequent equations we will show *only* the scaling of the error term with the number of steps.

For reasons that will not become clear until §4.2, equation (4.1.11) is in fact the most important equation in this section; it is the basis for most practical quadrature schemes.

The *extended formula of order  $1/N^3$*  is

$$\begin{aligned} \int_{x_0}^{x_{N-1}} f(x)dx = h &\left[ \frac{5}{12}f_0 + \frac{13}{12}f_1 + f_2 + f_3 + \right. \\ &\left. \cdots + f_{N-3} + \frac{13}{12}f_{N-2} + \frac{5}{12}f_{N-1} \right] + O\left(\frac{1}{N^3}\right) \end{aligned} \quad (4.1.12)$$

(We will see in a moment where this comes from.)

If we apply equation (4.1.4) to successive, nonoverlapping *pairs* of intervals, we get the *extended Simpson's rule*:

$$\begin{aligned} \int_{x_0}^{x_{N-1}} f(x)dx = h &\left[ \frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \right. \\ &\left. \cdots + \frac{2}{3}f_{N-3} + \frac{4}{3}f_{N-2} + \frac{1}{3}f_{N-1} \right] + O\left(\frac{1}{N^4}\right) \end{aligned} \quad (4.1.13)$$

Notice that the  $2/3$ ,  $4/3$  alternation continues throughout the interior of the evaluation. Many people believe that the wobbling alternation somehow contains deep information about the integral of their function that is not apparent to mortal eyes. In fact, the alternation is an artifact of using the building block (4.1.4). Another extended formula with the same order as Simpson's rule is

$$\begin{aligned} \int_{x_0}^{x_{N-1}} f(x)dx = h &\left[ \frac{3}{8}f_0 + \frac{7}{6}f_1 + \frac{23}{24}f_2 + f_3 + f_4 + \right. \\ &\left. \cdots + f_{N-5} + f_{N-4} + \frac{23}{24}f_{N-3} + \frac{7}{6}f_{N-2} + \frac{3}{8}f_{N-1} \right] \\ &+ O\left(\frac{1}{N^4}\right) \end{aligned} \quad (4.1.14)$$

This equation is constructed by fitting cubic polynomials through successive groups of four points; we defer details to §19.3, where a similar technique is used in the solution of integral equations. We can, however, tell you where equation (4.1.12) came from. It is Simpson's extended rule, averaged with a modified version of itself in which the first and last steps are done with the trapezoidal rule (4.1.3). The trapezoidal step is *two* orders lower than Simpson's rule; however, its contribution to the integral goes down as an additional power of  $N$  (since it is used only twice, not  $N$  times). This makes the resulting formula of degree *one* less than Simpson.

#### 4.1.4 Extended Formulas (Open and Semi-Open)

We can construct open and semi-open extended formulas by adding the closed formulas (4.1.11) – (4.1.14), evaluated for the second and subsequent steps, to the

extrapolative open formulas for the first step, (4.1.7) – (4.1.10). As discussed immediately above, it is consistent to use an end step that is of one order lower than the (repeated) interior step. The resulting formulas for an interval open at both ends are as follows.

Equations (4.1.7) and (4.1.11) give

$$\int_{x_0}^{x_{N-1}} f(x)dx = h \left[ \frac{3}{2}f_1 + f_2 + f_3 + \cdots + f_{N-3} + \frac{3}{2}f_{N-2} \right] + O\left(\frac{1}{N^2}\right) \quad (4.1.15)$$

Equations (4.1.8) and (4.1.12) give

$$\begin{aligned} \int_{x_0}^{x_{N-1}} f(x)dx = h & \left[ \frac{23}{12}f_1 + \frac{7}{12}f_2 + f_3 + f_4 + \right. \\ & \left. \cdots + f_{N-4} + \frac{7}{12}f_{N-3} + \frac{23}{12}f_{N-2} \right] + O\left(\frac{1}{N^3}\right) \end{aligned} \quad (4.1.16)$$

Equations (4.1.9) and (4.1.13) give

$$\begin{aligned} \int_{x_0}^{x_{N-1}} f(x)dx = h & \left[ \frac{27}{12}f_1 + 0 + \frac{13}{12}f_3 + \frac{4}{3}f_4 + \right. \\ & \left. \cdots + \frac{4}{3}f_{N-5} + \frac{13}{12}f_{N-4} + 0 + \frac{27}{12}f_{N-2} \right] + O\left(\frac{1}{N^4}\right) \end{aligned} \quad (4.1.17)$$

The interior points alternate  $4/3$  and  $2/3$ . If we want to avoid this alternation, we can combine equations (4.1.9) and (4.1.14), giving

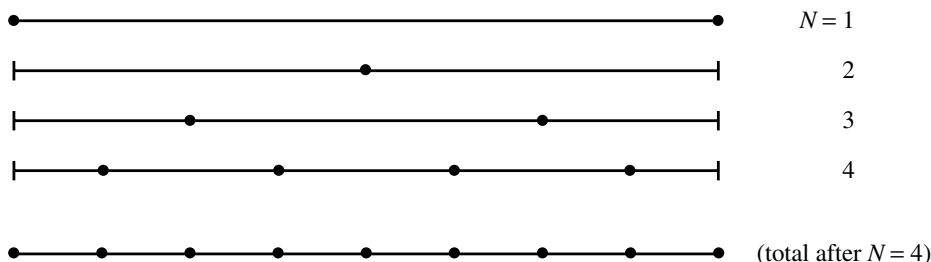
$$\begin{aligned} \int_{x_0}^{x_{N-1}} f(x)dx = h & \left[ \frac{55}{24}f_1 - \frac{1}{6}f_2 + \frac{11}{8}f_3 + f_4 + f_5 + f_6 + \right. \\ & \left. \cdots + f_{N-6} + f_{N-5} + \frac{11}{8}f_{N-4} - \frac{1}{6}f_{N-3} + \frac{55}{24}f_{N-2} \right] \\ & + O\left(\frac{1}{N^4}\right) \end{aligned} \quad (4.1.18)$$

We should mention in passing another extended open formula, for use where the limits of integration are located halfway between tabulated abscissas. This one is known as the *extended midpoint rule* and is accurate to the same order as (4.1.15):

$$\int_{x_0}^{x_{N-1}} f(x)dx = h[f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-5/2} + f_{N-3/2}] + O\left(\frac{1}{N^2}\right) \quad (4.1.19)$$

There are also formulas of higher order for this situation, but we will refrain from giving them.

The *semi-open formulas* are just the obvious combinations of equations (4.1.11) – (4.1.14) with (4.1.15) – (4.1.18), respectively. At the closed end of the integration, use the weights from the former equations; at the open end, use the weights from



**Figure 4.2.1.** Sequential calls to the routine `Trapzd` incorporate the information from previous calls and evaluate the integrand only at those new points necessary to refine the grid. The bottom line shows the totality of function evaluations after the fourth call. The routine `qsimp`, by weighting the intermediate results, transforms the trapezoid rule into Simpson's rule with essentially no additional overhead.

the latter equations. One example should give the idea, the formula with error term decreasing as  $1/N^3$ , which is closed on the right and open on the left:

$$\int_{x_0}^{x_{N-1}} f(x)dx = h \left[ \frac{23}{12}f_1 + \frac{7}{12}f_2 + f_3 + f_4 + \cdots + f_{N-3} + \frac{13}{12}f_{N-2} + \frac{5}{12}f_{N-1} \right] + O\left(\frac{1}{N^3}\right) \quad (4.1.20)$$

#### CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, §25.4.[1]

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), §7.1.

## 4.2 Elementary Algorithms

Our starting point is equation (4.1.11), the extended trapezoidal rule. There are two facts about the trapezoidal rule that make it the starting point for a variety of algorithms. One fact is rather obvious, while the second is rather “deep.”

The obvious fact is that, for a fixed function  $f(x)$  to be integrated between fixed limits  $a$  and  $b$ , one can double the number of intervals in the extended trapezoidal rule without losing the benefit of previous work. The coarsest implementation of the trapezoidal rule is to average the function at its endpoints  $a$  and  $b$ . The first stage of refinement is to add to this average the value of the function at the halfway point. The second stage of refinement is to add the values at the  $1/4$  and  $3/4$  points. And so on (see Figure 4.2.1).

As we will see, a number of elementary quadrature algorithms involve adding successive stages of refinement. It is convenient to encapsulate this feature in a Quadrature structure:

```

struct Quadrature{                                     quadrature.h
Abstract base class for elementary quadrature algorithms.
  Int n;                                         Current level of refinement.
  virtual Doub next() = 0;
Returns the value of the integral at the nth stage of refinement. The function next() must
be defined in the derived class.
};

```

Then the Trapzd structure is derived from this as follows:

```

template<class T>                               quadrature.h
struct Trapzd : Quadrature {
Routine implementing the extended trapezoidal rule.
  Doub a,b,s;                                Limits of integration and current value of integral.
  T &func;
  Trapzd() {};
  Trapzd(T &funcc, const Doub aa, const Doub bb) :
    func(funcc), a(aa), b(bb) {n=0;}
  The constructor takes as inputs func, the function or functor to be integrated between
  limits a and b, also input.
  Doub next() {
  Returns the nth stage of refinement of the extended trapezoidal rule. On the first call (n=1),
  the routine returns the crudest estimate of  $\int_a^b f(x)dx$ . Subsequent calls set n=2,3,... and
  improve the accuracy by adding  $2^{n-2}$  additional interior points.
    Doub x,tnm,sum,del;
    Int it,j;
    n++;
    if (n == 1) {
      return (s=0.5*(b-a)*(func(a)+func(b)));
    } else {
      for (it=1,j=1;j<n-1;j++) it <<= 1;
      tnm=it;
      del=(b-a)/tnm;                           This is the spacing of the points to be added.
      x=a+0.5*del;
      for (sum=0.0,j=0;j<it;j++,x+=del) sum += func(x);
      s=0.5*(s+(b-a)*sum/tnm);                 This replaces s by its refined value.
      return s;
    }
  }
};

```

Note that Trapzd is templated on the whole struct and does not just contain a templated function. This is necessary because it retains a reference to the supplied function or functor as a member variable.

The Trapzd structure is a workhorse that can be harnessed in several ways. The simplest and crudest is to integrate a function by the extended trapezoidal rule where you know in advance (we can't imagine how!) the number of steps you want. If you want  $2^M + 1$ , you can accomplish this by the fragment

```

Ftor func;                                     Functor func here has no parameters.
Trapzd<Ftor> s(func,a,b);
for(j=1;j<=m+1;j++) val=s.next();

```

with the answer returned as val. Here Ftor is a functor containing the function to be integrated.

Much better, of course, is to refine the trapezoidal rule until some specified degree of accuracy has been achieved. A function for this is

quadrature.h

```

template<class T>
Doub qtrap(T &func, const Doub a, const Doub b, const Doub eps=1.0e-10) {
    Returns the integral of the function or functor func from a to b. The constants EPS can be
    set to the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
    allowed number of steps. Integration is performed by the trapezoidal rule.
    const Int JMAX=20;
    Doub s,olds=0.0;           Initial value of olds is arbitrary.
    Trapzd<T> t(func,a,b);
    for (Int j=0;j<JMAX;j++) {
        s=t.next();
        if (j > 5)            Avoid spurious early convergence.
            if (abs(s-olds) < eps*abs(olds) ||
                (s == 0.0 && olds == 0.0)) return s;
        olds=s;
    }
    throw("Too many steps in routine qtrap");
}

```

The optional argument `eps` sets the desired fractional accuracy. Unsophisticated as it is, routine `qtrap` is in fact a fairly robust way of doing integrals of functions that are not very smooth. Increased sophistication will usually translate into a higher-order method whose efficiency will be greater only for sufficiently smooth integrands. `qtrap` is the method of choice, e.g., for an integrand that is a function of a variable that is linearly interpolated between measured data points. Be sure that you do not require too stringent an `eps`, however: If `qtrap` takes too many steps in trying to achieve your required accuracy, accumulated roundoff errors may start increasing, and the routine may never converge. A value of  $10^{-10}$  or even smaller is usually no problem in double precision when the convergence is moderately rapid, but not otherwise. (Of course, very few problems really require such precision.)

We come now to the “deep” fact about the extended trapezoidal rule, equation (4.1.11). It is this: The error of the approximation, which begins with a term of order  $1/N^2$ , is in fact *entirely even* when expressed in powers of  $1/N$ . This follows directly from the *Euler-Maclaurin summation formula*,

$$\begin{aligned} \int_{x_0}^{x_{N-1}} f(x)dx &= h \left[ \frac{1}{2} f_0 + f_1 + f_2 + \cdots + f_{N-2} + \frac{1}{2} f_{N-1} \right] \\ &\quad - \frac{B_2 h^2}{2!} (f'_{N-1} - f'_0) - \cdots - \frac{B_{2k} h^{2k}}{(2k)!} (f_{N-1}^{(2k-1)} - f_0^{(2k-1)}) - \cdots \end{aligned} \tag{4.2.1}$$

Here  $B_{2k}$  is a *Bernoulli number*, defined by the generating function

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!} \tag{4.2.2}$$

with the first few even values (odd values vanish except for  $B_1 = -1/2$ )

$$\begin{aligned} B_0 &= 1 & B_2 &= \frac{1}{6} & B_4 &= -\frac{1}{30} & B_6 &= \frac{1}{42} \\ B_8 &= -\frac{1}{30} & B_{10} &= \frac{5}{66} & B_{12} &= -\frac{691}{2730} \end{aligned} \tag{4.2.3}$$

Equation (4.2.1) is not a convergent expansion, but rather only an asymptotic expansion whose error when truncated at any point is always less than twice the magnitude

of the first neglected term. The reason that it is not convergent is that the Bernoulli numbers become very large, e.g.,

$$B_{50} = \frac{495057205241079648212477525}{66}$$

The key point is that only even powers of  $h$  occur in the error series of (4.2.1). This fact is not, in general, shared by the higher-order quadrature rules in §4.1. For example, equation (4.1.12) has an error series beginning with  $O(1/N^3)$ , but continuing with all subsequent powers of  $N$ :  $1/N^4, 1/N^5$ , etc.

Suppose we evaluate (4.1.11) with  $N$  steps, getting a result  $S_N$ , and then again with  $2N$  steps, getting a result  $S_{2N}$ . (This is done by any two consecutive calls of Trapzd.) The leading error term in the second evaluation will be 1/4 the size of the error in the first evaluation. Therefore the combination

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N \quad (4.2.4)$$

will cancel out the leading order error term. But there *is* no error term of order  $1/N^3$ , by (4.2.1). The surviving error is of order  $1/N^4$ , the same as Simpson's rule. In fact, it should not take long for you to see that (4.2.4) is *exactly* Simpson's rule (4.1.13), alternating 2/3's, 4/3's, and all. This is the preferred method for evaluating that rule, and we can write it as a routine exactly analogous to qtrap above:

```
template<class T>
Doub qsimp(T &func, const Doub a, const Doub b, const Doub eps=1.0e-10) {
    Returns the integral of the function or functor func from a to b. The constants EPS can be
    set to the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
    allowed number of steps. Integration is performed by Simpson's rule.
    const Int JMAX=20;
    Doub s,st,ost=0.0,os=0.0;
    Trapzd<T> t(func,a,b);
    for (Int j=0;j<JMAX;j++) {
        st=t.next();
        s=(4.0*st-ost)/3.0;           Compare equation (4.2.4), above.
        if (j > 5)                  Avoid spurious early convergence.
            if (abs(s-os) < eps*abs(os) ||
                (s == 0.0 && os == 0.0)) return s;
        os=s;
        ost=st;
    }
    throw("Too many steps in routine qsimp");
}
```

quadrature.h

The routine qsimp will in general be more efficient than qtrap (i.e., require fewer function evaluations) when the function to be integrated has a finite fourth derivative (i.e., a continuous third derivative). The combination of qsimp and its necessary workhorse Trapzd is a good one for light-duty work.

#### CITED REFERENCES AND FURTHER READING:

- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §3.1.
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §7.4.1 – §7.4.2.
- Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.

## 4.3 Romberg Integration

We can view Romberg's method as the natural generalization of the routine `qsimp` in the last section to integration schemes that are of higher order than Simpson's rule. The basic idea is to use the results from  $k$  successive refinements of the extended trapezoidal rule (implemented in `trapzd`) to remove all terms in the error series up to but not including  $O(1/N^{2k})$ . The routine `qsimp` is the case of  $k = 2$ . This is one example of a very general idea that goes by the name of *Richardson's deferred approach to the limit*: Perform some numerical algorithm for various values of a parameter  $h$ , and then extrapolate the result to the continuum limit  $h = 0$ .

Equation (4.2.4), which subtracts off the leading error term, is a special case of polynomial extrapolation. In the more general Romberg case, we can use Neville's algorithm (see §3.2) to extrapolate the successive refinements to zero stepsize. Neville's algorithm can in fact be coded very concisely within a Romberg integration routine. For clarity of the program, however, it seems better to do the extrapolation by a function call to `Poly_interp::rawinterp`, as given in §3.2.

`romberg.h`

```
template <class T>
Doub qromb(T &func, Doub a, Doub b, const Doub eps=1.0e-10) {
    Returns the integral of the function or functor func from a to b. Integration is performed by
    Romberg's method of order 2K, where, e.g., K=2 is Simpson's rule.

    const Int JMAX=20, JMAXP=JMAX+1, K=5;
    Here EPS is the fractional accuracy desired, as determined by the extrapolation error es-
    timate; JMAX limits the total number of steps; K is the number of points used in the
    extrapolation.
    VecDoub s(JMAX),h(JMAXP);           These store the successive trapezoidal approxi-
    Poly_interp polint(h,s,K);           mations and their relative stepsizes.
    h[0]=1.0;
    Trapzd<T> t(func,a,b);
    for (Int j=1;j<=JMAX;j++) {
        s[j-1]=t.next();
        if (j >= K) {
            Doub ss=polint.rawinterp(j-K,0.0);
            if (abs(polint.dy) <= eps*abs(ss)) return ss;
        }
        h[j]=0.25*h[j-1];
    }
    throw("Too many steps in routine qromb");
}
```

The routine `qromb` is quite powerful for sufficiently smooth (e.g., analytic) integrands, integrated over intervals that contain no singularities, and where the endpoints are also nonsingular. `qromb`, in such circumstances, takes many, *many* fewer function evaluations than either of the routines in §4.2. For example, the integral

$$\int_0^2 x^4 \log(x + \sqrt{x^2 + 1}) dx$$

converges (with parameters as shown above) on the second extrapolation, after just 6 calls to `trapzd`, while `qsimp` requires 11 calls (32 times as many evaluations of the integrand) and `qtrap` requires 19 calls (8192 times as many evaluations of the integrand).

**CITED REFERENCES AND FURTHER READING:**

- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §3.4 – §3.5.
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §7.4.1 – §7.4.2.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §4.10–2.

## 4.4 Improper Integrals

For our present purposes, an integral will be “improper” if it has any of the following problems:

- its integrand goes to a finite limiting value at finite upper and lower limits, but cannot be evaluated *right on* one of those limits (e.g.,  $\sin x/x$  at  $x = 0$ )
- its upper limit is  $\infty$ , or its lower limit is  $-\infty$
- it has an integrable singularity at either limit (e.g.,  $x^{-1/2}$  at  $x = 0$ )
- it has an integrable singularity at a known place between its upper and lower limits
- it has an integrable singularity at an unknown place between its upper and lower limits

If an integral is infinite (e.g.,  $\int_1^\infty x^{-1} dx$ ), or does not exist in a limiting sense (e.g.,  $\int_{-\infty}^\infty \cos x dx$ ), we do not call it improper; we call it impossible. No amount of clever algorithmics will return a meaningful answer to an ill-posed problem.

In this section we will generalize the techniques of the preceding two sections to cover the first four problems on the above list. A more advanced discussion of quadrature with integrable singularities occurs in Chapter 19, notably §19.3. The fifth problem, singularity at an unknown location, can really only be handled by the use of a variable stepsize differential equation integration routine, as will be given in Chapter 17, or an adaptive quadrature routine such as in §4.7.

We need a workhorse like the extended trapezoidal rule (equation 4.1.11), but one that is an *open* formula in the sense of §4.1, i.e., does not require the integrand to be evaluated at the endpoints. Equation (4.1.19), the extended midpoint rule, is the best choice. The reason is that (4.1.19) shares with (4.1.11) the “deep” property of having an error series that is entirely even in  $h$ . Indeed there is a formula, not as well known as it ought to be, called the *Second Euler-Maclaurin summation formula*,

$$\begin{aligned} \int_{x_0}^{x_N-1} f(x) dx &= h[f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-5/2} + f_{N-3/2}] \\ &\quad + \frac{B_2 h^2}{4}(f'_{N-1} - f'_0) + \cdots \\ &\quad + \frac{B_{2k} h^{2k}}{(2k)!}(1 - 2^{-2k+1})(f_{N-1}^{(2k-1)} - f_0^{(2k-1)}) + \cdots \end{aligned} \tag{4.4.1}$$

This equation can be derived by writing out (4.2.1) with stepsize  $h$ , then writing it out again with stepsize  $h/2$ , and then subtracting the first from twice the second.

It is not possible to double the number of steps in the extended midpoint rule and still have the benefit of previous function evaluations (try it!). However, it is possible to *triple* the number of steps and do so. Shall we do this, or double and accept the loss? On the average, tripling does a factor  $\sqrt{3}$  of unnecessary work, since the “right” number of steps for a desired accuracy criterion may in fact fall anywhere in the logarithmic interval implied by tripling. For doubling, the factor is only  $\sqrt{2}$ , but we lose an extra factor of 2 in being unable to use all the previous evaluations. Since  $1.732 < 2 \times 1.414$ , it is better to triple.

Here is the resulting structure, which is directly comparable to Trapzd.

```
quadrature.h
template <class T>
struct Midpnt : Quadrature {
    Routine implementing the extended midpoint rule.
    Doub a,b,s;                                Limits of integration and current value of integral.
    T &funk;
    Midpnt(T &funcc, const Doub aa, const Doub bb) :
        funk(funcc), a(aa), b(bb) {n=0;}
    The constructor takes as inputs func, the function or functor to be integrated between
    limits a and b, also input.
    Doub next() {
        Returns the nth stage of refinement of the extended midpoint rule. On the first call (n=1),
        the routine returns the crudest estimate of  $\int_a^b f(x)dx$ . Subsequent calls set n=2,3,... and
        improve the accuracy by adding  $(2/3) \times 3^{n-1}$  additional interior points.
        Int it,j;
        Doub x,tnm,sum,del,ddel;
        n++;
        if (n == 1) {
            return (s=(b-a)*func(0.5*(a+b)));
        } else {
            for(it=1,j=1;j<n-1;j++) it *= 3;
            tnm=it;
            del=(b-a)/(3.0*tnm);
            ddel=del+del;
            x=a+0.5*del;
            sum=0.0;
            for (j=0;j<it;j++) {
                sum += func(x);
                x += ddel;
                sum += func(x);
                x += del;
            }
            s=(s+(b-a)*sum/tnm)/3.0;      The new sum is combined with the old integral
            return s;                      to give a refined integral.
        }
    }
    virtual Doub func(const Doub x) {return funk(x);}  Identity mapping.
};
```

You may have spotted a seemingly unnecessary extra level of indirection in Midpnt, namely its calling the user-supplied function funk through an identity function func. The reason for this is that we are going to use mappings other than the identity mapping between funk and func to solve the problems of improper integrals listed above. The new quadratures will simply be derived from Midpnt with func overridden.

The structure Midpnt could be used to exactly replace Trapzd in a driver routine like qtrap (§4.2); one could simply change Trapzd<T> t(func,a,b) to Midpnt<T> t(func,a,b), and perhaps also decrease the parameter JMAX since

$3^{JMAX-1}$  (from step tripling) is a much larger number than  $2^{JMAX-1}$  (step doubling). The open formula implementation analogous to Simpson's rule (`qsimp` in §4.2) could also substitute `Midpnt` for `Trapzd`, decreasing `JMAX` as above, but now also changing the extrapolation step to be

```
s=(9.0*st-ost)/8.0;
```

since, when the number of steps is tripled, the error decreases to 1/9th its size, not 1/4th as with step doubling.

Either the thus modified `qtrap` or `qsimp` will fix the first problem on the list at the beginning of this section. More sophisticated, and allowing us to fix more problems, is to generalize Romberg integration in like manner:

```
template<class T>
Doub qromo(Midpnt<T> &q, const Doub eps=3.0e-9) {
    Romberg integration on an open interval. Returns the integral of a function using any specified
    elementary quadrature algorithm q and Romberg's method. Normally q will be an open formula,
    not evaluating the function at the endpoints. It is assumed that q triples the number of steps
    on each call, and that its error series contains only even powers of the number of steps. The
    routines midpnt, midinf, midsql, midsqu, midexp are possible choices for q. The constants
    below have the same meanings as in qromb.
    const Int JMAX=14, JMAXP=JMAX+1, K=5;
    VecDoub h(JMAXP),s(JMAX);
    Poly_interp polint(h,s,K);
    h[0]=1.0;
    for (Int j=1;j<=JMAX;j++) {
        s[j-1]=q.next();
        if (j >= K) {
            Doub ss=polint.rawinterp(j-K,0.0);
            if (abs(polint.dy) <= eps*abs(ss)) return ss;
        }
        h[j]=h[j-1]/9.0;           This is where the assumption of step tripling and an even
                                    error series is used.
    throw("Too many steps in routine qromo");
}
```

romberg.h

Notice that we now pass a `Midpnt` object instead of the user function and limits of integration. There is a good reason for this, as we will see below. It does, however, mean that you have to bind things together before calling `qromo`, something like this, where we integrate from `a` to `b`:

```
Midpnt<Ftor> q(ftor,a,b);
Doub integral=qromo(q);
```

or, for a bare function,

```
Midpnt<Doub(Doub)> q(fbare,a,b);
Doub integral=qromo(q);
```

Laid back C++ compilers will let you condense these to

```
Doub integral = qromo(Midpnt<Ftor>(Ftor(),a,b));
```

or

```
Doub integral = qromo(Midpnt<Doub(Doub)>(fbare,a,b));
```

but uptight compilers may object to the way that a temporary is passed by reference, in which case use the two-line forms above.

As we shall now see, the function `qromo`, with its peculiar interface, is an excellent driver routine for solving all the other problems of improper integrals in our first list (except the intractable fifth).

The basic trick for improper integrals is to make a change of variables to eliminate the singularity or to map an infinite range of integration to a finite one. For example, the identity

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt \quad ab > 0 \quad (4.4.2)$$

can be used with either  $b \rightarrow \infty$  and  $a$  positive, or with  $a \rightarrow -\infty$  and  $b$  negative, and works for any function that decreases toward infinity faster than  $1/x^2$ .

You can make the change of variable implied by (4.4.2) either analytically and then use, e.g., `qromo` and `Midpnt` to do the numerical evaluation, or you can let the numerical algorithm make the change of variable for you. We prefer the latter method as being more transparent to the user. To implement equation (4.4.2) we simply write a modified version of `Midpnt`, called `Midinf`, which allows  $b$  to be infinite (or, more precisely, a very large number on your particular machine, such as  $1 \times 10^{99}$ ), or  $a$  to be negative and infinite. Since all the machinery is already in place in `Midpnt`, we write `Midinf` as a derived class and simply override the mapping function.

`quadrature.h`

```
template <class T>
struct Midinf : Midpnt<T>{
    This routine is an exact replacement for midpnt, i.e., returns the nth stage of refinement of the
    integral of func from aa to bb, except that the function is evaluated at evenly spaced points in
    1/x rather than in x. This allows the upper limit bb to be as large and positive as the computer
    allows, or the lower limit aa to be as large and negative, but not both. aa and bb must have
    the same sign.
    Doub func(const Doub x) {
        return Midpnt<T>::funk(1.0/x)/(x*x);      Effect the change of variable.
    }
    Midinf(T &funcc, const Doub aa, const Doub bb) :
        Midpnt<T>(funcc, aa, bb) {
        Midpnt<T>::a=1.0/bb;                      Set the limits of integration.
        Midpnt<T>::b=1.0/aa;
    }
};
```

An integral from 2 to  $\infty$ , for example, might be calculated by

```
Midinf<Ftor> q(ftor,2.,1.e99);
Doub integral=qromo(q);
```

If you need to integrate from a negative lower limit to positive infinity, you do this by breaking the integral into two pieces at some positive value, for example,

```
Midpnt<Ftor> q1(ftor,-5.,2.);
Midinf<Ftor> q2(ftor,2.,1.e99);
integral=qromo(q1)+qromo(q2);
```

Where should you choose the breakpoint? At a sufficiently large positive value so that the function `funk` is at least beginning to approach its asymptotic decrease to zero value at infinity. The polynomial extrapolation implicit in the second call to `qromo` deals with a polynomial in  $1/x$ , not in  $x$ .

To deal with an integral that has an integrable power-law singularity at its lower limit, one also makes a change of variable. If the integrand diverges as  $(x - a)^{-\gamma}$ ,  $0 \leq \gamma < 1$ , near  $x = a$ , use the identity

$$\int_a^b f(x)dx = \frac{1}{1-\gamma} \int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(t^{\frac{1}{1-\gamma}} + a)dt \quad (b > a) \quad (4.4.3)$$

If the singularity is at the upper limit, use the identity

$$\int_a^b f(x)dx = \frac{1}{1-\gamma} \int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(b - t^{\frac{1}{1-\gamma}})dt \quad (b > a) \quad (4.4.4)$$

If there is a singularity at both limits, divide the integral at an interior breakpoint as in the example above.

Equations (4.4.3) and (4.4.4) are particularly simple in the case of inverse square-root singularities, a case that occurs frequently in practice:

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(a + t^2)dt \quad (b > a) \quad (4.4.5)$$

for a singularity at  $a$ , and

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(b - t^2)dt \quad (b > a) \quad (4.4.6)$$

for a singularity at  $b$ . Once again, we can implement these changes of variable transparently to the user by defining substitute routines for `Midpnt` that make the change of variable automatically:

```
template <class T>
struct Midsq : Midpnt<T>{
    This routine is an exact replacement for midpnt, except that it allows for an inverse square-root
    singularity in the integrand at the lower limit aa.
    Doub aorig;
    Doub func(const Doub x) {
        return 2.0*x*Midpnt<T>::funk(aorig+x*x);      Effect the change of variable.
    }
    Midsq(T &funcc, const Doub aa, const Doub bb) :
        Midpnt<T>(funcc, aa, bb), aorig(aa) {
        Midpnt<T>::a=0;
        Midpnt<T>::b=sqrt(bb-aa);
    }
};
```

`quadrature.h`

Similarly,

```
template <class T>
struct Midsq : Midpnt<T>{
    This routine is an exact replacement for midpnt, except that it allows for an inverse square-root
    singularity in the integrand at the upper limit bb.
    Doub borig;
    Doub func(const Doub x) {
        return 2.0*x*Midpnt<T>::funk(borig-x*x);      Effect the change of variable.
    }
    Midsq(T &funcc, const Doub aa, const Doub bb) :
        Midpnt<T>(funcc, aa, bb), borig(bb) {
        Midpnt<T>::a=0;
        Midpnt<T>::b=sqrt(bb-aa);
    }
};
```

`quadrature.h`

One last example should suffice to show how these formulas are derived in general. Suppose the upper limit of integration is infinite and the integrand falls off exponentially. Then we want a change of variable that maps  $e^{-x}dx$  into  $(\pm)dt$  (with the sign chosen to keep the upper limit of the new variable larger than the lower limit). Doing the integration gives by inspection

$$t = e^{-x} \quad \text{or} \quad x = -\log t \quad (4.4.7)$$

so that

$$\int_{x=a}^{x=\infty} f(x)dx = \int_{t=0}^{t=e^{-a}} f(-\log t) \frac{dt}{t} \quad (4.4.8)$$

The user-transparent implementation would be

quadrature.h

```
template <class T>
struct Midexp : Midpnt<T>{
    Doub func(const Doub x) {
        return Midpnt<T>::funk(-log(x))/x;           Effect the change of variable.
    }
    Midexp(T &funcc, const Doub aa, const Doub bb) :
        Midpnt<T>(funcc, aa, bb) {
        Midpnt<T>::a=0.0;
        Midpnt<T>::b=exp(-aa);
    }
};
```

#### CITED REFERENCES AND FURTHER READING:

- Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), Chapter 4.
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §7.4.3, p. 294.
- Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §3.7.

## 4.5 Quadrature by Variable Transformation

Imagine a simple general quadrature algorithm that is very rapidly convergent and allows you to ignore endpoint singularities completely. Sound too good to be true? In this section we'll describe an algorithm that in fact handles large classes of integrals in exactly this way.

Consider evaluating the integral

$$I = \int_a^b f(x)dx \quad (4.5.1)$$

As we saw in the construction of equations (4.1.11) – (4.1.20), quadrature formulas of arbitrarily high order can be constructed with interior weights unity, just by tuning the weights near the endpoints. But if a function dies off rapidly enough near

the endpoints, then those weights don't matter at all. In such a case, an  $N$ -point quadrature with uniform weights converges exponentially with  $N$ . (For a more rigorous motivation of this idea, see §4.5.1. For the connection to Gaussian quadrature, see the discussion at the end of §20.7.4.)

What about a function that doesn't vanish at the endpoints? Consider a change of variables  $x = x(t)$ , such that  $x \in [a, b] \rightarrow t \in [c, d]$ :

$$I = \int_c^d f[x(t)] \frac{dx}{dt} dt \quad (4.5.2)$$

Choose the transformation such that the factor  $dx/dt$  goes rapidly to zero at the endpoints of the interval. Then the simple trapezoidal rule applied to (4.5.2) will give extremely accurate results. (In this section, we'll call quadrature with uniform weights trapezoidal quadrature, with the understanding that it's a matter of taste whether you weight the endpoints with weight  $1/2$  or  $1$ , since they don't count anyway.)

Even when  $f(x)$  has integrable singularities at the endpoints of the interval, their effect can be overwhelmed by a suitable transformation  $x = x(t)$ . One need not tailor the transformation to the specific nature of the singularity: We will discuss several transformations that are effective at obliterating just about any kind of endpoint singularity.

The first transformation of this kind was introduced by Schwartz [1] and has become known as the TANH rule:

$$\begin{aligned} x &= \frac{1}{2}(b+a) + \frac{1}{2}(b-a) \tanh t, \quad x \in [a, b] \rightarrow t \in [-\infty, \infty] \\ \frac{dx}{dt} &= \frac{1}{2}(b-a) \operatorname{sech}^2 t = \frac{2}{b-a}(b-x)(x-a) \end{aligned} \quad (4.5.3)$$

The sharp decrease of  $\operatorname{sech}^2 t$  as  $t \rightarrow \pm\infty$  explains the efficiency of the algorithm and its ability to deal with singularities. Another similar algorithm is the IMT rule [2]. However,  $x(t)$  for the IMT rule is not given by a simple analytic expression, and its performance is not too different from the TANH rule.

There are two kinds of errors to consider when using something like the TANH rule. The *discretization error* is just the truncation error because you are using the trapezoidal rule to approximate  $I$ . The *trimming error* is the result of truncating the infinite sum in the trapezoidal rule at a finite value of  $N$ . (Recall that the limits are now  $\pm\infty$ .) You might think that the sharper the decrease of  $dx/dt$  as  $t \rightarrow \pm\infty$ , the more efficient the algorithm. But if the decrease is too sharp, then the density of quadrature points near the center of the original interval  $[a, b]$  is low and the discretization error is large. The optimal strategy is to try to arrange that the discretization and trimming errors are approximately equal.

For the TANH rule, Schwartz [1] showed that the discretization error is of order

$$\epsilon_d \sim e^{-2\pi w/h} \quad (4.5.4)$$

where  $w$  is the distance from the real axis to the nearest singularity of the integrand. There is a pole when  $\operatorname{sech}^2 t \rightarrow \infty$ , i.e., when  $t = \pm i\pi/2$ . If there are no poles closer to the real axis in  $f(x)$ , then  $w = \pi/2$ . The trimming error, on the other hand, is

$$\epsilon_t \sim \operatorname{sech}^2 t_N \sim e^{-2Nh} \quad (4.5.5)$$

Setting  $\epsilon_d \sim \epsilon_t$ , we find

$$h \sim \frac{\pi}{(2N)^{1/2}}, \quad \epsilon \sim e^{-\pi(2N)^{1/2}} \quad (4.5.6)$$

as the optimum  $h$  and the corresponding error. Note that  $\epsilon$  decreases with  $N$  faster than any power of  $N$ . If  $f$  is singular at the endpoints, this can modify equation (4.5.5) for  $\epsilon_t$ . This usually results in the constant  $\pi$  in (4.5.6) being reduced. Rather than developing an algorithm where we try to estimate the optimal  $h$  for each integrand a priori, we recommend simple step doubling and testing for convergence. We expect convergence to set in for  $h$  around the value given by equation (4.5.6).

The TANH rule essentially uses an exponential mapping to achieve the desired rapid fall-off at infinity. On the theory that more is better, one can try repeating the procedure. This leads to the DE (double exponential) rule:

$$\begin{aligned} x &= \frac{1}{2}(b+a) + \frac{1}{2}(b-a) \tanh(c \sinh t), \quad x \in [a, b] \rightarrow t \in [-\infty, \infty] \\ \frac{dx}{dt} &= \frac{1}{2}(b-a) \operatorname{sech}^2(c \sinh t) c \cosh t \sim \exp(-c \exp |t|) \quad \text{as } |t| \rightarrow \infty \end{aligned} \quad (4.5.7)$$

Here the constant  $c$  is usually taken to be 1 or  $\pi/2$ . (Values larger than  $\pi/2$  are not useful since  $w = \pi/2$  for  $0 < c \leq \pi/2$ , but  $w$  decreases rapidly for larger  $c$ .) By an analysis similar to equations (4.5.4) – (4.5.6), one can show that the optimal  $h$  and corresponding error for the DE rule are of order

$$h \sim \frac{\log(2\pi N w/c)}{N}, \quad \epsilon \sim e^{-kN/\log N} \quad (4.5.8)$$

where  $k$  is a constant. The improved performance of the DE rule over the TANH rule indicated by comparing equations (4.5.6) and (4.5.8) is borne out in practice.

### 4.5.1 Exponential Convergence of the Trapezoidal Rule

The error in evaluating the integral (4.5.1) by the trapezoidal rule is given by the Euler-Maclaurin summation formula,

$$I \approx \frac{h}{2}[f(a) + f(b)] + h \sum_{j=1}^{N-1} f(a+jh) - \sum_{k=1}^{\infty} \frac{B_{2k} h^{2k}}{(2k)!} [f^{(2k-1)}(b) - f^{(2k-1)}(a)] \quad (4.5.9)$$

Note that this is in general an asymptotic expansion, not a convergent series. If all the derivatives of the function  $f$  vanish at the endpoints, then all the “correction terms” in equation (4.5.9) are zero. The error in this case is very small — it goes to zero with  $h$  faster than any power of  $h$ . We say that the method converges *exponentially*. The straight trapezoidal rule is thus an excellent method for integrating functions such as  $\exp(-x^2)$  on  $(-\infty, \infty)$ , whose derivatives all vanish at the endpoints.

The class of transformations that will produce exponential convergence for a function whose derivatives do not all vanish at the endpoints is those for which  $dx/dt$  and all its derivatives go to zero at the endpoints of the interval. For functions with singularities at the endpoints, we require that  $f(x) dx/dt$  and all its derivatives vanish at the endpoints. This is a more precise statement of “ $dx/dt$  goes rapidly to zero” given above.

### 4.5.2 Implementation

Implementing the DE rule is a little tricky. It's not a good idea to simply use `Trapzd` on the function  $f(x) dx/dt$ . First, the factor  $\operatorname{sech}^2(c \sinh t)$  in equation (4.5.7) can overflow if  $\operatorname{sech}$  is computed as  $1/\cosh$ . We follow [3] and avoid this by using the variable  $q$  defined by

$$q = e^{-2 \sinh t} \quad (4.5.10)$$

(we take  $c = 1$  for simplicity) so that

$$\frac{dx}{dt} = 2(b-a) \frac{q}{(1+q)^2} \cosh t \quad (4.5.11)$$

For large positive  $t$ ,  $q$  just underflows harmlessly to zero. Negative  $t$  is handled by using the symmetry of the trapezoidal rule about the midpoint of the interval. We write

$$\begin{aligned} I &\simeq h \sum_{j=-N}^N f(x_j) \left. \frac{dx}{dt} \right|_j \\ &= h \left\{ f[(a+b)/2] \left. \frac{dx}{dt} \right|_0 + \sum_{j=1}^N [f(a+\delta_j) + f(b-\delta_j)] \left. \frac{dx}{dt} \right|_j \right\} \end{aligned} \quad (4.5.12)$$

where

$$\delta = b - x = (b-a) \frac{q}{1+q} \quad (4.5.13)$$

A second possible problem is that cancellation errors in computing  $a+\delta$  or  $b-\delta$  can cause the computed value of  $f(x)$  to blow up near the endpoint singularities. To handle this, you should code the function  $f(x)$  as a function of two arguments,  $f(x, \delta)$ . Then compute the singular part using  $\delta$  directly. For example, code the function  $x^{-\alpha}(1-x)^{-\beta}$  as  $\delta^{-\alpha}(1-x)^{-\beta}$  near  $x = 0$  and  $x^{-\alpha}\delta^{-\beta}$  near  $x = 1$ . (See §6.10 for another example of a  $f(x, \delta)$ .) Accordingly, the routine `DErule` below expects the function  $f$  to have two arguments. If your function has no singularities, or the singularities are “mild” (e.g., no worse than logarithmic), you can ignore  $\delta$  when coding  $f(x, \delta)$  and code it as if it were just  $f(x)$ .

The routine `DErule` implements equation (4.5.12). It contains an argument  $h_{\max}$  that corresponds to the upper limit for  $t$ . The first approximation to  $I$  is given by the first term on the right-hand side of (4.5.12) with  $h = h_{\max}$ . Subsequent refinements correspond to halving  $h$  as usual. We typically take  $h_{\max} = 3.7$  in double precision, corresponding to  $q = 3 \times 10^{-18}$ . This is generally adequate for “mild” singularities, like logarithms. If you want high accuracy for stronger singularities, you may have to increase  $h_{\max}$ . For example, for  $1/\sqrt{x}$  you need  $h_{\max} = 4.3$  to get full double precision. This corresponds to  $q = 10^{-32} = (10^{-16})^2$ , as you might expect.

```
template<class T>
struct DErule : Quadrature {
    Structure for implementing the DE rule.
    Doub a,b,hmax,s;
    T &func;
```

[derule.h](#)

```
DErule(T &funcc, const Doub aa, const Doub bb, const Doub hmaxx=3.7)
    : func(funcc), a(aa), b(bb), hmax(hmaxx) {n=0;}
```

Constructor. `funcc` is the function or functor that provides the function to be integrated between limits `aa` and `bb`, also input. The function operator in `funcc` takes two arguments,  $x$  and  $\delta$ , as described in the text. The range of integration in the transformed variable  $t$  is  $(-\text{hmaxx}, \text{hmaxx})$ . Typical values of `hmaxx` are 3.7 for logarithmic or milder singularities, and 4.3 for square-root singularities, as discussed in the text.

```
Doub next() {
    On the first call to the function next ( $n = 1$ ), the routine returns the crudest estimate of
     $\int_a^b f(x)dx$ . Subsequent calls to next ( $n = 2, 3, \dots$ ) will improve the accuracy by adding
     $2^{n-1}$  additional interior points.
    Doub del,fact,q,sum,t,twoh;
    Int it,j;
    n++;
    if (n == 1) {
        fact=0.25;
        return s=hmax*2.0*(b-a)*fact*func(0.5*(b+a),0.5*(b-a));
    } else {
        for (it=1,j=1;j<n-1;j++) it <= 1;
        twoh=hmax/it;                         Twice the spacing of the points to be added.
        t=0.5*twoh;
        for (sum=0.0,j=0;j<it;j++) {
            q=exp(-2.0*sinh(t));
            del=(b-a)*q/(1.0+q);
            fact=q/SQR(1.0+q)*cosh(t);
            sum += fact*(func(a+del,del)+func(b-del,del));
            t += twoh;
        }
        return s=0.5*s+(b-a)*twoh*sum; Replace s by its refined value and return.
    }
}
};
```

If the double exponential rule (DE rule) is generally better than the single exponential rule (TANH rule), why don't we keep going and use a triple exponential rule, quadruple exponential rule, ...? As we mentioned earlier, the discretization error is dominated by the pole nearest to the real axis. It turns out that beyond the double exponential the poles come nearer and nearer to the real axis, so the methods tend to get worse, not better.

If the function to be integrated itself has a pole near the real axis (much nearer than the  $\pi/2$  that comes from the DE or TANH rules), the convergence of the method slows down. In analytically tractable cases, one can find a “pole correction term” to add to the trapezoidal rule to restore rapid convergence [4].

### 4.5.3 Infinite Ranges

Simple variations of the TANH or DE rules can be used if either or both of the limits of integration is infinite:

| Range               | TANH Rule     | DE Rule                | Mixed Rule         |          |
|---------------------|---------------|------------------------|--------------------|----------|
| $(0, \infty)$       | $x = e^t$     | $x = e^{2c \sinh t}$   | $x = e^t - e^{-t}$ | (4.5.14) |
| $(-\infty, \infty)$ | $x = \sinh t$ | $x = \sinh(c \sinh t)$ | —                  |          |

The last column gives a mixed rule for functions that fall off rapidly ( $e^{-x}$  or  $e^{-x^2}$ ) at infinity. It is a DE rule at  $x = 0$  but only a single exponential at infinity. The expo-

nential fall-off of the integrand makes it behave like a DE rule there too. The mixed rule for  $(-\infty, \infty)$  is constructed by splitting the range into  $(-\infty, 0)$  and  $(0, \infty)$  and making the substitution  $x \rightarrow -x$  in the first range. This gives two integrals on  $(0, \infty)$ .

To implement the DE rule for infinite ranges we don't need the precautions we used in coding the finite range DE rule. It's fine to simply use the routine `Trapzd` directly as a function of  $t$ , with the function `func` that it calls returning  $f(x) dx/dt$ . So if `funk` is your function returning  $f(x)$ , then you define the function `func` as a function of `t` by code of the following form (for the mixed rule)

```
x=exp(t-exp(-t));
dxdt=x*(1.0+exp(-t));
return funk(x)*dxdt;
```

and pass `func` to `Trapzd`. The only care required is in deciding the range of integration. You want the contribution to the integral from the endpoints of the integration to be negligible. For example,  $(-4, 4)$  is typically adequate for  $x = \exp(\pi \sinh t)$ .

#### 4.5.4 Examples

As examples of the power of these methods, consider the following integrals:

$$\int_0^1 \log x \log(1-x) dx = 2 - \frac{\pi^2}{6} \quad (4.5.15)$$

$$\int_0^\infty \frac{1}{x^{1/2}(1+x)} dx = \pi \quad (4.5.16)$$

$$\int_0^\infty x^{-3/2} \sin \frac{x}{2} e^{-x} dx = [\pi(\sqrt{5}-2)]^{1/2} \quad (4.5.17)$$

$$\int_0^\infty x^{-2/7} e^{-x^2} dx = \frac{1}{2} \Gamma\left(\frac{5}{14}\right) \quad (4.5.18)$$

The integral (4.5.15) is easily handled by `DErule`. The routine converges to machine precision ( $10^{-16}$ ) with about 30 function evaluations, completely unfazed by the singularities at the endpoints. The integral (4.5.16) is an example of an integrand that is singular at the origin and falls off slowly at infinity. The routine `Midinf` fails miserably because of the slow fall-off. Yet the transformation  $x = \exp(\pi \sinh t)$  again gives machine precision in about 30 function evaluations, integrating  $t$  over the range  $(-4, 4)$ . By comparison, the transformation  $x = e^t$  for  $t$  in the range  $(-90, 90)$  requires about 500 function evaluations for the same accuracy.

The integral (4.5.17) combines a singularity at the origin with exponential fall-off at infinity. Here the “mixed” transformation  $x = \exp(t - e^{-t})$  is best, requiring about 60 function evaluations for  $t$  in the range  $(-4.5, 4)$ . Note that the exponential fall-off is crucial here; these transformations fail completely for slowly decaying oscillatory functions like  $x^{-3/2} \sin x$ . Fortunately the series acceleration algorithms of §5.3 work well in such cases.

The final integral (4.5.18) is similar to (4.5.17), and using the same transformation requires about the same number of function evaluations to achieve machine precision. The range of  $t$  can be smaller, say  $(-4, 3)$ , because of the more rapid fall-off of the integrand. Note that for all these integrals the number of function evaluations would be double the number we quote if we are using step doubling to

decide when the integrals have converged, since we need one extra set of trapezoidal evaluations to confirm convergence. In many cases, however, you don't need this extra set of function evaluations: Once the method starts converging, the number of significant digits approximately doubles with each iteration. Accordingly, you can set the convergence criterion to stop the procedure when two successive iterations agree to the *square root* of the desired precision. The last iteration will then have approximately the required precision. Even without this trick, the method is quite remarkable for the range of difficult integrals that it can tame efficiently.

An extended example of the use of the DE rule for finite and infinite ranges is given in §6.10. There we give a routine for computing the generalized Fermi-Dirac integrals

$$F_k(\eta, \theta) = \int_0^\infty \frac{x^k (1 + \frac{1}{2} \theta x)^{1/2}}{e^{x-\eta} + 1} dx \quad (4.5.19)$$

Another example is given in the routine `Stiel` in §4.6.

### 4.5.5 Relation to the Sampling Theorem

The *sinc expansion* of a function is

$$f(x) \simeq \sum_{k=-\infty}^{\infty} f(kh) \operatorname{sinc}\left[\frac{\pi}{h}(x - kh)\right] \quad (4.5.20)$$

where  $\operatorname{sinc}(x) \equiv \sin x / x$ . The expansion is exact for a limited class of analytic functions. However, it can be a good approximation for other functions too, and the sampling theorem characterizes these functions, as will be discussed in §13.11. There we will use the sinc expansion of  $e^{-x^2}$  to get an approximation for the complex error function. Functions well-approximated by the sinc expansion typically fall off rapidly as  $x \rightarrow \pm\infty$ , so truncating the expansion at  $k = \pm N$  still gives a good approximation to  $f(x)$ .

If we integrate both sides of equation (4.5.20), we find

$$\int_{-\infty}^{\infty} f(x) dx \simeq h \sum_{k=-\infty}^{\infty} f(kh) \quad (4.5.21)$$

which is just the trapezoidal formula! Thus, rapid convergence of the trapezoidal formula for the integral of  $f$  corresponds to  $f$  being well-approximated by its sinc expansion. The various transformations described earlier can be used to map  $x \rightarrow x(t)$  and produce good sinc approximations with uniform samples in  $t$ . These approximations can be used not only for the trapezoidal quadrature of  $f$ , but also for good approximations to derivatives, integral transforms, Cauchy principal value integrals, and solving differential and integral equations [5].

#### CITED REFERENCES AND FURTHER READING:

- Schwartz, C. 1969, "Numerical Integration of Analytic Functions," *Journal of Computational Physics*, vol. 4, pp. 19–29.[1]
- Iri, M., Moriguti, S., and Takasawa, Y. 1987, "On a Certain Quadrature Formula," *Journal of Computational and Applied Mathematics*, vol. 17, pp. 3–20. (English version of Japanese article originally published in 1970.)[2]

- Evans, G.A., Forbes, R.C., and Hyslop, J. 1984, “The Tanh Transformation for Singular Integrals,” *International Journal of Computer Mathematics*, vol. 15, pp. 339–358.[3]
- Bialecki, B. 1989, *BIT*, “A Modified Sinc Quadrature Rule for Functions with Poles near the Arc of Integration,” vol. 29, pp. 464–476.[4]
- Stenger, F. 1981, “Numerical Methods Based on Whittaker Cardinal or Sinc Functions,” *SIAM Review*, vol. 23, pp. 165–224.[5]
- Takahasi, H., and Mori, H. 1973, “Quadrature Formulas Obtained by Variable Transformation,” *Numerische Mathematik*, vol. 21, pp. 206–219.
- Mori, M. 1985, “Quadrature Formulas Obtained by Variable Transformation and DE Rule,” *Journal of Computational and Applied Mathematics*, vol. 12&13, pp. 119–130.
- Sikorski, K., and Stenger, F. 1984, “Optimal Quadratures in  $H_p$  Spaces,” *ACM Transactions on Mathematical Software*, vol. 10, pp. 140–151; *op. cit.*, pp. 152–160.

## 4.6 Gaussian Quadratures and Orthogonal Polynomials

In the formulas of §4.1, the integral of a function was approximated by the sum of its functional values at a set of equally spaced points, multiplied by certain aptly chosen weighting coefficients. We saw that as we allowed ourselves more freedom in choosing the coefficients, we could achieve integration formulas of higher and higher order. The idea of *Gaussian quadratures* is to give ourselves the freedom to choose not only the weighting coefficients, but also the location of the abscissas at which the function is to be evaluated. They will no longer be equally spaced. Thus, we will have *twice* the number of degrees of freedom at our disposal; it will turn out that we can achieve Gaussian quadrature formulas whose order is, essentially, twice that of the Newton-Cotes formula with the same number of function evaluations.

Does this sound too good to be true? Well, in a sense it is. The catch is a familiar one, which cannot be overemphasized: High order is not the same as high accuracy. High order translates to high accuracy only when the integrand is very smooth, in the sense of being “well-approximated by a polynomial.”

There is, however, one additional feature of Gaussian quadrature formulas that adds to their usefulness: We can arrange the choice of weights and abscissas to make the integral exact for a class of integrands “polynomials times some known function  $W(x)$ ” rather than for the usual class of integrands “polynomials.” The function  $W(x)$  can then be chosen to remove integrable singularities from the desired integral. Given  $W(x)$ , in other words, and given an integer  $N$ , we can find a set of weights  $w_j$  and abscissas  $x_j$  such that the approximation

$$\int_a^b W(x) f(x) dx \approx \sum_{j=0}^{N-1} w_j f(x_j) \quad (4.6.1)$$

is exact if  $f(x)$  is a polynomial. For example, to do the integral

$$\int_{-1}^1 \frac{\exp(-\cos^2 x)}{\sqrt{1-x^2}} dx \quad (4.6.2)$$

(not a very natural looking integral, it must be admitted), we might well be interested in a Gaussian quadrature formula based on the choice

$$W(x) = \frac{1}{\sqrt{1-x^2}} \quad (4.6.3)$$

in the interval  $(-1, 1)$ . (This particular choice is called *Gauss-Chebyshev integration*, for reasons that will become clear shortly.)

Notice that the integration formula (4.6.1) can also be written with the weight function  $W(x)$  not overtly visible: Define  $g(x) \equiv W(x)f(x)$  and  $v_j \equiv w_j/W(x_j)$ . Then (4.6.1) becomes

$$\int_a^b g(x)dx \approx \sum_{j=0}^{N-1} v_j g(x_j) \quad (4.6.4)$$

Where did the function  $W(x)$  go? It is lurking there, ready to give high-order accuracy to integrands of the form polynomials times  $W(x)$ , and ready to *deny* high-order accuracy to integrands that are otherwise perfectly smooth and well-behaved. When you find tabulations of the weights and abscissas for a given  $W(x)$ , you have to determine carefully whether they are to be used with a formula in the form of (4.6.1), or like (4.6.4).

So far our introduction to Gaussian quadrature is pretty standard. However, there is an aspect of the method that is not as widely appreciated as it should be: For smooth integrands (after factoring out the appropriate weight function), Gaussian quadrature converges *exponentially* fast as  $N$  increases, because the order of the method, not just the density of points, increases with  $N$ . This behavior should be contrasted with the power-law behavior (e.g.,  $1/N^2$  or  $1/N^4$ ) of the Newton-Cotes based methods in which the order remains fixed (e.g., 2 or 4) even as the density of points increases. For a more rigorous discussion, see §20.7.4.

Here is an example of a quadrature routine that contains the tabulated abscissas and weights for the case  $W(x) = 1$  and  $N = 10$ . Since the weights and abscissas are, in this case, symmetric around the midpoint of the range of integration, there are actually only five distinct values of each:

`qgaus.h`

```
template <class T>
Doub qgaus(T &func, const Doub a, const Doub b)
>Returns the integral of the function or functor func between a and b, by ten-point Gauss-
Legendre integration: the function is evaluated exactly ten times at interior points in the range
of integration.
{
    Here are the abscissas and weights:
    static const Doub x[]={0.1488743389816312, 0.4333953941292472,
        0.6794095682990244, 0.8650633666889845, 0.9739065285171717};
    static const Doub w[]={0.2955242247147529, 0.2692667193099963,
        0.2190863625159821, 0.1494513491505806, 0.0666713443086881};
    Doub xm=0.5*(b+a);
    Doub xr=0.5*(b-a);
    Doub s=0;                                Will be twice the average value of the function, since the
    for (Int j=0;j<5;j++) {                  ten weights (five numbers above each used twice)
        Doub dx=xr*x[j];
        s += w[j]*(func(xm+dx)+func(xm-dx));
    }
    return s *= xr;                            Scale the answer to the range of integration.
}
```

The above routine illustrates that one can use Gaussian quadratures without necessarily understanding the theory behind them: One just locates tabulated weights and abscissas in a book (e.g., [1] or [2]). However, the theory is very pretty, and it will come in handy if you ever need to construct your own tabulation of weights and abscissas for an unusual choice of  $W(x)$ . We will therefore give, without any proofs, some useful results that will enable you to do this. Several of the results assume that  $W(x)$  does not change sign inside  $(a, b)$ , which is usually the case in practice.

The theory behind Gaussian quadratures goes back to Gauss in 1814, who used continued fractions to develop the subject. In 1826, Jacobi rederived Gauss's results by means of orthogonal polynomials. The systematic treatment of arbitrary weight functions  $W(x)$  using orthogonal polynomials is largely due to Christoffel in 1877. To introduce these orthogonal polynomials, let us fix the interval of interest to be  $(a, b)$ . We can define the “scalar product of two functions  $f$  and  $g$  over a weight function  $W$ ” as

$$\langle f | g \rangle \equiv \int_a^b W(x) f(x) g(x) dx \quad (4.6.5)$$

The scalar product is a number, not a function of  $x$ . Two functions are said to be *orthogonal* if their scalar product is zero. A function is said to be *normalized* if its scalar product with itself is unity. A set of functions that are all mutually orthogonal and also all individually normalized is called an *orthonormal* set.

We can find a set of polynomials (i) that includes exactly one polynomial of order  $j$ , called  $p_j(x)$ , for each  $j = 0, 1, 2, \dots$ , and (ii) all of which are mutually orthogonal over the specified weight function  $W(x)$ . A constructive procedure for finding such a set is the recurrence relation

$$\begin{aligned} p_{-1}(x) &\equiv 0 \\ p_0(x) &\equiv 1 \\ p_{j+1}(x) &= (x - a_j) p_j(x) - b_j p_{j-1}(x) \quad j = 0, 1, 2, \dots \end{aligned} \quad (4.6.6)$$

where

$$\begin{aligned} a_j &= \frac{\langle x p_j | p_j \rangle}{\langle p_j | p_j \rangle} \quad j = 0, 1, \dots \\ b_j &= \frac{\langle p_j | p_j \rangle}{\langle p_{j-1} | p_{j-1} \rangle} \quad j = 1, 2, \dots \end{aligned} \quad (4.6.7)$$

The coefficient  $b_0$  is arbitrary; we can take it to be zero.

The polynomials defined by (4.6.6) are *monic*, that is, the coefficient of their leading term [ $x^j$  for  $p_j(x)$ ] is unity. If we divide each  $p_j(x)$  by the constant  $\langle [p_j | p_j] \rangle^{1/2}$ , we can render the set of polynomials orthonormal. One also encounters orthogonal polynomials with various other normalizations. You can convert from a given normalization to monic polynomials if you know that the coefficient of  $x^j$  in  $p_j$  is  $\lambda_j$ , say; then the monic polynomials are obtained by dividing each  $p_j$  by  $\lambda_j$ . Note that the coefficients in the recurrence relation (4.6.6) depend on the adopted normalization.

The polynomial  $p_j(x)$  can be shown to have exactly  $j$  distinct roots in the interval  $(a, b)$ . Moreover, it can be shown that the roots of  $p_j(x)$  “interleave” the  $j - 1$  roots of  $p_{j-1}(x)$ , i.e., there is exactly one root of the former in between each two adjacent roots of the latter. This fact comes in handy if you need to find all the

roots. You can start with the one root of  $p_1(x)$  and then, in turn, bracket the roots of each higher  $j$ , pinning them down at each stage more precisely by Newton's rule or some other root-finding scheme (see Chapter 9).

Why would you ever want to find all the roots of an orthogonal polynomial  $p_j(x)$ ? Because the abscissas of the  $N$ -point Gaussian quadrature formulas (4.6.1) and (4.6.4) with weighting function  $W(x)$  in the interval  $(a, b)$  are precisely the roots of the orthogonal polynomial  $p_N(x)$  for the same interval and weighting function. This is the fundamental theorem of Gaussian quadratures, and it lets you find the abscissas for any particular case.

Once you know the abscissas  $x_0, \dots, x_{N-1}$ , you need to find the weights  $w_j$ ,  $j = 0, \dots, N - 1$ . One way to do this (not the most efficient) is to solve the set of linear equations

$$\begin{bmatrix} p_0(x_0) & \dots & p_0(x_{N-1}) \\ p_1(x_0) & \dots & p_1(x_{N-1}) \\ \vdots & & \vdots \\ p_{N-1}(x_0) & \dots & p_{N-1}(x_{N-1}) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \end{bmatrix} = \begin{bmatrix} \int_a^b W(x)p_0(x)dx \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad (4.6.8)$$

Equation (4.6.8) simply solves for those weights such that the quadrature (4.6.1) gives the correct answer for the integral of the first  $N$  orthogonal polynomials. Note that the zeros on the right-hand side of (4.6.8) appear because  $p_1(x), \dots, p_{N-1}(x)$  are all orthogonal to  $p_0(x)$ , which is a constant. It can be shown that, with those weights, the integral of the *next*  $N - 1$  polynomials is also exact, so that the quadrature is exact for all polynomials of degree  $2N - 1$  or less. Another way to evaluate the weights (though one whose proof is beyond our scope) is by the formula

$$w_j = \frac{\langle p_{N-1}|p_{N-1} \rangle}{p_{N-1}(x_j)p'_N(x_j)} \quad (4.6.9)$$

where  $p'_N(x_j)$  is the derivative of the orthogonal polynomial at its zero  $x_j$ .

The computation of Gaussian quadrature rules thus involves two distinct phases: (i) the generation of the orthogonal polynomials  $p_0, \dots, p_N$ , i.e., the computation of the coefficients  $a_j, b_j$  in (4.6.6), and (ii) the determination of the zeros of  $p_N(x)$ , and the computation of the associated weights. For the case of the “classical” orthogonal polynomials, the coefficients  $a_j$  and  $b_j$  are explicitly known (equations 4.6.10 – 4.6.14 below) and phase (i) can be omitted. However, if you are confronted with a “nonclassical” weight function  $W(x)$ , and you don’t know the coefficients  $a_j$  and  $b_j$ , the construction of the associated set of orthogonal polynomials is not trivial. We discuss it at the end of this section.

### 4.6.1 Computation of the Abscissas and Weights

This task can range from easy to difficult, depending on how much you already know about your weight function and its associated polynomials. In the case of classical, well-studied, orthogonal polynomials, practically everything is known, including good approximations for their zeros. These can be used as starting guesses, enabling Newton’s method (to be discussed in §9.4) to converge very rapidly. Newton’s method requires the derivative  $p'_N(x)$ , which is evaluated by standard relations in terms of  $p_N$  and  $p_{N-1}$ . The weights are then conveniently evaluated by equation

(4.6.9). For the following named cases, this direct root finding is faster, by a factor of 3 to 5, than any other method.

Here are the weight functions, intervals, and recurrence relations that generate the most commonly used orthogonal polynomials and their corresponding Gaussian quadrature formulas.

*Gauss-Legendre:*

$$\begin{aligned} W(x) &= 1 & -1 < x < 1 \\ (j+1)P_{j+1} &= (2j+1)xP_j - jP_{j-1} \end{aligned} \quad (4.6.10)$$

*Gauss-Chebyshev:*

$$\begin{aligned} W(x) &= (1-x^2)^{-1/2} & -1 < x < 1 \\ T_{j+1} &= 2xT_j - T_{j-1} \end{aligned} \quad (4.6.11)$$

*Gauss-Laguerre:*

$$\begin{aligned} W(x) &= x^\alpha e^{-x} & 0 < x < \infty \\ (j+1)L_{j+1}^\alpha &= (-x + 2j + \alpha + 1)L_j^\alpha - (j + \alpha)L_{j-1}^\alpha \end{aligned} \quad (4.6.12)$$

*Gauss-Hermite:*

$$\begin{aligned} W(x) &= e^{-x^2} & -\infty < x < \infty \\ H_{j+1} &= 2xH_j - 2jH_{j-1} \end{aligned} \quad (4.6.13)$$

*Gauss-Jacobi:*

$$\begin{aligned} W(x) &= (1-x)^\alpha(1+x)^\beta & -1 < x < 1 \\ c_j P_{j+1}^{(\alpha, \beta)} &= (d_j + e_j x)P_j^{(\alpha, \beta)} - f_j P_{j-1}^{(\alpha, \beta)} \end{aligned} \quad (4.6.14)$$

where the coefficients  $c_j$ ,  $d_j$ ,  $e_j$ , and  $f_j$  are given by

$$\begin{aligned} c_j &= 2(j+1)(j+\alpha+\beta+1)(2j+\alpha+\beta) \\ d_j &= (2j+\alpha+\beta+1)(\alpha^2-\beta^2) \\ e_j &= (2j+\alpha+\beta)(2j+\alpha+\beta+1)(2j+\alpha+\beta+2) \\ f_j &= 2(j+\alpha)(j+\beta)(2j+\alpha+\beta+2) \end{aligned} \quad (4.6.15)$$

We now give individual routines that calculate the abscissas and weights for these cases. First comes the most common set of abscissas and weights, those of Gauss-Legendre. The routine, due to G.B. Rybicki, uses equation (4.6.9) in the special form for the Gauss-Legendre case,

$$w_j = \frac{2}{(1-x_j^2)[P'_N(x_j)]^2} \quad (4.6.16)$$

The routine also scales the range of integration from  $(x_1, x_2)$  to  $(-1, 1)$ , and provides abscissas  $x_j$  and weights  $w_j$  for the Gaussian formula

$$\int_{x_1}^{x_2} f(x)dx = \sum_{j=0}^{N-1} w_j f(x_j) \quad (4.6.17)$$

gauss.wgts.h

```

void gauleg(const Doub x1, const Doub x2, VecDoub_0 &x, VecDoub_0 &w)
Given the lower and upper limits of integration x1 and x2, this routine returns arrays x[0..n-1]
and w[0..n-1] of length n, containing the abscissas and weights of the Gauss-Legendre n-point
quadrature formula.
{
    const Doub EPS=1.0e-14;                                EPS is the relative precision.
    Doub z1,z,xm,xl,pp,p3,p2,p1;
    Int n=x.size();
    Int m=(n+1)/2;                                         The roots are symmetric in the interval, so
    xm=0.5*(x2+x1);                                       we only have to find half of them.
    xl=0.5*(x2-x1);
    for (Int i=0;i<m;i++) {                                Loop over the desired roots.
        z=cos(3.141592654*(i+0.75)/(n+0.5));
        Starting with this approximation to the ith root, we enter the main loop of refinement
        by Newton's method.
        do {
            p1=1.0;
            p2=0.0;
            for (Int j=0;j<n;j++) {          Loop up the recurrence relation to get the
                p3=p2;                      Legendre polynomial evaluated at z.
                p2=p1;
                p1=((2.0*j+1.0)*z*p2-j*p3)/(j+1);
            }
            p1 is now the desired Legendre polynomial. We next compute pp, its derivative,
            by a standard relation involving also p2, the polynomial of one lower order.
            pp=n*(z*p1-p2)/(z*z-1.0);
            z1=z;
            z=z1-p1/pp;                     Newton's method.
            } while (abs(z-z1) > EPS);
            x[i]=xm-xl*z;
            x[n-1-i]=xm+xl*z;
            w[i]=2.0*xl/((1.0-z*z)*pp*pp);
            w[n-1-i]=w[i];
        }
}

```

Next we give three routines that use initial approximations for the roots given by Stroud and Secrest [2]. The first is for Gauss-Laguerre abscissas and weights, to be used with the integration formula

$$\int_0^\infty x^\alpha e^{-x} f(x) dx = \sum_{j=0}^{N-1} w_j f(x_j) \quad (4.6.18)$$

gauss.wgts.h

```

void gaulag(VecDoub_0 &x, VecDoub_0 &w, const Doub alf)
Given alf, the parameter  $\alpha$  of the Laguerre polynomials, this routine returns arrays x[0..n-1]
and w[0..n-1] containing the abscissas and weights of the n-point Gauss-Laguerre quadrature
formula. The smallest abscissa is returned in x[0], the largest in x[n-1].
{
```

```

    const Int MAXIT=10;
    const Doub EPS=1.0e-14;                                EPS is the relative precision.
    Int i,its,j;
    Doub ai,p1,p2,p3,pp,z,z1;
    Int n=x.size();
    for (i=0;i<n;i++) {                                Loop over the desired roots.
        if (i == 0) {                                     Initial guess for the smallest root.
            z=(1.0+alf)*(3.0+0.92*alf)/(1.0+2.4*n+1.8*alf);
        } else if (i == 1) {                               Initial guess for the second root.
            z += (15.0+6.25*alf)/(1.0+0.9*alf+2.5*n);
        } else {                                         Initial guess for the other roots.
            ai=i-1;
        }
    }
}
```

```

        z += ((1.0+2.55*ai)/(1.9*ai)+1.26*ai*alf/
              (1.0+3.5*ai))*(z-x[i-2])/(1.0+0.3*alf);
    }
    for (its=0;its<MAXIT;its++) {      Refinement by Newton's method.
        p1=1.0;
        p2=0.0;
        for (j=0;j<n;j++) {            Loop up the recurrence relation to get the
            p3=p2;                      Laguerre polynomial evaluated at z.
            p2=p1;
            p1=((2*j+1+alf-z)*p2-(j+alf)*p3)/(j+1);
        }
        p1 is now the desired Laguerre polynomial. We next compute pp, its derivative,
        by a standard relation involving also p2, the polynomial of one lower order.
        pp=(n*p1-(n+alf)*p2)/z;
        z1=z;
        z=z1-p1/pp;                  Newton's formula.
        if (abs(z-z1) <= EPS) break;
    }
    if (its >= MAXIT) throw("too many iterations in gaulag");
    x[i]=z;                      Store the root and the weight.
    w[i] = -exp(gammln(alf+n)-gammln(Doub(n)))/(pp*n*p2);
}

```

Next is a routine for Gauss-Hermite abscissas and weights. If we use the “standard” normalization of these functions, as given in equation (4.6.13), we find that the computations overflow for large  $N$  because of various factorials that occur. We can avoid this by using instead the orthonormal set of polynomials  $\tilde{H}_j$ . They are generated by the recurrence

$$\tilde{H}_{-1} = 0, \quad \tilde{H}_0 = \frac{1}{\pi^{1/4}}, \quad \tilde{H}_{j+1} = x \sqrt{\frac{2}{j+1}} \tilde{H}_j - \sqrt{\frac{j}{j+1}} \tilde{H}_{j-1} \quad (4.6.19)$$

The formula for the weights becomes

$$w_j = \frac{2}{[\tilde{H}'_N(x_j)]^2} \quad (4.6.20)$$

while the formula for the derivative with this normalization is

$$\tilde{H}'_j = \sqrt{2j} \tilde{H}_{j-1} \quad (4.6.21)$$

The abscissas and weights returned by gauher are used with the integration formula

$$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx = \sum_{j=0}^{N-1} w_j f(x_j) \quad (4.6.22)$$

```
void gauher(VecDoub_0 &x, VecDoub_0 &w)
```

gauss\_wgts.h

This routine returns arrays  $x[0..n-1]$  and  $w[0..n-1]$  containing the abscissas and weights of the  $n$ -point Gauss-Hermite quadrature formula. The largest abscissa is returned in  $x[0]$ , the most negative in  $x[n-1]$ .

```
{
    const Doub EPS=1.0e-14,PIM4=0.7511255444649425;
    Relative precision and  $1/\pi^{1/4}$ .
    const Int MAXIT=10;                         Maximum iterations.
    Int i,its,j,m;
```

```

Doub p1,p2,p3,pp,z,z1;
Int n=x.size();
m=(n+1)/2;
The roots are symmetric about the origin, so we have to find only half of them.
for (i=0;i<m;i++) {
    if (i == 0) {
        z=sqrt(Doub(2*n+1))-1.85575*pow(Doub(2*n+1),-0.16667);
    } else if (i == 1) {
        z -= 1.14*pow(Doub(n),0.426)/z;
    } else if (i == 2) {
        z=1.86*z-0.86*x[0];
    } else if (i == 3) {
        z=1.91*z-0.91*x[1];
    } else {
        z=2.0*z-x[i-2];
    }
    for (its=0;its<MAXIT;its++) {
        p1=PIM4;
        p2=0.0;
        for (j=0;j<n;j++) {
            p3=p2;
            p2=p1;
            p1=z*sqrt(2.0/(j+1))*p2-sqrt(Doub(j)/(j+1))*p3;
        }
        p1 is now the desired Hermite polynomial. We next compute pp, its derivative, by
        the relation (4.6.21) using p2, the polynomial of one lower order.
        pp=sqrt(Doub(2*n))*p2;
        z1=z;
        z=z1-p1/pp;
        if (abs(z-z1) <= EPS) break;
    }
    if (its >= MAXIT) throw("too many iterations in gauher");
    x[i]=z;
    x[n-1-i] = -z;
    w[i]=2.0/(pp*pp);
    w[n-1-i]=w[i];
}
}

```

Loop over the desired roots.  
Initial guess for the largest root.  
Initial guess for the second largest root.  
Initial guess for the third largest root.  
Initial guess for the fourth largest root.  
Initial guess for the other roots.  
Refinement by Newton's method.  
Loop up the recurrence relation to get  
the Hermite polynomial evaluated at  
z.  
Newton's formula.  
Store the root  
and its symmetric counterpart.  
Compute the weight  
and its symmetric counterpart.

Finally, here is a routine for Gauss-Jacobi abscissas and weights, which implement the integration formula

$$\int_{-1}^1 (1-x)^\alpha (1+x)^\beta f(x) dx = \sum_{j=0}^{N-1} w_j f(x_j) \quad (4.6.23)$$

gauss.wgts.h

```
void gaujac(VecDoub_0 &x, VecDoub_0 &w, const Doub alf, const Doub bet)
```

Given alf and bet, the parameters  $\alpha$  and  $\beta$  of the Jacobi polynomials, this routine returns arrays  $x[0..n-1]$  and  $w[0..n-1]$  containing the abscissas and weights of the  $n$ -point Gauss-Jacobi quadrature formula. The largest abscissa is returned in  $x[0]$ , the smallest in  $x[n-1]$ .

```
{
    const Int MAXIT=10;
    const Doub EPS=1.0e-14;           EPS is the relative precision.
    Int i,its,j;
    Doub alfbet,an,bn,r1,r2,r3;
    Doub a,b,c,p1,p2,p3,pp,temp,z,z1;
    Int n=x.size();
    for (i=0;i<n;i++) {
        if (i == 0) {
            an=alf/n;
            Loop over the desired roots.
            Initial guess for the largest root.
        }
    }
}
```

```

bn=bet/n;
r1=(1.0+alf)*(2.78/(4.0+n*n)+0.768*an/n);
r2=1.0+1.48*an+0.96*bn+0.452*an*an+0.83*an*bn;
z=1.0-r1/r2;
} else if (i == 1) {           Initial guess for the second largest root.
    r1=(4.1+alf)/((1.0+alf)*(1.0+0.156*alf));
    r2=1.0+0.06*(n-8.0)*(1.0+0.12*alf)/n;
    r3=1.0+0.012*bet*(1.0+0.25*abs(alf))/n;
    z -= (1.0-z)*r1*r2*r3;
} else if (i == 2) {           Initial guess for the third largest root.
    r1=(1.67+0.28*alf)/(1.0+0.37*alf);
    r2=1.0+0.22*(n-8.0)/n;
    r3=1.0+8.0*bet/((6.28+bet)*n*n);
    z -= (x[0]-z)*r1*r2*r3;
} else if (i == n-2) {         Initial guess for the second smallest root.
    r1=(1.0+0.235*bet)/(0.766+0.119*bet);
    r2=1.0/(1.0+0.639*(n-4.0)/(1.0+0.71*(n-4.0)));
    r3=1.0/(1.0+20.0*alf/((7.5+alf)*n*n));
    z += (z-x[n-4])*r1*r2*r3;
} else if (i == n-1) {         Initial guess for the smallest root.
    r1=(1.0+0.37*bet)/(1.67+0.28*bet);
    r2=1.0/(1.0+0.22*(n-8.0)/n);
    r3=1.0/(1.0+8.0*alf/((6.28+alf)*n*n));
    z += (z-x[n-3])*r1*r2*r3;
} else {                      Initial guess for the other roots.
    z=3.0*x[i-1]-3.0*x[i-2]+x[i-3];
}
alfbet=alf+bet;
for (its=1;its<=MAXIT;its++) {   Refinement by Newton's method.
    temp=2.0+alfbet;
    p1=(alf-bet+temp*z)/2.0;      Start the recurrence with  $P_0$  and  $P_1$  to avoid
    p2=1.0;                      a division by zero when  $\alpha + \beta = 0$  or
    for (j=2;j<=n;j++) {        Loop up the recurrence relation to get the
        p3=p2;                  Jacobi polynomial evaluated at  $z$ .
        p2=p1;
        p1=(alf-bet+temp*z)/2.0;
        temp=2*j+alfbet;
        a=2*j*(j+alfbet)*(temp-2.0);
        b=(temp-1.0)*(alf*alf-bet*bet+temp*(temp-2.0)*z);
        c=2.0*(j-1+alf)*(j-1+bet)*temp;
        p1=(b*p2-c*p3)/a;
    }
    pp=(n*(alf-bet-temp*z)*p1+2.0*(n+alf)*(n+bet)*p2)/(temp*(1.0-z*z));
    p1 is now the desired Jacobi polynomial. We next compute pp, its derivative, by
    a standard relation involving also p2, the polynomial of one lower order.
    z1=z;
    z=z1-p1/pp;                  Newton's formula.
    if (abs(z-z1) <= EPS) break;
}
if (its > MAXIT) throw("too many iterations in gaujac");
x[i]=z;                         Store the root and the weight.
w[i]=exp(gammln(alf+n)+gammln(bet+n)-gammln(n+1.0)-
    gammln(n+alfbet+1.0))*temp*pow(2.0,alfbet)/(pp*p2);
}
}

```

Legendre polynomials are special cases of Jacobi polynomials with  $\alpha = \beta = 0$ , but it is worth having the separate routine for them, `gauleg`, given above. Chebyshev polynomials correspond to  $\alpha = \beta = -1/2$  (see §5.8). They have analytic abscissas and weights:

$$\begin{aligned} x_j &= \cos\left(\frac{\pi(j + \frac{1}{2})}{N}\right) \\ w_j &= \frac{\pi}{N} \end{aligned} \quad (4.6.24)$$

### 4.6.2 Case of Known Recurrences

Turn now to the case where you do not know good initial guesses for the zeros of your orthogonal polynomials, but you do have available the coefficients  $a_j$  and  $b_j$  that generate them. As we have seen, the zeros of  $p_N(x)$  are the abscissas for the  $N$ -point Gaussian quadrature formula. The most useful computational formula for the weights is equation (4.6.9) above, since the derivative  $p'_N$  can be efficiently computed by the derivative of (4.6.6) in the general case, or by special relations for the classical polynomials. Note that (4.6.9) is valid as written only for monic polynomials; for other normalizations, there is an extra factor of  $\lambda_N/\lambda_{N-1}$ , where  $\lambda_N$  is the coefficient of  $x^N$  in  $p_N$ .

Except in those special cases already discussed, the best way to find the abscissas is *not* to use a root-finding method like Newton's method on  $p_N(x)$ . Rather, it is generally faster to use the Golub-Welsch [3] algorithm, which is based on a result of Wilf [4]. This algorithm notes that if you bring the term  $x p_j$  to the left-hand side of (4.6.6) and the term  $p_{j+1}$  to the right-hand side, the recurrence relation can be written in matrix form as

$$x \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} = \begin{bmatrix} a_0 & 1 & & & \\ b_1 & a_1 & 1 & & \\ & \vdots & \vdots & & \\ & & b_{N-2} & a_{N-2} & 1 \\ & & & b_{N-1} & a_{N-1} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ p_N \end{bmatrix} \quad (4.6.25)$$

or

$$x \mathbf{p} = \mathbf{T} \cdot \mathbf{p} + p_N \mathbf{e}_{N-1} \quad (4.6.26)$$

Here  $\mathbf{T}$  is a tridiagonal matrix;  $\mathbf{p}$  is a column vector of  $p_0, p_1, \dots, p_{N-1}$ ; and  $\mathbf{e}_{N-1}$  is a unit vector with a 1 in the  $(N - 1)$ st (last) position and zeros elsewhere. The matrix  $\mathbf{T}$  can be symmetrized by a diagonal similarity transformation  $\mathbf{D}$  to give

$$\mathbf{J} = \mathbf{D} \mathbf{T} \mathbf{D}^{-1} = \begin{bmatrix} a_0 & \sqrt{b_1} & & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & & \\ & \vdots & \vdots & & \\ & & \sqrt{b_{N-2}} & a_{N-2} & \sqrt{b_{N-1}} \\ & & & \sqrt{b_{N-1}} & a_{N-1} \end{bmatrix} \quad (4.6.27)$$

The matrix  $\mathbf{J}$  is called the *Jacobi matrix* (not to be confused with other matrices named after Jacobi that arise in completely different problems!). Now we see from (4.6.26) that  $p_N(x_j) = 0$  is equivalent to  $x_j$  being an eigenvalue of  $\mathbf{T}$ . Since eigenvalues are preserved by a similarity transformation,  $x_j$  is an eigenvalue of the symmetric tridiagonal matrix  $\mathbf{J}$ . Moreover, Wilf [4] shows that if  $\mathbf{v}_j$  is the eigenvector corresponding to the eigenvalue  $x_j$ , normalized so that  $\mathbf{v} \cdot \mathbf{v} = 1$ , then

$$w_j = \mu_0 v_{j,0}^2 \quad (4.6.28)$$

where

$$\mu_0 = \int_a^b W(x) dx \quad (4.6.29)$$

and where  $v_{j,0}$  is the zeroth component of  $\mathbf{v}$ . As we shall see in Chapter 11, finding all eigenvalues and eigenvectors of a symmetric tridiagonal matrix is a relatively efficient and well-conditioned procedure. We accordingly give a routine, `gaucof`, for finding the abscissas and weights, given the coefficients  $a_j$  and  $b_j$ . Remember that if you know the recurrence relation for orthogonal polynomials that are not normalized to be monic, you can easily convert it to monic form by means of the quantities  $\lambda_j$ .

```

void gaucof(VecDoub_Io &a, VecDoub_Io &b, const Doub amu0, VecDoub_O &x,
            VecDoub_O &w)
Computes the abscissas and weights for a Gaussian quadrature formula from the Jacobi matrix.
On input, a[0..n-1] and b[0..n-1] are the coefficients of the recurrence relation for the set of
monic orthogonal polynomials. The quantity  $\mu_0 \equiv \int_a^b W(x) dx$  is input as amu0. The abscissas
x[0..n-1] are returned in descending order, with the corresponding weights in w[0..n-1]. The
arrays a and b are modified. Execution can be speeded up by modifying tqli and eigrt to
compute only the zeroth component of each eigenvector.
{
    Int n=a.size();
    for (Int i=0;i<n;i++)
        if (i != 0) b[i]=sqrt(b[i]);           Set up superdiagonal of Jacobi matrix.
    Symmeig sym(a,b);
    for (Int i=0;i<n;i++) {
        x[i]=sym.d[i];
        w[i]=amu0*sym.z[0][i]*sym.z[0][i];      Equation (4.6.28).
    }
}

```

### 4.6.3 Orthogonal Polynomials with Nonclassical Weights

What do you do if your weight function is not one of the classical ones dealt with above and you do not know the  $a_j$ 's and  $b_j$ 's of the recurrence relation (4.6.6) to use in gaucof? Obviously, you need a method of finding the  $a_j$ 's and  $b_j$ 's.

The best general method is the *Stieltjes procedure*: First compute  $a_0$  from (4.6.7), and then  $p_1(x)$  from (4.6.6). Knowing  $p_0$  and  $p_1$ , compute  $a_1$  and  $b_1$  from (4.6.7), and so on. But how are we to compute the inner products in (4.6.7)?

The textbook approach is to represent each  $p_j(x)$  explicitly as a polynomial in  $x$  and to compute the inner products by multiplying out term by term. This will be feasible if we know the first  $2N$  moments of the weight function,

$$\mu_j = \int_a^b x^j W(x) dx \quad j = 0, 1, \dots, 2N - 1 \quad (4.6.30)$$

However, the solution of the resulting set of algebraic equations for the coefficients  $a_j$  and  $b_j$  in terms of the moments  $\mu_j$  is in general *extremely* ill-conditioned. Even in double precision, it is not unusual to lose all accuracy by the time  $N = 12$ . We thus reject any procedure based on the moments (4.6.30).

Gautschi [5] showed that the Stieltjes procedure is feasible if the inner products in (4.6.7) are computed directly by numerical quadrature. This is only practicable if you can find a quadrature scheme that can compute the integrals to high accuracy despite the singularities in the weight function  $W(x)$ . Gautschi advocates the Fejér quadrature scheme [5] as a general-purpose scheme for handling singularities when no better method is available. We have personally had much better experience with the transformation methods of §4.5, particularly the DE rule and its variants.

We use a structure **Stiel** that implements the Stieltjes procedure. Its member function `get_weights` generates the coefficients  $a_j$  and  $b_j$  of the recurrence relation, and then calls gaucof to find the abscissas and weights. You can easily modify it to return the  $a_j$ 's and  $b_j$ 's if you want them as well. Internally, the routine calls the function quad to do the integrals in (4.6.7). For a finite range of integration, the routine uses the straight DE rule. This is effected by invoking the constructor with five parameters: the number of quadrature abscissas (and weights) desired, the lower and upper limits of integration, the parameter  $h_{\max}$  to be passed to the DE rule (see §4.5), and the weight function  $W(x)$ . For an infinite range of integration, the routine invokes the trapezoidal rule with one of the coordinate transformations discussed in §4.5. For this case you invoke the constructor that has no  $h_{\max}$ , but takes the mapping function  $x = x(t)$  and its derivative  $dx/dt$  in addition to  $W(x)$ . Now the range of integration you input is the finite range of the trapezoidal rule.

This will all be clearer with some examples. Consider first the weight function

$$W(x) = -\log x \quad (4.6.31)$$

on the finite interval  $(0, 1)$ . Normally, for the finite range case (DE rule), the weight function must be coded as a function of two variables,  $W(x, \delta)$ , where  $\delta$  is the distance from the endpoint singularity. Since the logarithmic singularity at the endpoint  $x = 0$  is “mild,” there is no need to use the argument  $\delta$  in coding the function:

```
Doub wt(const Doub x, const Doub del)
{
    return -log(x);
}
```

A value of  $h_{\max} = 3.7$  will give full double precision, as discussed in §4.5, so the calling code looks like this:

```
n= ...
VecDoub x(n),w(n);
Stiel s(n,0.0,1.0,3.7,wt);
s.get_weights(x,w);
```

For the infinite range case, in addition to the weight function  $W(x)$ , you have to supply two functions for the coordinate transformation you want to use (see equation 4.5.14). We’ll denote the mapping  $x = x(t)$  by  $fx$  and  $dx/dt$  by  $fdxdt$ , but you can use any names you like. All these functions are coded as functions of one variable.

Here is an example of the user-supplied functions for the weight function

$$W(x) = \frac{x^{1/2}}{e^x + 1} \quad (4.6.32)$$

on the interval  $(0, \infty)$ . Gaussian quadrature based on  $W(x)$  has been proposed for evaluating generalized Fermi-Dirac integrals [6] (cf. §4.5). We use the “mixed” DE rule of equation (4.5.14),  $x = e^t - e^{-t}$ . As is typical with the Stieltjes procedure, you get abscissas and weights within about one or two significant digits of machine accuracy for  $N$  of a few dozen.

```
Doub wt(const Doub x)
{
    Doub s=exp(-x);
    return sqrt(x)*s/(1.0+s);
}

Doub fx(const Doub t)
{
    return exp(t-exp(-t));
}

Doub fwdxdt(const Doub t)
{
    Doub s=exp(-t);
    return exp(t-s)*(1.0+s);
}
...

Stiel ss(n,-5.5,6.5,wt,fx,fwdxdt);
ss.get_weights(x,w);
```

The listing of the `Stiel` object, and discussion of some of the C++ intricacies of its coding, are in a Webnote [9].

Two other algorithms exist [7,8] for finding abscissas and weights for Gaussian quadratures. The first starts similarly to the Stieltjes procedure by representing the inner product integrals in equation (4.6.7) as discrete quadratures using some quadrature rule. This defines a matrix whose elements are formed from the abscissas and weights in your chosen quadrature rule, together with the given weight function. Then an algorithm due to Lanczos is used to transform this to a matrix that is essentially the Jacobi matrix (4.6.27).

The second algorithm is based on the idea of *modified moments*. Instead of using powers of  $x$  as a set of basis functions to represent the  $p_j$ ’s, one uses some other known set of orthogonal polynomials  $\pi_j(x)$ , say. Then the inner products in equation (4.6.7) will be expressible

in terms of the modified moments

$$v_j = \int_a^b \pi_j(x) W(x) dx \quad j = 0, 1, \dots, 2N - 1 \quad (4.6.33)$$

The *modified Chebyshev algorithm* (due to Sack and Donovan [10] and later improved by Wheeler [11]) is an efficient algorithm that generates the desired  $a_j$ 's and  $b_j$ 's from the modified moments. Roughly speaking, the improved stability occurs because the polynomial basis “samples” the interval  $(a, b)$  better than the power basis when the inner product integrals are evaluated, especially if its weight function resembles  $W(x)$ . The algorithm requires that the modified moments (4.6.33) be accurately computed. Sometimes there is a closed form, for example, for the important case of the  $\log x$  weight function [12,8]. Otherwise you have to use a suitable discretization procedure to compute the modified moments [7,8], just as we did for the inner products in the Stieltjes procedure. There is some art in choosing the auxiliary polynomials  $\pi_j$ , and in practice it is not always possible to find a set that removes the ill-conditioning.

Gautschi [8] has given an extensive suite of routines that handle all three of the algorithms we have described, together with many other aspects of orthogonal polynomials and Gaussian quadrature. However, for most straightforward applications, you should find `Stiel` together with a suitable DE rule quadrature more than adequate.

#### 4.6.4 Extensions of Gaussian Quadrature

There are many different ways in which the ideas of Gaussian quadrature have been extended. One important extension is the case of *preassigned nodes*: Some points are required to be included in the set of abscissas, and the problem is to choose the weights and the remaining abscissas to maximize the degree of exactness of the the quadrature rule. The most common cases are *Gauss-Radau* quadrature, where one of the nodes is an endpoint of the interval, either  $a$  or  $b$ , and *Gauss-Lobatto* quadrature, where both  $a$  and  $b$  are nodes. Golub [13,8] has given an algorithm similar to `gaucof` for these cases.

An  $N$ -point Gauss-Radau rule has the form of equation (4.6.1), where  $x_1$  is chosen to be either  $a$  or  $b$  ( $x_1$  must be finite). You can construct the rule from the coefficients for the corresponding ordinary  $N$ -point Gaussian quadrature. Simply set up the Jacobi matrix equation (4.6.27), but modify the entry  $a_{N-1}$ :

$$a'_{N-1} = x_1 - b_{N-1} \frac{p_{N-2}(x_1)}{p_{N-1}(x_1)} \quad (4.6.34)$$

Here is the routine:

```
void radau(VecDoub_Io &a, VecDoub_Io &b, const Doub amu0, const Doub x1,
           VecDoub_O &x, VecDoub_O &w)                                gauss_wgts2.h
Computes the abscissas and weights for a Gauss-Radau quadrature formula. On input, a[0..n-1]
and b[0..n-1] are the coefficients of the recurrence relation for the set of monic orthogonal
polynomials corresponding to the weight function. (b[0] is not referenced.) The quantity
mu0 ≡ ∫_a^b W(x) dx is input as amu0. x1 is input as either endpoint of the interval. The abscissas
x[0..n-1] are returned in descending order, with the corresponding weights in w[0..n-1]. The
arrays a and b are modified.
{
    Int n=a.size();
    if (n == 1) {
        x[0]=x1;
        w[0]=amu0;
    } else {                                         Compute p_{N-1} and p_{N-2} by recurrence.
        Doub p=x1-a[0];
```

```

Doub pm1=1.0;
Doub p1=p;
for (Int i=1;i<n-1;i++) {
    p=(xi-a[i])*p1-b[i]*pm1;
    pm1=p1;
    p1=p;
}
a[n-1]=xi-b[n-1]*pm1/p;           Equation (4.6.34).
gaucof(a,b,amu0,x,w);
}
}

```

An  $N$ -point Gauss-Lobatto rule has the form of equation (4.6.1) where  $x_1 = a$ ,  $x_N = b$  (both finite). This time you modify the entries  $a_{N-1}$  and  $b_{N-1}$  in equation (4.6.27) by solving two linear equations:

$$\begin{bmatrix} p_{N-1}(x_1) & p_{N-2}(x_1) \\ p_{N-1}(x_N) & p_{N-2}(x_N) \end{bmatrix} \begin{bmatrix} a'_{N-1} \\ b'_{N-1} \end{bmatrix} = \begin{bmatrix} x_1 p_{N-1}(x_1) \\ x_N p_{N-1}(x_N) \end{bmatrix} \quad (4.6.35)$$

gauss\_wgts2.h

```

void lobatto(VecDoub_Io &a, VecDoub_Io &b, const Doub amu0, const Doub x1,
             const Doub xn, VecDoub_O &x, VecDoub_O &w)
Computes the abscissas and weights for a Gauss-Lobatto quadrature formula. On input, the
vectors a[0..n-1] and b[0..n-1] are the coefficients of the recurrence relation for the set of
monic orthogonal polynomials corresponding to the weight function. (b[0] is not referenced.)
The quantity  $\mu_0 \equiv \int_a^b W(x) dx$  is input as amu0. x1 and xn are input as the endpoints of
the interval. The abscissas x[0..n-1] are returned in descending order, with the corresponding
weights in w[0..n-1]. The arrays a and b are modified.
{
    Doub det,pl,pr,p1l,p1r,pm1l,pm1r;
    Int n=a.size();
    if (n <= 1)
        throw("n must be bigger than 1 in lobatto");
    pl=x1-a[0];                      Compute p_{N-1} and p_{N-2} at x_1 and x_N by recur-
    pr=xn-a[0];                      rence.
    pm1l=1.0;
    pm1r=1.0;
    p1l=pl;
    p1r=pr;
    for (Int i=1;i<n-1;i++) {
        pl=(x1-a[i])*p1l-b[i]*pm1l;
        pr=(xn-a[i])*p1r-b[i]*pm1r;
        pm1l=p1l;
        pm1r=p1r;
        p1l=pl;
        p1r=pr;
    }
    det=pl*pm1r-pr*pm1l;           Solve equation (4.6.35).
    a[n-1]=(x1*pl*pm1r-xn*pr*pm1l)/det;
    b[n-1]=(xn-x1)*pl*p1r/det;
    gaucof(a,b,amu0,x,w);
}

```

The second important extension of Gaussian quadrature is the *Gauss-Kronrod* formulas. For ordinary Gaussian quadrature formulas, as  $N$  increases, the sets of abscissas have no points in common. This means that if you compare results with increasing  $N$  as a way of estimating the quadrature error, you cannot reuse the previous function evaluations. Kronrod [14] posed the problem of searching for optimal sequences of rules, each of which reuses all abscissas of its predecessor. If one starts with  $N = m$ , say, and then adds  $n$  new points, one has  $2n + m$  free parameters: the

$n$  new abscissas and weights, and  $m$  new weights for the fixed previous abscissas. The maximum degree of exactness one would expect to achieve would therefore be  $2n + m - 1$ . The question is whether this maximum degree of exactness can actually be achieved in practice, when the abscissas are required to all lie inside  $(a, b)$ . The answer to this question is not known in general.

Kronrod showed that if you choose  $n = m + 1$ , an optimal extension can be found for Gauss-Legendre quadrature. Patterson [15] showed how to compute continued extensions of this kind. Sequences such as  $N = 10, 21, 43, 87, \dots$  are popular in automatic quadrature routines [16] that attempt to integrate a function until some specified accuracy has been achieved.

#### CITED REFERENCES AND FURTHER READING:

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, §25.4.[1]
- Stroud, A.H., and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice-Hall).[2]
- Golub, G.H., and Welsch, J.H. 1969, "Calculation of Gauss Quadrature Rules," *Mathematics of Computation*, vol. 23, pp. 221–230 and A1–A10.[3]
- Wilf, H.S. 1962, *Mathematics for the Physical Sciences* (New York: Wiley), Problem 9, p. 80.[4]
- Gautschi, W. 1968, "Construction of Gauss-Christoffel Quadrature Formulas," *Mathematics of Computation*, vol. 22, pp. 251–270.[5]
- Sagar, R.P. 1991, "A Gaussian Quadrature for the Calculation of Generalized Fermi-Dirac Integrals," *Computer Physics Communications*, vol. 66, pp. 271–275.[6]
- Gautschi, W. 1982, "On Generating Orthogonal Polynomials," *SIAM Journal on Scientific and Statistical Computing*, vol. 3, pp. 289–317.[7]
- Gautschi, W. 1994, "ORTHPOL: A Package of Routines for Generating Orthogonal Polynomials and Gauss-type Quadrature Rules," *ACM Transactions on Mathematical Software*, vol. 20, pp. 21–62 (Algorithm 726 available from netlib).[8]
- Numerical Recipes Software 2007, "Implementation of Stiel," *Numerical Recipes Webnote No. 3*, at <http://www.nr.com/webnotes?3> [9]
- Sack, R.A., and Donovan, A.F. 1971/72, "An Algorithm for Gaussian Quadrature Given Modified Moments," *Numerische Mathematik*, vol. 18, pp. 465–478.[10]
- Wheeler, J.C. 1974, "Modified Moments and Gaussian Quadratures," *Rocky Mountain Journal of Mathematics*, vol. 4, pp. 287–296.[11]
- Gautschi, W. 1978, in *Recent Advances in Numerical Analysis*, C. de Boor and G.H. Golub, eds. (New York: Academic Press), pp. 45–72.[12]
- Golub, G.H. 1973, "Some Modified Matrix Eigenvalue Problems," *SIAM Review*, vol. 15, pp. 318–334.[13]
- Kronrod, A.S. 1964, *Doklady Akademii Nauk SSSR*, vol. 154, pp. 283–286 (in Russian); translated as *Soviet Physics "Doklady"*[14]
- Patterson, T.N.L. 1968, "The Optimum Addition of Points to Quadrature Formulae," *Mathematics of Computation*, vol. 22, pp. 847–856 and C1–C11; 1969, *op. cit.*, vol. 23, p. 892.[15]
- Piessens, R., de Doncker-Kapenga, E., Überhuber, C., and Kahaner, D. 1983 *QUADPACK, A Subroutine Package for Automatic Integration* (New York: Springer). Software at <http://www.netlib.org/quadpack>.[16]
- Gautschi, W. 1981, in *E.B. Christoffel*, P.L. Butzer and F. Fehér, eds. (Basel: Birkhäuser), pp. 72–147.
- Gautschi, W. 1990, in *Orthogonal Polynomials*, P. Nevai, ed. (Dordrecht: Kluwer Academic Publishers), pp. 181–216.

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §3.6.

## 4.7 Adaptive Quadrature

The idea behind adaptive quadrature is very simple. Suppose you have two different numerical estimates  $I_1$  and  $I_2$  of the integral

$$I = \int_a^b f(x) dx \quad (4.7.1)$$

Suppose  $I_1$  is more accurate. Use the relative difference between  $I_1$  and  $I_2$  as an error estimate. If it is less than  $\epsilon$ , accept  $I_1$  as the answer. Otherwise divide the interval  $[a, b]$  into two subintervals,

$$I = \int_a^m f(x) dx + \int_m^b f(x) dx \quad m = (a + b)/2 \quad (4.7.2)$$

and compute the two integrals independently. For each one, compute an  $I_1$  and  $I_2$ , estimate the error, and continue subdividing if necessary. Dividing any given subinterval stops when its contribution to  $\epsilon$  is sufficiently small. (Obviously recursion will be a good way to implement this algorithm.)

The most important criterion for an adaptive quadrature routine is reliability: If you request an accuracy of  $10^{-6}$ , you would like to be sure that the answer is at least that good. From a theoretical point of view, however, it is impossible to design an adaptive quadrature routine that will work for all possible functions. The reason is simple: A quadrature is based on the value of the integrand  $f(x)$  at a *finite* set of points. You can alter the function at all the other points in an arbitrary way without affecting the estimate your algorithm returns, while the true value of the integral changes unpredictably. Despite this point of principle, however, in practice good routines are reliable for a high fraction of functions they encounter. Our favorite routine is one proposed by Gander and Gautschi [1], which we now describe. It is relatively simple, yet scores well on reliability and efficiency.

A key component of a good adaptive algorithm is the termination criterion. The usual criterion

$$|I_1 - I_2| < \epsilon |I_1| \quad (4.7.3)$$

is problematic. In the neighborhood of a singularity,  $I_1$  and  $I_2$  might never agree to the requested tolerance, even if it's not particularly small. Instead, you need to somehow come up with an estimate of the *whole* integral  $I$  of equation (4.7.1). Then you can terminate when the error in  $I_1$  is negligible compared to the whole integral:

$$|I_1 - I_2| < \epsilon |I_s| \quad (4.7.4)$$

where  $I_s$  is the estimate of  $I$ . Gander and Gautschi implement this test by writing

```
if (is + (i1-i2) == is)
```

which is equivalent to setting  $\epsilon$  to the machine precision. However, modern optimizing compilers have become too good at recognizing that this is algebraically equivalent to

```
if (i1-i2 == 0.0)
```

which might never be satisfied in floating point arithmetic. Accordingly, we implement the test with an explicit  $\epsilon$ .

The other problem you need to take care of is when an interval gets subdivided so small that it contains no interior machine-representable point. You then need to terminate the recursion and alert the user that the full accuracy might not have been attained. In the case where the points in an interval are supposed to be  $\{a, m = (a + b)/2, b\}$ , you can test for  $m \leq a$  or  $b \leq m$ .

The lowest order integration method in the Gander-Gautschi method is the four-point Gauss-Lobatto quadrature (cf. §4.6)

$$\int_{-1}^1 f(x) dx = \frac{1}{6} [f(-1) + f(1)] + \frac{5}{6} \left[ f\left(-\frac{1}{\sqrt{5}}\right) + f\left(\frac{1}{\sqrt{5}}\right) \right] \quad (4.7.5)$$

This formula, which is exact for polynomials of degree 5, is used to compute  $I_2$ . To reuse these function evaluations in computing  $I_1$ , they find the seven-point Kronrod extension,

$$\begin{aligned} \int_{-1}^1 f(x) dx &= \frac{11}{210} [f(-1) + f(1)] + \frac{72}{245} \left[ f\left(-\sqrt{\frac{2}{3}}\right) + f\left(\sqrt{\frac{2}{3}}\right) \right] \\ &\quad + \frac{125}{294} \left[ f\left(-\frac{1}{\sqrt{5}}\right) + f\left(\frac{1}{\sqrt{5}}\right) \right] + \frac{16}{35} f(0) \end{aligned} \quad (4.7.6)$$

whose degree of exactness is nine. The formulas (4.7.5) and (4.7.6) get scaled from  $[-1, 1]$  to an arbitrary subinterval  $[a, b]$ .

For  $I_s$ , Gander and Gautschi find a 13-point Kronrod extension of equation (4.7.6), which lets them reuse the previous function evaluations. The formula is coded into the routine below. You can think of this initial 13-point evaluation as a kind of Monte Carlo sampling to get an idea of the order of magnitude of the integral. But if the integrand is smooth, this initial evaluation will itself be quite accurate already. The routine below takes advantage of this.

Note that to reuse the four function evaluations in (4.7.5) in the seven-point formula (4.7.6), you can't simply bisect intervals. But dividing into six subintervals works (there are six intervals between seven points).

To use the routine, you need to initialize an `Adapt` object with your required tolerance,

```
Adapt s(1.0e-6);
```

and then call the `integrate` function:

```
ans=s.integrate(func,a,b);
```

You should check that the desired tolerance could be met:

```
if (s.out_of_tolerance)
    cout << "Required tolerance may not be met" << endl;
```

The smallest allowed tolerance is 10 times the machine precision. If you enter a smaller tolerance, it gets reset internally. (The routine will work using the machine precision itself, but then it usually just takes lots of function evaluations for little additional benefit.)

The implementation of the `Adapt` object is given in a Webnote [2].

Adaptive quadrature is no panacea. The above routine has no special machinery to deal with singularities other than to refine the neighboring intervals. By using

suitable schemes for  $I_1$  and  $I_2$ , one can customize an adaptive routine to deal with a particular kind of singularity (cf. [3]).

#### CITED REFERENCES AND FURTHER READING:

- Gander, W., and Gautschi, W. 2000, "Adaptive Quadrature — Revisited," *BIT* vol. 40, pp. 84–101.[1]
- Numerical Recipes Software 2007, "Implementation of Adapt," *Numerical Recipes Webnote No. 4*, at <http://www.nr.com/webnotes?4> [2]
- Piessens, R., de Doncker-Kapenga, E., Überhuber, C., and Kahaner, D. 1983 *QUADPACK, A Subroutine Package for Automatic Integration* (New York: Springer). Software at <http://www.netlib.org/quadpack>.[3]
- Davis, P.J., and Rabinowitz, P. 1984, *Methods of Numerical Integration*, 2nd ed., (Orlando, FL: Academic Press), Chapter 6.

## 4.8 Multidimensional Integrals

Integrals of functions of several variables, over regions with dimension greater than one, are *not easy*. There are two reasons for this. First, the number of function evaluations needed to sample an  $N$ -dimensional space increases as the  $N$ th power of the number needed to do a one-dimensional integral. If you need 30 function evaluations to do a one-dimensional integral crudely, then you will likely need on the order of 30000 evaluations to reach the same crude level for a three-dimensional integral. Second, the region of integration in  $N$ -dimensional space is defined by an  $N - 1$  dimensional boundary that can itself be terribly complicated: It need not be convex or simply connected, for example. By contrast, the boundary of a one-dimensional integral consists of two numbers, its upper and lower limits.

The first question to be asked, when faced with a multidimensional integral, is, can it be reduced analytically to a lower dimensionality? For example, so-called *iterated integrals* of a function of one variable  $f(t)$  can be reduced to one-dimensional integrals by the formula

$$\int_0^x dt_n \int_0^{t_n} dt_{n-1} \cdots \int_0^{t_3} dt_2 \int_0^{t_2} f(t_1) dt_1 = \frac{1}{(n-1)!} \int_0^x (x-t)^{n-1} f(t) dt \quad (4.8.1)$$

Alternatively, the function may have some special symmetry in the way it depends on its independent variables. If the boundary also has this symmetry, then the dimension can be reduced. In three dimensions, for example, the integration of a spherically symmetric function over a spherical region reduces, in polar coordinates, to a one-dimensional integral.

The next questions to be asked will guide your choice between two entirely different approaches to doing the problem. The questions are: Is the shape of the boundary of the region of integration simple or complicated? Inside the region, is the integrand smooth and simple, or complicated, or locally strongly peaked? Does the problem require high accuracy, or does it require an answer accurate only to a percent, or a few percent?

If your answers are that the boundary is complicated, the integrand is *not* strongly peaked in very small regions, and relatively low accuracy is tolerable, then your problem is a good candidate for *Monte Carlo integration*. This method is very straightforward to program, in its cruder forms. One needs only to know a region with simple boundaries that *includes* the complicated region of integration, plus a method of determining whether a random point is inside or outside the region of integration. Monte Carlo integration evaluates the function at a random sample of points and estimates its integral based on that random sample. We will discuss it in more detail, and with more sophistication, in Chapter 7.

If the boundary is simple, and the function is very smooth, then the remaining approaches, breaking up the problem into repeated one-dimensional integrals, or multidimensional Gaussian quadratures, will be effective and relatively fast [1]. If you require high accuracy, these approaches are in any case the *only* ones available to you, since Monte Carlo methods are by nature asymptotically slow to converge.

For low accuracy, use repeated one-dimensional integration or multidimensional Gaussian quadratures when the integrand is slowly varying and smooth in the region of integration, Monte Carlo when the integrand is oscillatory or discontinuous but not strongly peaked in small regions.

If the integrand *is* strongly peaked in small regions, and you know where those regions are, break the integral up into several regions so that the integrand is smooth in each, and do each separately. If you don't know where the strongly peaked regions are, you might as well (at the level of sophistication of this book) quit: It is hopeless to expect an integration routine to search out unknown pockets of large contribution in a huge  $N$ -dimensional space. (But see §7.9.)

If, on the basis of the above guidelines, you decide to pursue the repeated one-dimensional integration approach, here is how it works. For definiteness, we will consider the case of a three-dimensional integral in  $x, y, z$ -space. Two dimensions, or more than three dimensions, are entirely analogous.

The first step is to specify the region of integration by (i) its lower and upper limits in  $x$ , which we will denote  $x_1$  and  $x_2$ ; (ii) its lower and upper limits in  $y$  at a specified value of  $x$ , denoted  $y_1(x)$  and  $y_2(x)$ ; and (iii) its lower and upper limits in  $z$  at specified  $x$  and  $y$ , denoted  $z_1(x, y)$  and  $z_2(x, y)$ . In other words, find the numbers  $x_1$  and  $x_2$ , and the functions  $y_1(x)$ ,  $y_2(x)$ ,  $z_1(x, y)$ , and  $z_2(x, y)$  such that

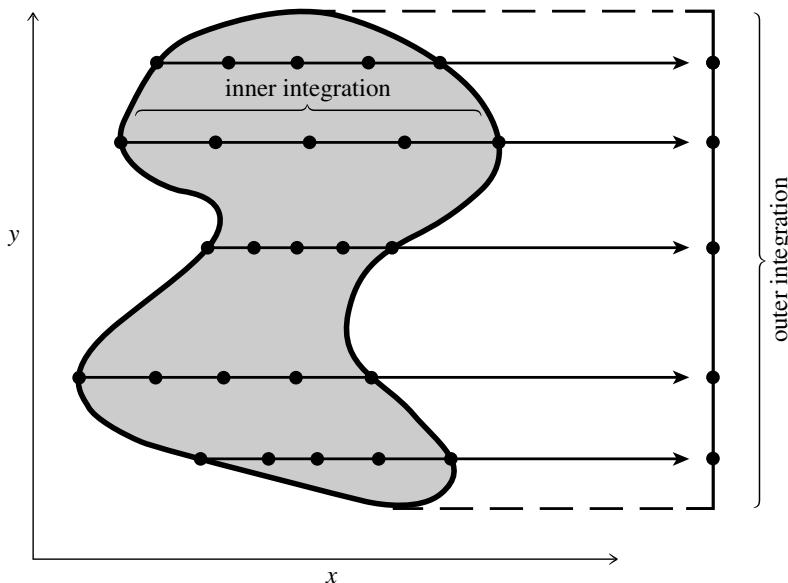
$$\begin{aligned} I &\equiv \iiint dx dy dz f(x, y, z) \\ &= \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x, y)}^{z_2(x, y)} dz f(x, y, z) \end{aligned} \quad (4.8.2)$$

For example, a two-dimensional integral over a circle of radius one centered on the origin becomes

$$\int_{-1}^1 dx \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} dy f(x, y) \quad (4.8.3)$$

Now we can define a function  $G(x, y)$  that does the innermost integral,

$$G(x, y) \equiv \int_{z_1(x, y)}^{z_2(x, y)} f(x, y, z) dz \quad (4.8.4)$$



**Figure 4.8.1.** Function evaluations for a two-dimensional integral over an irregular region, shown schematically. The outer integration routine, in  $y$ , requests values of the inner,  $x$ , integral at locations along the  $y$ -axis of its own choosing. The inner integration routine then evaluates the function at  $x$  locations suitable to it. This is more accurate in general than, e.g., evaluating the function on a Cartesian mesh of points.

and a function  $H(x)$  that does the integral of  $G(x, y)$ ,

$$H(x) \equiv \int_{y_1(x)}^{y_2(x)} G(x, y) dy \quad (4.8.5)$$

and finally our answer as an integral over  $H(x)$

$$I = \int_{x_1}^{x_2} H(x) dx \quad (4.8.6)$$

In an implementation of equations (4.8.4) – (4.8.6), some basic one-dimensional integration routine (e.g., `qgaus` in the program following) gets called recursively: once to evaluate the outer integral  $I$ , then many times to evaluate the middle integral  $H$ , then even more times to evaluate the inner integral  $G$  (see Figure 4.8.1). Current values of  $x$  and  $y$ , and the pointers to the user-supplied functions for the integrand and the boundaries, are passed “over the head” of the intermediate calls through member variables in the three functors defining the integrands for  $G$ ,  $H$  and  $I$ .

```
quad3d.h struct NRf3 {
    Doub xsav,ysav;
    Doub (*func3d)(const Doub, const Doub, const Doub);
    Doub operator()(const Doub z)  The integrand f(x,y,z) evaluated at fixed x and
    {                                y.
        return func3d(xsav,ysav,z);
    }
};
```

```

struct NRf2 {
    NRf3 f3;
    Doub (*z1)(Doub, Doub);
    Doub (*z2)(Doub, Doub);
    NRf2(Doub zz1(Doub, Doub), Doub zz2(Doub, Doub)) : z1(zz1), z2(zz2) {}
    Doub operator()(const Doub y)  This is G of eq. (4.8.4).
    {
        f3.y sav=y;
        return qgaus(f3,z1(f3.x sav,y),z2(f3.x sav,y));
    }
};

struct NRf1 {
    Doub (*y1)(Doub);
    Doub (*y2)(Doub);
    NRf2 f2;
    NRf1(Doub yy1(Doub), Doub yy2(Doub), Doub z1(Doub, Doub),
          Doub z2(Doub, Doub)) : y1(yy1),y2(yy2), f2(z1,z2) {}
    Doub operator()(const Doub x)  This is H of eq. (4.8.5).
    {
        f2.f3.x sav=x;
        return qgaus(f2,y1(x),y2(x));
    }
};

template <class T>
Doub quad3d(T &func, const Doub x1, const Doub x2, Doub y1(Doub), Doub y2(Doub),
             Doub z1(Doub, Doub), Doub z2(Doub, Doub))
Returns the integral of a user-supplied function func over a three-dimensional region specified by the limits x1, x2, and by the user-supplied functions y1, y2, z1, and z2, as defined in (4.8.2). Integration is performed by calling qgaus recursively.
{
    NRf1 f1(y1,y2,z1,z2);
    f1.f2.f3.func3d=func;
    return qgaus(f1,x1,x2);
}

```

Note that while the function to be integrated can be supplied either as a simple function

```
Doub func(const Doub x, const Doub y, const Doub z);
```

or as the equivalent functor, the functions defining the boundary can only be functions:

```

Doub y1(const Doub x);
Doub y2(const Doub x);
Doub z1(const Doub x, const Doub y);
Doub z2(const Doub x, const Doub y);

```

This is for simplicity; you can easily modify the code to take functors if you need to.

The Gaussian quadrature routine used in *quad3d* is simple, but its accuracy is not controllable. An alternative is to use a one-dimensional integration routine like *qtrap*, *qsimp* or *qromb*, which have a user-definable tolerance *eps*. Simply replace all occurrences of *qgaus* in *quad3d* by *qromb*, say.

Note that multidimensional integration is likely to be very slow if you try for too much accuracy. You should almost certainly increase the default *eps* in *qromb* from  $10^{-10}$  to  $10^{-6}$  or bigger. You should also decrease *JMAX* to avoid a lot of waiting around for an answer. Some people advocate using a smaller *eps* for the inner quadrature (over *z* in our routine) than for the outer quadratures (over *x* or *y*).

**CITED REFERENCES AND FURTHER READING:**

- Stroud, A.H. 1971, *Approximate Calculation of Multiple Integrals* (Englewood Cliffs, NJ: Prentice-Hall). [1]
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §7.7, p. 318.
- Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.2.5, p. 307.
- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, equations 25.4.58ff.

# Evaluation of Functions

## 5.0 Introduction

The purpose of this chapter is to acquaint you with a selection of the techniques that are frequently used in evaluating functions. In Chapter 6, we will apply and illustrate these techniques by giving routines for a variety of specific functions. The purposes of this chapter and the next are thus mostly congruent. Occasionally, however, the method of choice for a particular special function in Chapter 6 is peculiar to that function. By comparing this chapter to the next one, you should get some idea of the balance between “general” and “special” methods that occurs in practice.

Insofar as that balance favors general methods, this chapter should give you ideas about how to write your own routine for the evaluation of a function that, while “special” to you, is not so special as to be included in Chapter 6 or the standard function libraries.

### CITED REFERENCES AND FURTHER READING:

- Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall).  
Lanczos, C. 1956, *Applied Analysis*; reprinted 1988 (New York: Dover), Chapter 7.

## 5.1 Polynomials and Rational Functions

A polynomial of degree  $N$  is represented numerically as a stored array of coefficients,  $c[j]$  with  $j = 0, \dots, N$ . We will always take  $c[0]$  to be the constant term in the polynomial and  $c[N]$  the coefficient of  $x^N$ ; but of course other conventions are possible. There are two kinds of manipulations that you can do with a polynomial: *numerical* manipulations (such as evaluation), where you are given the numerical value of its argument, or *algebraic* manipulations, where you want to transform the coefficient array in some way without choosing any particular argument. Let’s start with the numerical.

We assume that you know enough *never* to evaluate a polynomial this way:

```
p=c[0]+c[1]*x+c[2]*x*x+c[3]*x*x*x+c[4]*x*x*x*x;
```

or (even worse!),

```
p=c[0]+c[1]*x+c[2]*pow(x,2.0)+c[3]*pow(x,3.0)+c[4]*pow(x,4.0);
```

Come the (computer) revolution, all persons found guilty of such criminal behavior will be summarily executed, and their programs won't be! It is a matter of taste, however, whether to write

```
p=c[0]+x*(c[1]+x*(c[2]+x*(c[3]+x*c[4])));
```

or

```
p=((c[4]*x+c[3])*x+c[2])*x+c[1])*x+c[0];
```

If the number of coefficients  $c[0..n-1]$  is large, one writes

```
p=c[n-1];
for(j=n-2;j>=0;j--) p=p*x+c[j];
```

or

```
p=c[j=n-1];
while (j>0) p=p*x+c[--j];
```

We can formalize this by defining a function object (or functor) that binds a reference to an array of coefficients and endows them with a polynomial evaluation function,

```
poly.h
struct Poly {
    Polynomial function object that binds a reference to a vector of coefficients.
    VecDoub &c;
    Poly(VecDoub &cc) : c(cc) {}
    Doub operator() (Doub x) {
        Returns the value of the polynomial at x.
        Int j;
        Doub p = c[j=c.size()-1];
        while (j>0) p = p*x + c[--j];
        return p;
    }
};
```

which allows you to write things like

```
y = Poly(c)(x);
```

where  $c$  is a coefficient vector.

Another useful trick is for evaluating a polynomial  $P(x)$  and its derivative  $dP(x)/dx$  simultaneously:

```
p=c[n-1];
dp=0.;
for(j=n-2;j>=0;j--) {dp=dp*x+p; p=p*x+c[j];}
```

or

```
p=c[j=n-1];
dp=0.;
while (j>0) {dp=dp*x+p; p=p*x+c[--j];}
```

which yields the polynomial as  $p$  and its derivative as  $dp$  using coefficients  $c[0..n-1]$ .

The above trick, which is basically *synthetic division* [1,2], generalizes to the evaluation of the polynomial and  $nd$  of its derivatives simultaneously:

```
void ddpoly(VecDoub_I &c, const Doub x, VecDoub_O &pd)
Given the coefficients of a polynomial of degree nc as an array c[0..nc] of size nc+1 (with
c[0] being the constant term), and given a value x, this routine fills an output array pd of size
nd+1 with the value of the polynomial evaluated at x in pd[0], and the first nd derivatives at
x in pd[1..nd].
{
    Int nnd,j,i,nc=c.size()-1,nd=pd.size()-1;
    Doub cnst=1.0;
    pd[0]=c[nc];
    for (j=1;j<nd+1;j++) pd[j]=0.0;
    for (i=nc-1;i>=0;i--) {
        nnd=(nd < (nc-i) ? nd : nc-i);
        for (j=nnd;j>0;j--) pd[j]=pd[j]*x+pd[j-1];
        pd[0]=pd[0]*x+c[i];
    }
    for (i=2;i<nd+1;i++) {           After the first derivative, factorial constants come in.
        cnst *= i;
        pd[i] *= cnst;
    }
}
```

As a curiosity, you might be interested to know that polynomials of degree  $n > 3$  can be evaluated in *fewer* than  $n$  multiplications, at least if you are willing to precompute some auxiliary coefficients and, in some cases, do some extra addition. For example, the polynomial

$$P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 \quad (5.1.1)$$

where  $a_4 > 0$ , can be evaluated with three multiplications and five additions as follows:

$$P(x) = [(Ax + B)^2 + Ax + C][(Ax + B)^2 + D] + E \quad (5.1.2)$$

where  $A, B, C, D$ , and  $E$  are to be precomputed by

$$\begin{aligned} A &= (a_4)^{1/4} \\ B &= \frac{a_3 - A^3}{4A^3} \\ D &= 3B^2 + 8B^3 + \frac{a_1A - 2a_2B}{A^2} \\ C &= \frac{a_2}{A^2} - 2B - 6B^2 - D \\ E &= a_0 - B^4 - B^2(C + D) - CD \end{aligned} \quad (5.1.3)$$

Fifth-degree polynomials can be evaluated in four multiplies and five adds; sixth-degree polynomials can be evaluated in four multiplies and seven adds; if any of this strikes you as interesting, consult references [3-5]. The subject has something of the same flavor as that of fast matrix multiplication, discussed in §2.11.

Turn now to algebraic manipulations. You multiply a polynomial of degree  $n - 1$  (array of range  $[0..n-1]$ ) by a monomial factor  $x - a$  by a bit of code like the following,

```
c[n]=c[n-1];
for (j=n-1;j>=1;j--) c[j]=c[j-1]-c[j]*a;
c[0] *= (-a);
```

Likewise, you divide a polynomial of degree  $n$  by a monomial factor  $x - a$  (synthetic division again) using

```
rem=c[n];
c[n]=0.;
for(i=n-1;i>=0;i--) {
    swap=c[i];
    c[i]=rem;
    rem=swap+rem*a;
}
```

which leaves you with a new polynomial array and a numerical remainder `rem`.

Multiplication of two general polynomials involves straightforward summing of the products, each involving one coefficient from each polynomial. Division of two general polynomials, while it can be done awkwardly in the fashion taught using pencil and paper, is susceptible to a good deal of streamlining. Witness the following routine based on the algorithm in [3].

```
poly.h void poldiv(VecDoub_I &u, VecDoub_I &v, VecDoub_O &q, VecDoub_O &r)
Divide a polynomial u by a polynomial v, and return the quotient and remainder polynomials
in q and r, respectively. The four polynomials are represented as vectors of coefficients, each
starting with the constant term. There is no restriction on the relative lengths of u and v, and
either may have trailing zeros (represent a lower degree polynomial than its length allows). q
and r are returned with the size of u, but will usually have trailing zeros.
{
    Int k,j,n=u.size()-1,nv=v.size()-1;
    while (nv >= 0 && v[nv] == 0.) nv--;
    if (nv < 0) throw("poldiv divide by zero polynomial");
    r = u;                                May do a resize.
    q.assign(u.size(),0.);                  May do a resize.
    for (k=n-nv;k>=0;k--) {
        q[k]=r[nv+k]/v[nv];
        for (j=nv+k-1;j>=k;j--) r[j] -= q[k]*v[j-k];
    }
    for (j=nv;j<=n;j++) r[j]=0.0;
}
```

### 5.1.1 Rational Functions

You evaluate a rational function like

$$R(x) = \frac{P_\mu(x)}{Q_v(x)} = \frac{p_0 + p_1x + \cdots + p_\mu x^\mu}{q_0 + q_1x + \cdots + q_v x^v} \quad (5.1.4)$$

in the obvious way, namely as two separate polynomials followed by a divide. As a matter of convention one usually chooses  $q_0 = 1$ , obtained by dividing the numerator and denominator by any other  $q_0$ . In that case, it is often convenient to have both sets of coefficients, omitting  $q_0$ , stored in a single array, in the order

$$(p_0, p_1, \dots, p_\mu, q_1, \dots, q_v) \quad (5.1.5)$$

The following object encapsulates a rational function. It provides constructors from either separate numerator and denominator polynomials, or a single array like (5.1.5) with explicit values for  $n = \mu + 1$  and  $d = v + 1$ . The evaluation function makes `Ratfn` a functor, like `Poly`. We'll make use of this object in §5.12 and §5.13.

```

struct Ratfn {
    Function object for a rational function.
    VecDoub cofs;
    Int nn,dd;           Number of numerator, denominator coefficients.

    Ratfn(VecDoub_I &num, VecDoub_I &den) : cofs(num.size()+den.size()-1),
    nn(num.size()), dd(den.size()) {
        Constructor from numerator, denominator polynomials (as coefficient vectors).
        Int j;
        for (j=0;j<nn;j++) cofs[j] = num[j]/den[0];
        for (j=1;j<dd;j++) cofs[j+nn-1] = den[j]/den[0];
    }

    Ratfn(VecDoub_I &coffs, const Int n, const Int d) : cofs(coffs), nn(n),
    dd(d) {}
        Constructor from coefficients already normalized and in a single array.

    Doub operator() (Doub x) const {
        Evaluate the rational function at x and return result.
        Int j;
        Doub sumn = 0., sumd = 0.;
        for (j=nn-1;j>=0;j--) sumn = sumn*x + cofs[j];
        for (j=nn+dd-2;j>=nn;j--) sumd = sumd*x + cofs[j];
        return sumn/(1.0+x*sumd);
    }

};
```

### 5.1.2 Parallel Evaluation of a Polynomial

A polynomial of degree  $N$  can be evaluated in about  $\log_2 N$  parallel steps [6]. This is best illustrated by an example, for example with  $N = 5$ . Start with the vector of coefficients, imagining appended zeros:

$$c_0, \quad c_1, \quad c_2, \quad c_3, \quad c_4, \quad c_5, \quad 0, \quad \dots \quad (5.1.6)$$

Now add the elements by pairs, multiplying the second of each pair by  $x$ :

$$c_0 + c_1x, \quad c_2 + c_3x, \quad c_4 + c_5x, \quad 0, \quad \dots \quad (5.1.7)$$

Now the same operation, but with the multiplier  $x^2$ :

$$(c_0 + c_1x) + (c_2 + c_3x)x^2, \quad (c_4 + c_5x) + (0)x^2, \quad 0 \quad \dots \quad (5.1.8)$$

And a final time with multiplier  $x^4$ :

$$[(c_0 + c_1x) + (c_2 + c_3x)x^2] + [(c_4 + c_5x) + (0)x^2]x^4, \quad 0 \quad \dots \quad (5.1.9)$$

We are left with a vector of (active) length 1, whose value is the desired polynomial evaluation. You can see that the zeros are just a bookkeeping device for taking care of the case where the active subvector has an odd length; in an actual implementation you can avoid most operations on the zeros. This parallel method generally has better roundoff properties than the standard sequential coding.

#### CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), pp. 183, 190.[1]

- Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 361–363.[2]
- Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6.[3]
- Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 4.
- Winograd, S. 1970, “On the number of multiplications necessary to compute certain functions,” *Communications on Pure and Applied Mathematics*, vol. 23, pp. 165–179.[4]
- Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley).[5]
- Estrin, G. 1960, quoted in Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6.4.[6]

## 5.2 Evaluation of Continued Fractions

Continued fractions are often powerful ways of evaluating functions that occur in scientific applications. A continued fraction looks like this:

$$f(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cfrac{a_5}{b_5 + \dots}}}}} \quad (5.2.1)$$

Printers prefer to write this as

$$f(x) = b_0 + \frac{a_1}{b_1 +} \frac{a_2}{b_2 +} \frac{a_3}{b_3 +} \frac{a_4}{b_4 +} \frac{a_5}{b_5 +} \dots \quad (5.2.2)$$

In either (5.2.1) or (5.2.2), the  $a$ 's and  $b$ 's can themselves be functions of  $x$ , usually linear or quadratic monomials at worst (i.e., constants times  $x$  or times  $x^2$ ). For example, the continued fraction representation of the tangent function is

$$\tan x = \frac{x}{1 -} \frac{x^2}{3 -} \frac{x^2}{5 -} \frac{x^2}{7 -} \dots \quad (5.2.3)$$

Continued fractions frequently converge much more rapidly than power series expansions, and in a much larger domain in the complex plane (not necessarily including the domain of convergence of the series, however). Sometimes the continued fraction converges best where the series does worst, although this is not a general rule. Blanch [1] gives a good review of the most useful convergence tests for continued fractions.

There are standard techniques, including the important *quotient-difference algorithm*, for going back and forth between continued fraction approximations, power series approximations, and rational function approximations. Consult Acton [2] for an introduction to this subject, and Fike [3] for further details and references.

How do you tell how far to go when evaluating a continued fraction? Unlike a series, you can't just evaluate equation (5.2.1) from left to right, stopping when the change is small. Written in the form of (5.2.1), the only way to evaluate the continued fraction is from right to left, first (blindly!) guessing how far out to start. This is not the right way.

The right way is to use a result that relates continued fractions to rational approximations, and that gives a means of evaluating (5.2.1) or (5.2.2) from left to right. Let  $f_n$  denote the result of evaluating (5.2.2) with coefficients through  $a_n$  and  $b_n$ . Then

$$f_n = \frac{A_n}{B_n} \quad (5.2.4)$$

where  $A_n$  and  $B_n$  are given by the following recurrence:

$$\begin{aligned} A_{-1} &\equiv 1 & B_{-1} &\equiv 0 \\ A_0 &\equiv b_0 & B_0 &\equiv 1 \\ A_j = b_j A_{j-1} + a_j A_{j-2} && B_j = b_j B_{j-1} + a_j B_{j-2} & j = 1, 2, \dots, n \end{aligned} \quad (5.2.5)$$

This method was invented by J. Wallis in 1655 (!) and is discussed in his *Arithmetica Infinitorum* [4]. You can easily prove it by induction.

In practice, this algorithm has some unattractive features: The recurrence (5.2.5) frequently generates very large or very small values for the partial numerators and denominators  $A_j$  and  $B_j$ . There is thus the danger of overflow or underflow of the floating-point representation. However, the recurrence (5.2.5) is linear in the  $A$ 's and  $B$ 's. At any point you can rescale the currently saved two levels of the recurrence, e.g., divide  $A_j$ ,  $B_j$ ,  $A_{j-1}$ , and  $B_{j-1}$  all by  $B_j$ . This incidentally makes  $A_j = f_j$  and is convenient for testing whether you have gone far enough: See if  $f_j$  and  $f_{j-1}$  from the last iteration are as close as you would like them to be. If  $B_j$  happens to be zero, which can happen, just skip the renormalization for this cycle. A fancier level of optimization is to renormalize only when an overflow is imminent, saving the unnecessary divides. In fact, the C library function `ldexp` can be used to avoid division entirely. (See the end of §6.5 for an example.)

Two newer algorithms have been proposed for evaluating continued fractions. *Steed's method* does not use  $A_j$  and  $B_j$  explicitly, but only the ratio  $D_j = B_{j-1}/B_j$ . One calculates  $D_j$  and  $\Delta f_j = f_j - f_{j-1}$  recursively using

$$D_j = 1/(b_j + a_j D_{j-1}) \quad (5.2.6)$$

$$\Delta f_j = (b_j D_j - 1)\Delta f_{j-1} \quad (5.2.7)$$

*Steed's method* (see, e.g., [5]) avoids the need for rescaling of intermediate results. However, for certain continued fractions you can occasionally run into a situation where the denominator in (5.2.6) approaches zero, so that  $D_j$  and  $\Delta f_j$  are very large. The next  $\Delta f_{j+1}$  will typically cancel this large change, but with loss of accuracy in the numerical running sum of the  $f_j$ 's. It is awkward to program around this, so *Steed's method* can be recommended only for cases where you know in advance that no denominator can vanish. We will use it for a special purpose in the routine `besselik` (§6.6).

The best general method for evaluating continued fractions seems to be the *modified Lenz's method* [6]. The need for rescaling intermediate results is avoided by using *both* the ratios

$$C_j = A_j/A_{j-1}, \quad D_j = B_{j-1}/B_j \quad (5.2.8)$$

and calculating  $f_j$  by

$$f_j = f_{j-1} C_j D_j \quad (5.2.9)$$

From equation (5.2.5), one easily shows that the ratios satisfy the recurrence relations

$$D_j = 1/(b_j + a_j D_{j-1}), \quad C_j = b_j + a_j / C_{j-1} \quad (5.2.10)$$

In this algorithm there is the danger that the denominator in the expression for  $D_j$ , or the quantity  $C_j$  itself, might approach zero. Either of these conditions invalidates (5.2.10). However, Thompson and Barnett [5] show how to modify Lentz's algorithm to fix this: Just shift the offending term by a small amount, e.g.,  $10^{-30}$ . If you work through a cycle of the algorithm with this prescription, you will see that  $f_{j+1}$  is accurately calculated.

In detail, the modified Lentz's algorithm is this:

- Set  $f_0 = b_0$ ; if  $b_0 = 0$ , set  $f_0 = \text{tiny}$ .
- Set  $C_0 = f_0$ .
- Set  $D_0 = 0$ .
- For  $j = 1, 2, \dots$

```

Set  $D_j = b_j + a_j D_{j-1}$ .
If  $D_j = 0$ , set  $D_j = \text{tiny}$ .
Set  $C_j = b_j + a_j / C_{j-1}$ .
If  $C_j = 0$ , set  $C_j = \text{tiny}$ .
Set  $D_j = 1/D_j$ .
Set  $\Delta_j = C_j D_j$ .
Set  $f_j = f_{j-1} \Delta_j$ .
If  $|\Delta_j - 1| < \text{eps}$ , then exit.
```

Here  $\text{eps}$  is your floating-point precision, say  $10^{-7}$  or  $10^{-15}$ . The parameter  $\text{tiny}$  should be less than typical values of  $\text{eps} |b_j|$ , say  $10^{-30}$ .

The above algorithm assumes that you can terminate the evaluation of the continued fraction when  $|f_j - f_{j-1}|$  is sufficiently small. This is usually the case, but by no means guaranteed. Jones [7] gives a list of theorems that can be used to justify this termination criterion for various kinds of continued fractions.

There is at present no rigorous analysis of error propagation in Lentz's algorithm. However, empirical tests suggest that it is at least as good as other methods.

### 5.2.1 Manipulating Continued Fractions

Several important properties of continued fractions can be used to rewrite them in forms that can speed up numerical computation. An *equivalence transformation*

$$a_n \rightarrow \lambda a_n, \quad b_n \rightarrow \lambda b_n, \quad a_{n+1} \rightarrow \lambda a_{n+1} \quad (5.2.11)$$

leaves the value of a continued fraction unchanged. By a suitable choice of the scale factor  $\lambda$  you can often simplify the form of the  $a$ 's and the  $b$ 's. Of course, you can carry out successive equivalence transformations, possibly with different  $\lambda$ 's, on successive terms of the continued fraction.

The *even* and *odd* parts of a continued fraction are continued fractions whose successive convergents are  $f_{2n}$  and  $f_{2n+1}$ , respectively. Their main use is that they converge twice as fast as the original continued fraction, and so if their terms are not much more complicated than the terms in the original, there can be a big savings in computation. The formula for the even part of (5.2.2) is

$$f_{\text{even}} = d_0 + \frac{c_1}{d_1 +} \frac{c_2}{d_2 +} \dots \quad (5.2.12)$$

where in terms of intermediate variables

$$\begin{aligned}\alpha_1 &= \frac{a_1}{b_1} \\ \alpha_n &= \frac{a_n}{b_n b_{n-1}}, \quad n \geq 2\end{aligned}\tag{5.2.13}$$

we have

$$\begin{aligned}d_0 &= b_0, \quad c_1 = \alpha_1, \quad d_1 = 1 + \alpha_2 \\ c_n &= -\alpha_{2n-1}\alpha_{2n-2}, \quad d_n = 1 + \alpha_{2n-1} + \alpha_{2n}, \quad n \geq 2\end{aligned}\tag{5.2.14}$$

You can find the similar formula for the odd part in the review by Blanch [1]. Often a combination of the transformations (5.2.14) and (5.2.11) is used to get the best form for numerical work.

We will make frequent use of continued fractions in the next chapter.

#### CITED REFERENCES AND FURTHER READING:

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, §3.10.
- Blanch, G. 1964, "Numerical Evaluation of Continued Fractions," *SIAM Review*, vol. 6, pp. 383–421.[1]
- Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), Chapter 11.[2]
- Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 1.
- Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall), §8.2, §10.4, and §10.5.[3]
- Wallis, J. 1695, in *Opera Mathematica*, vol. 1, p. 355, Oxoniae e Theatro Sheldoniano. Reprinted by Georg Olms Verlag, Hildesheim, New York (1972).[4]
- Thompson, I.J., and Barnett, A.R. 1986, "Coulomb and Bessel Functions of Complex Arguments and Order," *Journal of Computational Physics*, vol. 64, pp. 490–509.[5]
- Lentz, W.J. 1976, "Generating Bessel Functions in Mie Scattering Calculations Using Continued Fractions," *Applied Optics*, vol. 15, pp. 668–671.[6]
- Jones, W.B. 1973, in *Padé Approximants and Their Applications*, P.R. Graves-Morris, ed. (London: Academic Press), p. 125.[7]

## 5.3 Series and Their Convergence

Everybody knows that an analytic function can be expanded in the neighborhood of a point  $x_0$  in a power series,

$$f(x) = \sum_{k=0}^{\infty} a_k (x - x_0)^k\tag{5.3.1}$$

Such series are straightforward to evaluate. You don't, of course, evaluate the  $k$ th power of  $x - x_0$  *ab initio* for each term; rather, you keep the  $k - 1$ st power and update

it with a multiply. Similarly, the form of the coefficients  $a_k$  is often such as to make use of previous work: Terms like  $k!$  or  $(2k)!$  can be updated in a multiply or two.

How do you know when you have summed enough terms? In practice, the terms had better be getting small fast, otherwise the series is not a good technique to use in the first place. While not mathematically rigorous in all cases, standard practice is to quit when the term you have just added is smaller in magnitude than some small  $\epsilon$  times the magnitude of the sum thus far accumulated. (But watch out if isolated instances of  $a_k = 0$  are possible!)

Sometimes you will want to compute a function from a series representation even when the computation is *not* efficient. For example, you may be using the values obtained to fit the function to an approximating form that you will use subsequently (cf. §5.8). If you are summing very large numbers of slowly convergent terms, pay attention to roundoff errors! In floating-point representation it is more accurate to sum a list of numbers in the order starting with the smallest one, rather than starting with the largest one. It is even better to group terms pairwise, then in pairs of pairs, etc., so that all additions involve operands of comparable magnitude.

A weakness of a power series representation is that it is guaranteed *not* to converge farther than that distance from  $x_0$  at which a singularity is encountered *in the complex plane*. This catastrophe is not usually unexpected: When you find a power series in a book (or when you work one out yourself), you will generally also know the radius of convergence. An insidious problem occurs with series that converge everywhere (in the mathematical sense), but almost nowhere fast enough to be useful in a numerical method. Two familiar examples are the sine function and the Bessel function of the first kind,

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \quad (5.3.2)$$

$$J_n(x) = \left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!(k+n)!} \quad (5.3.3)$$

Both of these series converge for all  $x$ . But both don't even start to converge until  $k \gg |x|$ ; before this, their terms are increasing. Even worse, the terms alternate in sign, leading to large cancellation errors with finite precision arithmetic. This makes these series useless for large  $x$ .

### 5.3.1 Divergent Series

Divergent series are often very useful. One class consists of power series outside their radius of convergence, which can often be summed by the acceleration techniques we will describe below. Another class is asymptotic series, such as the Euler series that comes from Euler's integral (related to the exponential integral  $E_1$ ):

$$E(x) = \int_0^\infty \frac{e^{-t}}{1+xt} dt \simeq \sum_{k=0}^{\infty} (-1)^k k! x^k \quad (5.3.4)$$

Here the series is derived by expanding  $(1+xt)^{-1}$  in powers of  $x$  and integrating term by term. The series diverges for all  $x \neq 0$ . For  $x = 0.1$ , the series gives only three significant digits before diverging. Nevertheless, convergence acceleration

techniques allow effortless evaluation of the function  $E(x)$ , even for  $x \sim 2$ , when the series is wildly divergent!

### 5.3.2 Accelerating the Convergence of Series

There are several tricks for accelerating the rate of convergence of a series or, equivalently, of a sequence of partial sums

$$s_n = \sum_{k=0}^n a_k \quad (5.3.5)$$

(We'll use the terms sequence and series interchangeably in this section.) An excellent review has been given by Weniger [1]. Before we can describe the tricks and when to use them, we need to classify some of the ways in which a sequence can converge. Suppose  $s_n$  converges to  $s$ , say, and that

$$\lim_{n \rightarrow \infty} \frac{a_{n+1}}{a_n} = \rho \quad (5.3.6)$$

If  $0 < |\rho| < 1$ , we say the convergence is *linear*; if  $\rho = 1$ , it is *logarithmic*; and if  $\rho = 0$ , it is *hyperlinear*. Of course, if  $|\rho| > 1$ , the sequence diverges. (More rigorously, this classification should be given in terms of the so-called remainders  $s_n - s$  [1]. However, our definition is more practical and is equivalent if we restrict the logarithmic case to terms of the same sign.)

The prototype of linear convergence is a geometric series,

$$s_n = \sum_{k=0}^n x^k = \frac{1 - x^{n+1}}{1 - x} \quad (5.3.7)$$

It is easy to see that  $\rho = x$ , and so we have linear convergence for  $0 < |x| < 1$ . The prototype of logarithmic convergence is the series for the Riemann zeta function,

$$\zeta(x) = \sum_{k=1}^{\infty} \frac{1}{k^x}, \quad x > 1 \quad (5.3.8)$$

which is notoriously slowly convergent, especially as  $x \rightarrow 1$ . The series (5.3.2) and (5.3.3), or the series for  $e^x$ , exemplify hyperlinear convergence. We see that hyperlinear convergence doesn't necessarily imply that the series is easy to evaluate for all values of  $x$ . Sometimes convergence acceleration is helpful only once the terms start decreasing.

Probably the most famous series transformation for accelerating convergence is the Euler transformation (see, e.g., [2,3]), which dates from 1755. Euler's transformation works on *alternating series* (where the terms in the sum alternate in sign). Generally it is advisable to do a small number of terms directly, through term  $n - 1$ , say, and then apply the transformation to the rest of the series beginning with term  $n$ . The formula (for  $n$  even) is

$$\sum_{s=0}^{\infty} (-1)^s a_s = a_0 - a_1 + a_2 \dots - a_{n-1} + \sum_{s=0}^{\infty} \frac{(-1)^s}{2^{s+1}} [\Delta^s a_n] \quad (5.3.9)$$

Here  $\Delta$  is the *forward difference operator*, i.e.,

$$\begin{aligned}\Delta a_n &\equiv a_{n+1} - a_n \\ \Delta^2 a_n &\equiv a_{n+2} - 2a_{n+1} + a_n \\ \Delta^3 a_n &\equiv a_{n+3} - 3a_{n+2} + 3a_{n+1} - a_n \quad \text{etc.}\end{aligned}\tag{5.3.10}$$

Of course you don't actually do the infinite sum on the right-hand side of (5.3.9), but only the first, say,  $p$  terms, thus requiring the first  $p$  differences (5.3.10) obtained from the terms starting at  $a_n$ . There is an elegant and subtle implementation of Euler's transformation due to van Wijngaarden [6], discussed in full in a Webnote [7].

Euler's transformation is an example of a *linear* transformation: The partial sums of the transformed series are linear combinations of the partial sums of the original series. Euler's transformation and other linear transformations, while still important theoretically, have generally been superseded by newer *nonlinear* transformations that are considerably more powerful. As usual in numerical work, there is no free lunch: While the nonlinear transformations are more powerful, they are somewhat riskier than linear transformations in that they can occasionally fail spectacularly. But if you follow the guidance below, we think that you will never again resort to puny linear transformations.

The oldest example of a nonlinear sequence transformation is *Aitken's  $\Delta^2$ -process*. If  $s_n, s_{n+1}, s_{n+2}$  are three successive partial sums, then an improved estimate is

$$s'_n \equiv s_n - \frac{(s_{n+1} - s_n)^2}{s_{n+2} - 2s_{n+1} + s_n} = s_n - \frac{(\Delta s_n)^2}{\Delta^2 s_n}\tag{5.3.11}$$

The formula (5.3.11) is exact for a geometric series, which is one way of deriving it. If you form the sequence of  $s'_i$ 's, you can apply (5.3.11) a second time to *that* sequence, and so on. (In practice, this iteration will only rarely do much for you after the first stage.) Note that equation (5.3.11) should be computed as written; there exist algebraically equivalent forms that are much more susceptible to roundoff error.

Aitken's  $\Delta^2$ -process works only on linearly convergent sequences. Like Euler's transformation, it has also been superseded by algorithms such as the two we will now describe. After giving routines for these algorithms, we will supply some rules of thumb on when to use them.

The first "modern" nonlinear transformation was proposed by Shanks. An efficient recursive implementation was given by Wynn, called the  $\epsilon$  algorithm. Aitken's  $\Delta^2$ -process is a special case of the  $\epsilon$  algorithm, corresponding to using just three terms at a time. Although we will not give a derivation here, it is easy to state exactly what the  $\epsilon$  algorithm does: If you input the partial sums of a power series, the  $\epsilon$  algorithm returns the "diagonal" Padé approximants (§5.12) evaluated at the value of  $x$  used in the power series. (The coefficients in the approximant itself are not calculated.) That is, if  $[M/N]$  denotes the Padé approximant with a polynomial of degree  $M$  in the numerator and degree  $N$  in the denominator, the algorithm returns the numerical values of the approximants

$$[0, 0], \quad [1/0], \quad [1/1], \quad [2/1], \quad [2/2], \quad [3, 2], \quad [3, 3] \quad \dots\tag{5.3.12}$$

(The object `Epsalg` below is roughly equivalent to `pade` in §5.12 followed by an evaluation of the resulting rational function.)

In the object `Epsalg`, which is based on a routine in [1], you supply the sequence term by term and monitor the output for convergence in the calling program. Internally, the routine contains a check for division by zero and substitutes a large number

for the result. There are three conditions under which this check can be triggered: (i) Most likely, the algorithm has already converged, and should have been stopped earlier; (ii) there is an “accidental” zero term, and the program will recover; (iii) hardly ever in practice, the algorithm can actually fail because of a perverse combination of terms. Because (i) and (ii) are vastly more common than (iii), Epsalg hides the check condition and instead returns the last-known good estimate.

```
struct Epsalg {
    Convergence acceleration of a sequence by the  $\epsilon$  algorithm. Initialize by calling the constructor
    with arguments nmax, an upper bound on the number of terms to be summed, and epss, the
    desired accuracy. Then make successive calls to the function next, with argument the next
    partial sum of the sequence. The current estimate of the limit of the sequence is returned by
    next. The flag cnvgd is set when convergence is detected.
    VecDoub e;                                         Workspace.
    Int n,ncv;
    Bool cnvgd;
    Doub eps,small,big,lastval,lasteps;           Numbers near machine underflow and
                                                       overflow limits.
    Epsalg(Int nmax, Doub epss) : e(nmax), n(0), ncv(0),
    cnvgd(0), eps(epss), lastval(0.) {
        small = numeric_limits<Doub>::min()*10.0;
        big = numeric_limits<Doub>::max();
    }

    Doub next(Doub sum) {
        Doub diff,temp1,temp2,val;
        e[n]=sum;
        temp2=0.0;
        for (Int j=n; j>0; j--) {
            temp1=temp2;
            temp2=e[j-1];
            diff=e[j]-temp2;
            if (abs(diff) <= small)
                e[j-1]=big;
            else
                e[j-1]=temp1+1.0/diff;
        }
        n++;
        val = (n & 1) ? e[0] : e[1];             Cases of n even or odd.
        if (abs(val) > 0.01*big) val = lastval;
        lasteps = abs(val-lastval);
        if (lasteps > eps) ncv = 0;
        else ncv++;
        if (ncv >= 3) cnvgd = 1;
        return (lastval = val);
    }

};

series.h
```

The last few lines above implement a simple criterion for deciding whether the sequence has converged. For problems whose convergence is robust, you can simply put your calls to `next` inside a `while` loop like this:

```
Doub val, partialsum, eps=...;
Epsalg mysum(1000,eps);
while (! mysum.cnvgd) {
    partialsum = ...
    val = mysum.next(partialsum);
}
```

For more delicate cases, you can ignore the `cvgd` flag and just keep calling `next` until you are satisfied with the convergence.

A large class of modern nonlinear transformations can be derived by using the concept of a *model sequence*. The idea is to choose a “simple” sequence that approximates the asymptotic form of the given sequence and construct a transformation that sums the model sequence exactly. Presumably the transformation will work well for other sequences with similar asymptotic properties. For example, a geometric series provides the model sequence for Aitken’s  $\Delta^2$ -process.

The *Levin transformation* is probably the best single sequence acceleration method currently known. It is based on approximating a sequence asymptotically by an expression of the form

$$s_n = s + \omega_n \sum_{j=0}^{k-1} \frac{c_j}{(n + \beta)^j} \quad (5.3.13)$$

Here  $\omega_n$  is the dominant term in the remainder of the sequence:

$$s_n - s = \omega_n [c + O(n^{-1})], \quad n \rightarrow \infty \quad (5.3.14)$$

The constants  $c_j$  are arbitrary, and  $\beta$  is a parameter that is restricted to be positive. Levin showed that for a model sequence of the form (5.3.13), the following transformation gives the exact value of the series:

$$s = \frac{\sum_{j=0}^k (-1)^j \binom{k}{j} \frac{(\beta + n + j)^{k-1}}{(\beta + n + k)^{k-1}} \frac{s_{n+j}}{\omega_{n+j}}}{\sum_{j=0}^k (-1)^j \binom{k}{j} \frac{(\beta + n + j)^{k-1}}{(\beta + n + k)^{k-1}} \frac{1}{\omega_{n+j}}} \quad (5.3.15)$$

(The common factor  $(\beta + n + k)^{k-1}$  in the numerator and denominator reduces the chances of overflow for large  $k$ .) A derivation of equation (5.3.15) is given in a Webnote [4].

The numerator and denominator in (5.3.15) are not computed as written. Instead, they can be computed efficiently from a single recurrence relation with different starting values (see [1] for a derivation):

$$D_{k+1}^n(\beta) = D_k^{n+1}(\beta) - \frac{(\beta + n)(\beta + n + k)^{k-1}}{(\beta + n + k + 1)^k} D_k^n(\beta) \quad (5.3.16)$$

The starting values are

$$D_0^n(\beta) = \begin{cases} s_n / \omega_n, & \text{numerator} \\ 1 / \omega_n, & \text{denominator} \end{cases} \quad (5.3.17)$$

Although  $D_k^n$  is a two-dimensional object, the recurrence can be coded in a one-dimensional array proceeding up the counterdiagonal  $n + k = \text{constant}$ .

The choice (5.3.14) doesn’t determine  $\omega_n$  uniquely, but if you have analytic information about your series, this is where you can make use of it. Usually you won’t

be so lucky, in which case you can make a choice based on heuristics. For example, the remainder in an alternating series is approximately half the first neglected term, which suggests setting  $\omega_n$  equal to  $a_n$  or  $a_{n+1}$ . These are called the Levin  $t$  and  $d$  transformations, respectively. Similarly, the remainder for a geometric series is the difference between the partial sum (5.3.7) and its limit  $1/(1-x)$ . This can be written as  $a_n a_{n+1} / (a_n - a_{n+1})$ , which defines the Levin  $v$  transformation. The most popular choice comes from approximating the remainder in the  $\zeta$  function (5.3.8) by an integral:

$$\sum_{k=n+1}^{\infty} \frac{1}{k^x} \approx \int_{n+1}^{\infty} \frac{dk}{k^x} = \frac{(n+1)^{1-x}}{x-1} = \frac{(n+1)a_{n+1}}{x-1} \quad (5.3.18)$$

This motivates the choice  $(n + \beta)a_n$  (Levin  $u$  transformation), where  $\beta$  is usually chosen to be 1. To summarize:

$$\omega_n = \begin{cases} (\beta + n)a_n, & u \text{ transformation} \\ a_n, & t \text{ transformation} \\ a_{n+1}, & d \text{ transformation (modified } t \text{ transformation)} \\ \frac{a_n a_{n+1}}{a_n - a_{n+1}}, & v \text{ transformation} \end{cases} \quad (5.3.19)$$

For sequences that are not partial sums, so that the individual  $a_n$ 's are not defined, replace  $a_n$  by  $\Delta s_{n-1}$  in (5.3.19).

Here is the routine for Levin's transformation, also based on the routine in [1]:

```
series.h
struct Levin {
    // Convergence acceleration of a sequence by the Levin transformation. Initialize by calling the
    // constructor with arguments nmax, an upper bound on the number of terms to be summed, and
    // epss, the desired accuracy. Then make successive calls to the function next, which returns
    // the current estimate of the limit of the sequence. The flag cnvgd is set when convergence is
    // detected.
    VecDoub numer,denom;           // Numerator and denominator computed via (5.3.16).
    Int n,ncv;
    Bool cnvgd;
    Doub small,big;              // Numbers near machine underflow and overflow limits.
    Doub eps,lastval,lasteps;

    Levin(Int nmax, Doub epss) : numer(nmax), denom(nmax), n(0), ncv(0),
        cnvgd(0), eps(epss), lastval(0.) {
        small=numeric_limits<Doub>::min()*10.0;
        big=numeric_limits<Doub>::max();
    }

    Doub next(Doub sum, Doub omega, Doub beta=1.) {
        // Arguments: sum, the nth partial sum of the sequence; omega, the nth remainder estimate
        // omega_n, usually from (5.3.19); and the parameter beta, which should usually be set to 1, but
        // sometimes 0.5 works better. The current estimate of the limit of the sequence is returned.

        Int j;
        Doub fact,ratio,term,val;
        term=1.0/(beta+n);
        denom[n]=term/omega;
        numer[n]=sum*denom[n];
        if (n > 0) {
            ratio=(beta+n-1)*term;
            for (j=1;j<=n;j++) {

```

```

        fact=(n-j+beta)*term;
        numer[n-j]=numer[n-j+1]-fact*numer[n-j];
        denom[n-j]=denom[n-j+1]-fact*denom[n-j];
        term=term*ratio;
    }
}
n++;
val = abs(denom[0]) < small ? lastval : numer[0]/denom[0];
lasteps = abs(val-lastval);
if (lasteps <= eps) ncv++;
if (ncv >= 2) cnvgd = 1;
return (lastval = val);
}
};

```

You can use, or not use, the `cnvgd` flag exactly as previously discussed for `Epsalg`.

An alternative to the model sequence method of deriving sequence transformations is to use extrapolation of a polynomial or rational function approximation to a series, e.g., as in Wynn's  $\rho$  algorithm [1]. Since none of these methods generally beats the two we have given, we won't say any more about them.

### 5.3.3 Practical Hints and an Example

There is no general theoretical understanding of nonlinear sequence transformations. Accordingly, most of the practical advice is based on numerical experiments [5]. You might have thought that summing a wildly divergent series is the hardest problem for a sequence transformation. However, the difficulty of a problem depends more on whether the terms are all of the same sign or whether the signs alternate, rather than whether the sequence actually converges or not. In particular, logarithmically convergent series with terms all of the same sign are generally the most difficult to sum. Even the best acceleration methods are corrupted by rounding errors when accelerating logarithmic convergence. You should always use double precision and be prepared for some loss of significant digits. Typically one observes convergence up to some optimum number of terms, and then a loss of significant digits if one tries to go further. Moreover, there is no single algorithm that can accelerate every logarithmically convergent sequence. Nevertheless, there are some good rules of thumb.

First, note that among divergent series it is useful to separate out asymptotic series, where the terms first decrease before increasing, as a separate class from other divergent series, e.g., power series outside their radius of convergence. For alternating series, whether convergent, asymptotic, or divergent power series, Levin's  $u$  transformation is almost always the best choice. For monotonic linearly convergent or monotonic divergent power series, the  $\epsilon$  algorithm typically is the first choice, but the  $u$  transformation often does a reasonable job. For logarithmic convergence, the  $u$  transformation is clearly the best. (The  $\epsilon$  algorithm fails completely.) For series with irregular signs or other nonstandard features, typically the  $\epsilon$  algorithm is relatively robust, often succeeding where other algorithms fail. Finally, for monotonic asymptotic series, such as (6.3.11) for  $Ei(x)$ , there is nothing better than direct summation without acceleration.

The  $v$  and  $t$  transformations are almost as good as the  $u$  transformation, except that the  $t$  transformation typically fails for logarithmic convergence.

If you have only a few numerical terms of some sequence and no theoretical insight, blindly applying a convergence accelerator can be dangerous. The algorithm

can sometimes display “convergence” that is only apparent, not real. The remedy is to try two different transformations as a check.

Since convergence acceleration is so much more difficult for a series of positive terms than for an alternating series, occasionally it is useful to convert a series of positive terms into an alternating series. Van Wijngaarden has given a transformation for accomplishing this [6]:

$$\sum_{r=1}^{\infty} v_r = \sum_{r=1}^{\infty} (-1)^{r-1} w_r \quad (5.3.20)$$

where

$$w_r \equiv v_r + 2v_{2r} + 4v_{4r} + 8v_{8r} + \dots \quad (5.3.21)$$

Equations (5.3.20) and (5.3.21) replace a simple sum by a two-dimensional sum, each term in (5.3.20) being itself an infinite sum (5.3.21). This may seem a strange way to save on work! Since, however, the indices in (5.3.21) increase tremendously rapidly, as powers of 2, it often requires only a few terms to converge (5.3.21) to extraordinary accuracy. You do, however, need to be able to compute the  $v_r$ 's efficiently for “random” values  $r$ . The standard “updating” tricks for sequential  $r$ 's, mentioned above following equation (5.3.1), can't be used.

Once you've generated the alternating series by Van Wijngaarden's transformation, the Levin  $d$  transformation is particularly effective at summing the series [8]. This strategy is most useful for linearly convergent series with  $\rho$  close to 1. For logarithmically convergent series, even the transformed series (5.3.21) is often too slowly convergent to be useful numerically.

As an example of how to call the routines `Epsalg` or `Levin`, consider the problem of evaluating the integral

$$I = \int_0^{\infty} \frac{x}{1+x^2} J_0(x) dx = K_0(1) = 0.4210244382\dots \quad (5.3.22)$$

Standard quadrature methods such as `qromo` fail because the integrand has a long oscillatory tail, giving alternating positive and negative contributions that tend to cancel. A good way of evaluating such an integral is to split it into a sum of integrals between successive zeros of  $J_0(x)$ :

$$I = \int_0^{\infty} f(x) dx = \sum_{j=0}^{\infty} I_j \quad (5.3.23)$$

where

$$I_j = \int_{x_{j-1}}^{x_j} f(x) dx, \quad f(x_j) = 0, \quad j = 0, 1, \dots \quad (5.3.24)$$

We take  $x_{-1}$  equal to the lower limit of the integral, zero in this example. The idea is to evaluate the relatively simple integrals  $I_j$  by `qromb` or Gaussian quadrature, and then accelerate the convergence of the series (5.3.23), since we expect the contributions to alternate in sign. For the example (5.3.22), we don't even need accurate values of the zeros of  $J_0(x)$ . It is good enough to take  $x_j = (j+1)\pi$ , which is asymptotically correct. Here is the code:

```
levex.h Doub func(const Doub x)
Integrand for (5.3.22).
{
    if (x == 0.0)
        return 0.0;
    else {
        Bessel bess;
        return x*bess.jnu(0.0,x)/(1.0+x*x);
    }
}

Int main_levex(void)
This sample program shows how to use the Levin  $u$  transformation to evaluate an oscillatory
integral, equation (5.3.22).
{
    const Doub PI=3.141592653589793;
    Int nterm=12;
    Doub beta=1.0,a=0.0,b=0.0,sum=0.0;
    Levin series(100,0.0);
    cout << setw(5) << "N" << setw(19) << "Sum (direct)" << setw(21)
        << "Sum (Levin)" << endl;
    for (Int n=0; n<nterm; n++) {
        b+=PI;
        Doub s=qromb(func,a,b,1.e-8);
        a=b;
        sum+=s;
        Doub omega=(beta+n)*s;      Use  $u$  transformation.
        Doub ans=series.next(sum,omega,beta);
        cout << setw(5) << n << fixed << setprecision(14) << setw(21)
            << sum << setw(21) << ans << endl;
    }
    return 0;
}
```

Setting  $\text{eps}$  to  $1 \times 10^{-8}$  in `qromb`, we get 9 significant digits with about 200 function evaluations by  $n = 8$ . Replacing `qromb` with a Gaussian quadrature routine cuts the number of function evaluations in half. Note that  $n = 8$  corresponds to an upper limit in the integral of  $9\pi$ , where the amplitude of the integrand is still of order  $10^{-2}$ . This shows the remarkable power of convergence acceleration. (For more on oscillatory integrals, see §13.9.)

#### CITED REFERENCES AND FURTHER READING:

- Weniger, E.J. 1989, “Nonlinear Sequence Transformations for the Acceleration of Convergence and the Summation of Divergent Series,” *Computer Physics Reports*, vol. 10, pp. 189–371.[1]
- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, §3.6.[2]
- Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), §2.3.[3]
- Numerical Recipes Software 2007, “Derivation of the Levin Transformation,” *Numerical Recipes Webnote No. 6*, at <http://www.nr.com/webnotes?6> [4]
- Smith, D.A., and Ford, W.F. 1982, “Numerical Comparisons of Nonlinear Convergence Accelerators,” *Mathematics of Computation*, vol. 38, pp. 481–499.[5]
- Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 13 [van Wijngaarden’s transformations].[6]

- Numerical Recipes Software 2007, "Implementation of the Euler Transformation," *Numerical Recipes Webnote No. 5*, at <http://www.nr.com/webnotes?5> [7]
- Jentschura, U.D., Mohr, P.J., Soff, G., and Weniger, E.J. 1999, "Convergence Acceleration via Combined Nonlinear-Condensation Transformations," *Computer Physics Communications*, vol. 116, pp. 28–54.[8]
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), Chapter 3.

## 5.4 Recurrence Relations and Clenshaw's Recurrence Formula

Many useful functions satisfy recurrence relations, e.g.,

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \quad (5.4.1)$$

$$J_{n+1}(x) = \frac{2n}{x} J_n(x) - J_{n-1}(x) \quad (5.4.2)$$

$$nE_{n+1}(x) = e^{-x} - xE_n(x) \quad (5.4.3)$$

$$\cos n\theta = 2 \cos \theta \cos(n-1)\theta - \cos(n-2)\theta \quad (5.4.4)$$

$$\sin n\theta = 2 \cos \theta \sin(n-1)\theta - \sin(n-2)\theta \quad (5.4.5)$$

where the first three functions are Legendre polynomials, Bessel functions of the first kind, and exponential integrals, respectively. (For notation see [1].) These relations are useful for extending computational methods from two successive values of  $n$  to other values, either larger or smaller.

Equations (5.4.4) and (5.4.5) motivate us to say a few words about trigonometric functions. If your program's running time is dominated by evaluating trigonometric functions, you are probably doing something wrong. Trig functions whose arguments form a linear sequence  $\theta = \theta_0 + n\delta$ ,  $n = 0, 1, 2, \dots$ , are efficiently calculated by the recurrence

$$\begin{aligned} \cos(\theta + \delta) &= \cos \theta - [\alpha \cos \theta + \beta \sin \theta] \\ \sin(\theta + \delta) &= \sin \theta - [\alpha \sin \theta - \beta \cos \theta] \end{aligned} \quad (5.4.6)$$

where  $\alpha$  and  $\beta$  are the precomputed coefficients

$$\alpha \equiv 2 \sin^2\left(\frac{\delta}{2}\right) \quad \beta \equiv \sin \delta \quad (5.4.7)$$

The reason for doing things this way, rather than with the standard (and equivalent) identities for sums of angles, is that here  $\alpha$  and  $\beta$  do not lose significance if the incremental  $\delta$  is small. Likewise, the adds in equation (5.4.6) should be done in the order indicated by the square brackets. We will use (5.4.6) repeatedly in Chapter 12, when we deal with Fourier transforms.

Another trick, occasionally useful, is to note that both  $\sin \theta$  and  $\cos \theta$  can be calculated via a single call to  $\tan$ :

$$t \equiv \tan\left(\frac{\theta}{2}\right) \quad \cos \theta = \frac{1-t^2}{1+t^2} \quad \sin \theta = \frac{2t}{1+t^2} \quad (5.4.8)$$

The cost of getting both  $\sin$  and  $\cos$ , if you need them, is thus the cost of  $\tan$  plus 2 multiplies, 2 divides, and 2 adds. On machines with slow trig functions, this can be a savings. However, note that special treatment is required if  $\theta \rightarrow \pm\pi$ . And also note that many modern machines have *very fast* trig functions; so you should not assume that equation (5.4.8) is faster without testing.

### 5.4.1 Stability of Recurrences

You need to be aware that recurrence relations are not necessarily *stable* against roundoff error in the direction that you propose to go (either increasing  $n$  or decreasing  $n$ ). A three-term linear recurrence relation

$$y_{n+1} + a_n y_n + b_n y_{n-1} = 0, \quad n = 1, 2, \dots \quad (5.4.9)$$

has two linearly independent solutions,  $f_n$  and  $g_n$ , say. Only one of these corresponds to the sequence of functions  $f_n$  that you are trying to generate. The other one,  $g_n$ , may be exponentially growing in the direction that you want to go, or exponentially damped, or exponentially neutral (growing or dying as some power law, for example). If it is exponentially growing, then the recurrence relation is of little or no practical use in that direction. This is the case, e.g., for (5.4.2) in the direction of increasing  $n$ , when  $x < n$ . You cannot generate Bessel functions of high  $n$  by forward recurrence on (5.4.2).

To state things a bit more formally, if

$$f_n/g_n \rightarrow 0 \quad \text{as} \quad n \rightarrow \infty \quad (5.4.10)$$

then  $f_n$  is called the *minimal* solution of the recurrence relation (5.4.9). Nonminimal solutions like  $g_n$  are called *dominant* solutions. The minimal solution is unique, if it exists, but dominant solutions are not — you can add an arbitrary multiple of  $f_n$  to a given  $g_n$ . You can evaluate any dominant solution by forward recurrence, *but not the minimal solution*. (Unfortunately it is sometimes the one you want.)

Abramowitz and Stegun (in their Introduction!) [1] give a list of recurrences that are stable in the increasing or decreasing direction. That list does not contain all possible formulas, of course. Given a recurrence relation for some function  $f_n(x)$ , you can test it yourself with about five minutes of (human) labor: For a fixed  $x$  in your range of interest, start the recurrence not with true values of  $f_j(x)$  and  $f_{j+1}(x)$ , but (first) with the values 1 and 0, respectively, and then (second) with 0 and 1, respectively. Generate 10 or 20 terms of the recursive sequences in the direction that you want to go (increasing or decreasing from  $j$ ), for each of the two starting conditions. Look at the differences between the corresponding members of the two sequences. If the differences stay of order unity (absolute value less than 10, say), then the recurrence is stable. If they increase slowly, then the recurrence may be mildly unstable but quite tolerably so. If they increase catastrophically, then there is an exponentially growing solution of the recurrence. If you know that the function that you want actually corresponds to the growing solution, then you can keep the recurrence formula anyway (e.g., the case of the Bessel function  $Y_n(x)$  for increasing  $n$ ; see §6.5). If you don't know which solution your function corresponds to, you must at this point reject the recurrence formula. Notice that you can do this test *before* you go to the trouble of finding a numerical method for computing the two

starting functions  $f_j(x)$  and  $f_{j+1}(x)$ : Stability is a property of the recurrence, not of the starting values.

An alternative heuristic procedure for testing stability is to replace the recurrence relation by a similar one that is linear with constant coefficients. For example, the relation (5.4.2) becomes

$$y_{n+1} - 2\gamma y_n + y_{n-1} = 0 \quad (5.4.11)$$

where  $\gamma \equiv n/x$  is treated as a constant. You solve such recurrence relations by trying solutions of the form  $y_n = a^n$ . Substituting into the above recurrence gives

$$a^2 - 2\gamma a + 1 = 0 \quad \text{or} \quad a = \gamma \pm \sqrt{\gamma^2 - 1} \quad (5.4.12)$$

The recurrence is stable if  $|a| \leq 1$  for all solutions  $a$ . This holds (as you can verify) if  $|\gamma| \leq 1$  or  $n \leq x$ . The recurrence (5.4.2) thus cannot be used, starting with  $J_0(x)$  and  $J_1(x)$ , to compute  $J_n(x)$  for large  $n$ .

Possibly you would at this point like the security of some real theorems on this subject (although we ourselves always follow one of the heuristic procedures). Here are two theorems, due to Perron [2]:

*Theorem A.* If in (5.4.9)  $a_n \sim an^\alpha$ ,  $b_n \sim bn^\beta$  as  $n \rightarrow \infty$ , and  $\beta < 2\alpha$ , then

$$g_{n+1}/g_n \sim -an^\alpha, \quad f_{n+1}/f_n \sim -(b/a)n^{\beta-\alpha} \quad (5.4.13)$$

and  $f_n$  is the minimal solution to (5.4.9).

*Theorem B.* Under the same conditions as Theorem A, but with  $\beta = 2\alpha$ , consider the *characteristic polynomial*

$$t^2 + at + b = 0 \quad (5.4.14)$$

If the roots  $t_1$  and  $t_2$  of (5.4.14) have distinct moduli,  $|t_1| > |t_2|$  say, then

$$g_{n+1}/g_n \sim t_1 n^\alpha, \quad f_{n+1}/f_n \sim t_2 n^\alpha \quad (5.4.15)$$

and  $f_n$  is again the minimal solution to (5.4.9). Cases other than those in these two theorems are inconclusive for the existence of minimal solutions. (For more on the stability of recurrences, see [3].)

How do you proceed if the solution that you desire *is* the minimal solution? The answer lies in that old aphorism, that every cloud has a silver lining: If a recurrence relation is catastrophically unstable in one direction, then that (undesired) solution will decrease very rapidly in the reverse direction. This means that you can start with *any* seed values for the consecutive  $f_j$  and  $f_{j+1}$  and (when you have gone enough steps in the stable direction) you will converge to the sequence of functions that you want, times an unknown normalization factor. If there is some other way to normalize the sequence (e.g., by a formula for the sum of the  $f_n$ 's), then this can be a practical means of function evaluation. The method is called *Miller's algorithm*. An example often given [1,4] uses equation (5.4.2) in just this way, along with the normalization formula

$$1 = J_0(x) + 2J_2(x) + 2J_4(x) + 2J_6(x) + \dots \quad (5.4.16)$$

Incidentally, there is an important relation between three-term recurrence relations and *continued fractions*. Rewrite the recurrence relation (5.4.9) as

$$\frac{y_n}{y_{n-1}} = -\frac{b_n}{a_n + y_{n+1}/y_n} \quad (5.4.17)$$

Iterating this equation, starting with  $n$ , gives

$$\frac{y_n}{y_{n-1}} = -\frac{b_n}{a_n - \frac{b_{n+1}}{a_{n+1} - \dots}} \quad (5.4.18)$$

*Pincherle's theorem* [2] tells us that (5.4.18) converges if and only if (5.4.9) has a minimal solution  $f_n$ , in which case it converges to  $f_n/f_{n-1}$ . This result, usually for the case  $n = 1$  and combined with some way to determine  $f_0$ , underlies many of the practical methods for computing special functions that we give in the next chapter.

### 5.4.2 Clenshaw's Recurrence Formula

*Clenshaw's recurrence formula* [5] is an elegant and efficient way to evaluate a sum of coefficients times functions that obey a recurrence formula, e.g.,

$$f(\theta) = \sum_{k=0}^N c_k \cos k\theta \quad \text{or} \quad f(x) = \sum_{k=0}^N c_k P_k(x)$$

Here is how it works: Suppose that the desired sum is

$$f(x) = \sum_{k=0}^N c_k F_k(x) \quad (5.4.19)$$

and that  $F_k$  obeys the recurrence relation

$$F_{n+1}(x) = \alpha(n, x)F_n(x) + \beta(n, x)F_{n-1}(x) \quad (5.4.20)$$

for some functions  $\alpha(n, x)$  and  $\beta(n, x)$ . Now define the quantities  $y_k$  ( $k = N, N-1, \dots, 1$ ) by the recurrence

$$\begin{aligned} y_{N+2} &= y_{N+1} = 0 \\ y_k &= \alpha(k, x)y_{k+1} + \beta(k+1, x)y_{k+2} + c_k \quad (k = N, N-1, \dots, 1) \end{aligned} \quad (5.4.21)$$

If you solve equation (5.4.21) for  $c_k$  on the left, and then write out explicitly the sum (5.4.19), it will look (in part) like this:

$$\begin{aligned} f(x) &= \dots \\ &\quad + [y_8 - \alpha(8, x)y_9 - \beta(9, x)y_{10}]F_8(x) \\ &\quad + [y_7 - \alpha(7, x)y_8 - \beta(8, x)y_9]F_7(x) \\ &\quad + [y_6 - \alpha(6, x)y_7 - \beta(7, x)y_8]F_6(x) \\ &\quad + [y_5 - \alpha(5, x)y_6 - \beta(6, x)y_7]F_5(x) \\ &\quad + \dots \\ &\quad + [y_2 - \alpha(2, x)y_3 - \beta(3, x)y_4]F_2(x) \\ &\quad + [y_1 - \alpha(1, x)y_2 - \beta(2, x)y_3]F_1(x) \\ &\quad + [c_0 + \beta(1, x)y_2 - \beta(1, x)y_2]F_0(x) \end{aligned} \quad (5.4.22)$$

Notice that we have added and subtracted  $\beta(1, x)y_2$  in the last line. If you examine the terms containing a factor of  $y_8$  in (5.4.22), you will find that they sum to zero as a consequence of the recurrence relation (5.4.20); similarly for all the other  $y_k$ 's down through  $y_2$ . The only surviving terms in (5.4.22) are

$$f(x) = \beta(1, x)F_0(x)y_2 + F_1(x)y_1 + F_0(x)c_0 \quad (5.4.23)$$

Equations (5.4.21) and (5.4.23) are *Clenshaw's recurrence formula* for doing the sum (5.4.19): You make one pass down through the  $y_k$ 's using (5.4.21); when you have reached  $y_2$  and  $y_1$ , you apply (5.4.23) to get the desired answer.

Clenshaw's recurrence as written above incorporates the coefficients  $c_k$  in a downward order, with  $k$  decreasing. At each stage, the effect of all previous  $c_k$ 's is "remembered" as two coefficients that multiply the functions  $F_{k+1}$  and  $F_k$  (ultimately  $F_0$  and  $F_1$ ). If the functions  $F_k$  are small when  $k$  is large, *and* if the coefficients  $c_k$  are small when  $k$  is *small*, then the sum can be dominated by small  $F_k$ 's. In this case, the remembered coefficients will involve a delicate cancellation and there can be a catastrophic loss of significance. An example would be to sum the trivial series

$$J_{15}(1) = 0 \times J_0(1) + 0 \times J_1(1) + \dots + 0 \times J_{14}(1) + 1 \times J_{15}(1) \quad (5.4.24)$$

Here  $J_{15}$ , which is tiny, ends up represented as a canceling linear combination of  $J_0$  and  $J_1$ , which are of order unity.

The solution in such cases is to use an alternative Clenshaw recurrence that incorporates the  $c_k$ 's in an upward direction. The relevant equations are

$$y_{-2} = y_{-1} = 0 \quad (5.4.25)$$

$$y_k = \frac{1}{\beta(k+1, x)}[y_{k-2} - \alpha(k, x)y_{k-1} - c_k], \quad k = 0, 1, \dots, N-1 \quad (5.4.26)$$

$$f(x) = c_N F_N(x) - \beta(N, x)F_{N-1}(x)y_{N-1} - F_N(x)y_{N-2} \quad (5.4.27)$$

The rare case where equations (5.4.25) – (5.4.27) should be used instead of equations (5.4.21) and (5.4.23) can be detected automatically by testing whether the operands in the first sum in (5.4.23) are opposite in sign and nearly equal in magnitude. Other than in this special case, Clenshaw's recurrence is always stable, independent of whether the recurrence for the functions  $F_k$  is stable in the upward or downward direction.

### 5.4.3 Parallel Evaluation of Linear Recurrence Relations

When desirable, linear recurrence relations can be evaluated with a lot of parallelism. Consider the general first-order linear recurrence relation

$$u_j = a_j + b_{j-1}u_{j-1}, \quad j = 2, 3, \dots, n \quad (5.4.28)$$

with initial value  $u_1 = a_1$ . To parallelize the recurrence, we can employ the powerful general strategy of *recursive doubling*. Write down equation (5.4.28) for  $2j$  and for  $2j-1$ :

$$\begin{aligned} u_{2j} &= a_{2j} + b_{2j-1}u_{2j-1} \\ u_{2j-1} &= a_{2j-1} + b_{2j-2}u_{2j-2} \end{aligned} \quad (5.4.29)$$

Substitute the second of these equations into the first to eliminate  $u_{2j-1}$  and get

$$u_{2j} = (a_{2j} + a_{2j-1}b_{2j-1}) + (b_{2j-2}b_{2j-1})u_{2j-2} \quad (5.4.30)$$

This is a new recurrence of the same form as (5.4.28) but over only the even  $u_j$ , and hence involving only  $n/2$  terms. Clearly we can continue this process recursively, halving the number of terms in the recurrence at each stage, until we are left with a recurrence of length 1 or 2 that we can do explicitly. Each time we finish a subpart of the recursion, we fill in the odd terms in the recurrence, using the second equation in (5.4.29). In practice, it's even easier than it sounds. The total number of operations is the same as for serial evaluation, but they are done in about  $\log_2 n$  parallel steps.

There is a variant of recursive doubling, called *cyclic reduction*, that can be implemented with a straightforward iteration loop instead of a recursive procedure [6]. Here we start by writing down the recurrence (5.4.28) for all adjacent terms  $u_j$  and  $u_{j-1}$  (not just the even ones, as before). Eliminating  $u_{j-1}$ , just as in equation (5.4.30), gives

$$u_j = (a_j + a_{j-1}b_{j-1}) + (b_{j-2}b_{j-1})u_{j-2} \quad (5.4.31)$$

which is a first-order recurrence with new coefficients  $a'_j$  and  $b'_j$ . Repeating this process gives successive formulas for  $u_j$  in terms of  $u_{j-2}, u_{j-4}, u_{j-8}, \dots$ . The procedure terminates when we reach  $u_{j-n}$  (for  $n$  a power of 2), which is zero for all  $j$ . Thus the last step gives  $u_j$  equal to the last set of  $a'_j$ 's.

In cyclic reduction, the length of the vector  $u_j$  that is updated at each stage does not decrease by a factor of 2 at each stage, but rather only decreases from  $\sim n$  to  $\sim n/2$  during all  $\log_2 n$  stages. Thus the total number of operations carried out is  $O(n \log n)$ , as opposed to  $O(n)$  for recursive doubling. Whether this is important depends on the details of the computer's architecture.

Second-order recurrence relations can also be parallelized. Consider the second-order recurrence relation

$$y_j = a_j + b_{j-2}y_{j-1} + c_{j-2}y_{j-2}, \quad j = 3, 4, \dots, n \quad (5.4.32)$$

with initial values

$$y_1 = a_1, \quad y_2 = a_2 \quad (5.4.33)$$

With this numbering scheme, you supply coefficients  $a_1, \dots, a_n, b_1, \dots, b_{n-2}$ , and  $c_1, \dots, c_{n-2}$ . Rewrite the recurrence relation in the form [6]

$$\begin{pmatrix} y_j \\ y_{j+1} \end{pmatrix} = \begin{pmatrix} 0 \\ a_{j+1} \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ c_{j-1} & b_{j-1} \end{pmatrix} \begin{pmatrix} y_{j-1} \\ y_j \end{pmatrix}, \quad j = 2, \dots, n-1 \quad (5.4.34)$$

that is,

$$\mathbf{u}_j = \mathbf{a}_j + \mathbf{b}_{j-1} \cdot \mathbf{u}_{j-1}, \quad j = 2, \dots, n-1 \quad (5.4.35)$$

where

$$\mathbf{u}_j = \begin{pmatrix} y_j \\ y_{j+1} \end{pmatrix}, \quad \mathbf{a}_j = \begin{pmatrix} 0 \\ a_{j+1} \end{pmatrix}, \quad \mathbf{b}_{j-1} = \begin{pmatrix} 0 & 1 \\ c_{j-1} & b_{j-1} \end{pmatrix} \quad (5.4.36)$$

and

$$\mathbf{u}_1 = \mathbf{a}_1 = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \quad (5.4.37)$$

This is a first-order recurrence relation for the vectors  $\mathbf{u}_j$  and can be solved by either of the algorithms described above. The only difference is that the multiplications are matrix multiplications with the  $2 \times 2$  matrices  $\mathbf{b}_j$ . After the first recursive call, the zeros in  $\mathbf{a}$  and  $\mathbf{b}$  are lost, so we have to write the routine for general two-dimensional vectors and matrices. Note that this algorithm does not avoid the potential instability problems associated with second-order recurrences that were discussed in §5.4.1. Also note that the algorithm generalizes in the obvious way to higher-order recurrences: An  $n$ th-order recurrence can be written as a first-order recurrence involving vectors and matrices of dimension  $n$ .

#### CITED REFERENCES AND FURTHER READING:

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, pp. xiii, 697.[1]
- Gautschi, W. 1967, "Computational Aspects of Three-Term Recurrence Relations," *SIAM Review*, vol. 9, pp. 24–82.[2]
- Lakshmikantham, V., and Trigiante, D. 1988, *Theory of Difference Equations: Numerical Methods and Applications* (San Diego: Academic Press).[3]
- Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), pp. 20ff.[4]
- Clenshaw, C.W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory (London: H.M. Stationery Office).[5]
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §4.4.3, p. 111.
- Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), p. 76.
- Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2: Architecture, Programming, and Algorithms* (Bristol and Philadelphia: Adam Hilger), §5.2.4 and §5.4.2.[6]

## 5.5 Complex Arithmetic

Since C++ has a built-in class `complex`, you can generally let the compiler and the class library take care of complex arithmetic for you. Generally, but not always. For a program with only a small number of complex operations, you may want to code these yourself, in-line. Or, you may find that your compiler is not up to snuff: It is disconcertingly common to encounter complex operations that produce overflows or underflows when both the complex operands and the complex result are perfectly representable. This occurs, we think, because software companies mistake the implementation of complex arithmetic for a completely trivial task, not requiring any particular finesse.

Actually, complex arithmetic is not *quite* trivial. Addition and subtraction are done in the obvious way, performing the operation separately on the real and imaginary parts of the operands. Multiplication can also be done in the obvious way, with four multiplications, one addition, and one subtraction:

$$(a + ib)(c + id) = (ac - bd) + i(bc + ad) \quad (5.5.1)$$

(the addition sign before the  $i$  doesn't count; it just separates the real and imaginary parts notationally). But it is sometimes faster to multiply via

$$(a + ib)(c + id) = (ac - bd) + i[(a + b)(c + d) - ac - bd] \quad (5.5.2)$$

which has only three multiplications ( $ac$ ,  $bd$ ,  $(a+b)(c+d)$ ), plus two additions and three subtractions. The total operations count is higher by two, but multiplication is a slow operation on some machines.

While it is true that intermediate results in equations (5.5.1) and (5.5.2) can overflow even when the final result is representable, this happens only when the final

answer is on the edge of representability. Not so for the complex modulus, if you or your compiler is misguided enough to compute it as

$$|a + ib| = \sqrt{a^2 + b^2} \quad (\text{bad!}) \quad (5.5.3)$$

whose intermediate result will overflow if either  $a$  or  $b$  is as large as the square root of the largest representable number (e.g.,  $10^{19}$  as compared to  $10^{38}$ ). The right way to do the calculation is

$$|a + ib| = \begin{cases} |a|\sqrt{1 + (b/a)^2} & |a| \geq |b| \\ |b|\sqrt{1 + (a/b)^2} & |a| < |b| \end{cases} \quad (5.5.4)$$

Complex division should use a similar trick to prevent avoidable overflow, underflow, or loss of precision:

$$\frac{a + ib}{c + id} = \begin{cases} \frac{[a + b(d/c)] + i[b - a(d/c)]}{c + d(d/c)} & |c| \geq |d| \\ \frac{[a(c/d) + b] + i[b(c/d) - a]}{c(c/d) + d} & |c| < |d| \end{cases} \quad (5.5.5)$$

Of course you should calculate repeated subexpressions, like  $c/d$  or  $d/c$ , only once.

Complex square root is even more complicated, since we must both guard intermediate results and also enforce a chosen branch cut (here taken to be the negative real axis). To take the square root of  $c + id$ , first compute

$$w \equiv \begin{cases} 0 & c = d = 0 \\ \sqrt{|c|} \sqrt{\frac{1 + \sqrt{1 + (d/c)^2}}{2}} & |c| \geq |d| \\ \sqrt{|d|} \sqrt{\frac{|c/d| + \sqrt{1 + (c/d)^2}}{2}} & |c| < |d| \end{cases} \quad (5.5.6)$$

Then the answer is

$$\sqrt{c + id} = \begin{cases} 0 & w = 0 \\ w + i \left( \frac{d}{2w} \right) & w \neq 0, c \geq 0 \\ \frac{|d|}{2w} + iw & w \neq 0, c < 0, d \geq 0 \\ \frac{|d|}{2w} - iw & w \neq 0, c < 0, d < 0 \end{cases} \quad (5.5.7)$$

#### CITED REFERENCES AND FURTHER READING:

Midy, P., and Yakovlev, Y. 1991, "Computing Some Elementary Functions of a Complex Variable," *Mathematics and Computers in Simulation*, vol. 33, pp. 33–49.

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley) [see solutions to exercises 4.2.1.16 and 4.6.4.41].

## 5.6 Quadratic and Cubic Equations

The roots of simple algebraic equations can be viewed as being functions of the equations' coefficients. We are taught these functions in elementary algebra. Yet, surprisingly many people don't know the right way to solve a quadratic equation with two real roots, or to obtain the roots of a cubic equation.

There are two ways to write the solution of the *quadratic equation*

$$ax^2 + bx + c = 0 \quad (5.6.1)$$

with real coefficients  $a, b, c$ , namely

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (5.6.2)$$

and

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}} \quad (5.6.3)$$

If you use *either* (5.6.2) *or* (5.6.3) to get the two roots, you are asking for trouble: If either  $a$  or  $c$  (or both) is small, then one of the roots will involve the subtraction of  $b$  from a very nearly equal quantity (the discriminant); you will get that root very inaccurately. The correct way to compute the roots is

$$q \equiv -\frac{1}{2} \left[ b + \text{sgn}(b) \sqrt{b^2 - 4ac} \right] \quad (5.6.4)$$

Then the two roots are

$$x_1 = \frac{q}{a} \quad \text{and} \quad x_2 = \frac{c}{q} \quad (5.6.5)$$

If the coefficients  $a, b, c$ , are complex rather than real, then the above formulas still hold, except that in equation (5.6.4) the sign of the square root should be chosen so as to make

$$\text{Re}(b^* \sqrt{b^2 - 4ac}) \geq 0 \quad (5.6.6)$$

where  $\text{Re}$  denotes the real part and asterisk denotes complex conjugation.

Apropos of quadratic equations, this seems a convenient place to recall that the inverse hyperbolic functions  $\sinh^{-1}$  and  $\cosh^{-1}$  are in fact just logarithms of solutions to such equations

$$\sinh^{-1}(x) = \ln(x + \sqrt{x^2 + 1}) \quad (5.6.7)$$

$$\cosh^{-1}(x) = \pm \ln(x + \sqrt{x^2 - 1}) \quad (5.6.8)$$

Equation (5.6.7) is numerically robust for  $x \geq 0$ . For negative  $x$ , use the symmetry  $\sinh^{-1}(-x) = -\sinh^{-1}(x)$ . Equation (5.6.8) is of course valid only for  $x \geq 1$ .

For the *cubic equation*

$$x^3 + ax^2 + bx + c = 0 \quad (5.6.9)$$

with real or complex coefficients  $a, b, c$ , first compute

$$Q \equiv \frac{a^2 - 3b}{9} \quad \text{and} \quad R \equiv \frac{2a^3 - 9ab + 27c}{54} \quad (5.6.10)$$

If  $Q$  and  $R$  are real (always true when  $a, b, c$  are real) and  $R^2 < Q^3$ , then the cubic equation has three real roots. Find them by computing

$$\theta = \arccos(R / \sqrt{Q^3}) \quad (5.6.11)$$

in terms of which the three roots are

$$\begin{aligned} x_1 &= -2\sqrt{Q} \cos\left(\frac{\theta}{3}\right) - \frac{a}{3} \\ x_2 &= -2\sqrt{Q} \cos\left(\frac{\theta + 2\pi}{3}\right) - \frac{a}{3} \\ x_3 &= -2\sqrt{Q} \cos\left(\frac{\theta - 2\pi}{3}\right) - \frac{a}{3} \end{aligned} \quad (5.6.12)$$

(This equation first appears in Chapter VI of François Viète's treatise "De emendatione," published in 1615!)

Otherwise, compute

$$A = -\left[R + \sqrt{R^2 - Q^3}\right]^{1/3} \quad (5.6.13)$$

where the sign of the square root is chosen to make

$$\operatorname{Re}(R^* \sqrt{R^2 - Q^3}) \geq 0 \quad (5.6.14)$$

(asterisk again denoting complex conjugation). If  $Q$  and  $R$  are both real, equations (5.6.13) – (5.6.14) are equivalent to

$$A = -\operatorname{sgn}(R) \left[|R| + \sqrt{R^2 - Q^3}\right]^{1/3} \quad (5.6.15)$$

where the positive square root is assumed. Next compute

$$B = \begin{cases} Q/A & (A \neq 0) \\ 0 & (A = 0) \end{cases} \quad (5.6.16)$$

in terms of which the three roots are

$$x_1 = (A + B) - \frac{a}{3} \quad (5.6.17)$$

(the single real root when  $a, b, c$  are real) and

$$\begin{aligned} x_2 &= -\frac{1}{2}(A + B) - \frac{a}{3} + i \frac{\sqrt{3}}{2}(A - B) \\ x_3 &= -\frac{1}{2}(A + B) - \frac{a}{3} - i \frac{\sqrt{3}}{2}(A - B) \end{aligned} \quad (5.6.18)$$

(in that same case, a complex-conjugate pair). Equations (5.6.13) – (5.6.16) are arranged both to minimize roundoff error and also (as pointed out by A.J. Glassman) to ensure that no choice of branch for the complex cube root can result in the spurious loss of a distinct root.

If you need to solve many cubic equations with only slightly different coefficients, it is more efficient to use Newton's method (§9.4).

#### CITED REFERENCES AND FURTHER READING:

- Weast, R.C. (ed.) 1967, *Handbook of Tables for Mathematics*, 3rd ed. (Cleveland: The Chemical Rubber Co.), pp. 130–133.
- Pachner, J. 1983, *Handbook of Numerical Analysis Applications* (New York: McGraw-Hill), §6.1.
- McKelvey, J.P. 1984, "Simple Transcendental Expressions for the Roots of Cubic Equations," *American Journal of Physics*, vol. 52, pp. 269–270; see also vol. 53, p. 775, and vol. 55, pp. 374–375.

## 5.7 Numerical Derivatives

Imagine that you have a procedure that computes a function  $f(x)$ , and now you want to compute its derivative  $f'(x)$ . Easy, right? The definition of the derivative, the limit as  $h \rightarrow 0$  of

$$f'(x) \approx \frac{f(x + h) - f(x)}{h} \quad (5.7.1)$$

practically suggests the program: Pick a small value  $h$ ; evaluate  $f(x + h)$ ; you probably have  $f(x)$  already evaluated, but if not, do it too; finally, apply equation (5.7.1). What more needs to be said?

Quite a lot, actually. Applied uncritically, the above procedure is almost guaranteed to produce inaccurate results. Applied properly, it can be the right way to compute a derivative only when the function  $f$  is *fiercely* expensive to compute; when you already have invested in computing  $f(x)$ ; and when, therefore, you want to get the derivative in no more than a single additional function evaluation. In such a situation, the remaining issue is to choose  $h$  properly, an issue we now discuss.

There are two sources of error in equation (5.7.1), truncation error and roundoff error. The truncation error comes from higher terms in the Taylor series expansion,

$$f(x + h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \dots \quad (5.7.2)$$

whence

$$\frac{f(x + h) - f(x)}{h} = f' + \frac{1}{2}hf'' + \dots \quad (5.7.3)$$

The roundoff error has various contributions. First there is roundoff error in  $h$ : Suppose, by way of an example, that you are at a point  $x = 10.3$  and you blindly choose  $h = 0.0001$ . Neither  $x = 10.3$  nor  $x + h = 10.30001$  is a number with an exact representation in binary; each is therefore represented with some fractional error characteristic of the machine's floating-point format,  $\epsilon_m$ , whose value in single precision may be  $\sim 10^{-7}$ . The error in the *effective* value of  $h$ , namely the difference between  $x + h$  and  $x$  as represented in the machine, is therefore on the order of  $\epsilon_m x$ ,

which implies a fractional error in  $h$  of order  $\sim \epsilon_m x / h \sim 10^{-2}!$  By equation (5.7.1), this immediately implies at least the same large fractional error in the derivative.

We arrive at Lesson 1: Always choose  $h$  so that  $x + h$  and  $x$  differ by an exactly representable number. This can usually be accomplished by the program steps

$$\begin{aligned}\text{temp} &= x + h \\ h &= \text{temp} - x\end{aligned}\tag{5.7.4}$$

Some optimizing compilers, and some computers whose floating-point chips have higher internal accuracy than is stored externally, can foil this trick; if so, it is usually enough to declare `temp` as `volatile`, or else to call a dummy function `donothing(temp)` between the two equations (5.7.4). This forces `temp` into and out of addressable memory.

With  $h$  an “exact” number, the roundoff error in equation (5.7.1) is approximately  $e_r \sim \epsilon_f |f(x)/h|$ . Here  $\epsilon_f$  is the fractional accuracy with which  $f$  is computed; for a simple function this may be comparable to the machine accuracy,  $\epsilon_f \approx \epsilon_m$ , but for a complicated calculation with additional sources of inaccuracy it may be larger. The truncation error in equation (5.7.3) is on the order of  $e_t \sim |hf''(x)|$ . Varying  $h$  to minimize the sum  $e_r + e_t$  gives the optimal choice of  $h$ ,

$$h \sim \sqrt{\frac{\epsilon_f f}{f''}} \approx \sqrt{\epsilon_f} x_c\tag{5.7.5}$$

where  $x_c \equiv (f/f'')^{1/2}$  is the “curvature scale” of the function  $f$  or the “characteristic scale” over which it changes. In the absence of any other information, one often assumes  $x_c = x$  (except near  $x = 0$ , where some other estimate of the typical  $x$  scale should be used).

With the choice of equation (5.7.5), the fractional accuracy of the computed derivative is

$$(e_r + e_t)/|f'| \sim \sqrt{\epsilon_f} (ff''/f'^2)^{1/2} \sim \sqrt{\epsilon_f}\tag{5.7.6}$$

Here the last order-of-magnitude equality assumes that  $f$ ,  $f'$ , and  $f''$  all share the same characteristic length scale, which is usually the case. One sees that the simple finite difference equation (5.7.1) gives *at best* only the square root of the machine accuracy  $\epsilon_m$ .

If you can afford two function evaluations for each derivative calculation, then it is significantly better to use the symmetrized form

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}\tag{5.7.7}$$

In this case, by equation (5.7.2), the truncation error is  $e_t \sim h^2 f'''$ . The roundoff error  $e_r$  is about the same as before. The optimal choice of  $h$ , by a short calculation analogous to the one above, is now

$$h \sim \left(\frac{\epsilon_f f}{f'''}\right)^{1/3} \sim (\epsilon_f)^{1/3} x_c\tag{5.7.8}$$

and the fractional error is

$$(e_r + e_t)/|f'| \sim (\epsilon_f)^{2/3} f^{2/3} (f''')^{1/3} / f' \sim (\epsilon_f)^{2/3}\tag{5.7.9}$$

which will typically be an order of magnitude (single precision) or two orders of magnitude (double precision) *better* than equation (5.7.6). We have arrived at Lesson 2: Choose  $h$  to be *the correct* power of  $\epsilon_f$  or  $\epsilon_m$  times a characteristic scale  $x_c$ .

You can easily derive the correct powers for other cases [1]. For a function of two dimensions, for example, and the mixed derivative formula

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{[f(x+h, y+h) - f(x+h, y-h)] - [f(x-h, y+h) - f(x-h, y-h)]}{4h^2} \quad (5.7.10)$$

the correct scaling is  $h \sim \epsilon_f^{1/4} x_c$ .

It is disappointing, certainly, that no simple finite difference formula like equation (5.7.1) or (5.7.7) gives an accuracy comparable to the machine accuracy  $\epsilon_m$ , or even the lower accuracy to which  $f$  is evaluated,  $\epsilon_f$ . Are there no better methods?

Yes, there are. All, however, involve exploration of the function's behavior over scales comparable to  $x_c$ , plus some assumption of smoothness, or analyticity, so that the high-order terms in a Taylor expansion like equation (5.7.2) have some meaning. Such methods also involve multiple evaluations of the function  $f$ , so their increased accuracy must be weighed against increased cost.

The general idea of “Richardson’s deferred approach to the limit” is particularly attractive. For numerical integrals, that idea leads to so-called Romberg integration (for review, see §4.3). For derivatives, one seeks to extrapolate, to  $h \rightarrow 0$ , the result of finite difference calculations with smaller and smaller finite values of  $h$ . By the use of Neville’s algorithm (§3.2), one uses each new finite difference calculation to produce both an extrapolation of higher order and also extrapolations of previous, lower, orders but with smaller scales  $h$ . Ridders [2] has given a nice implementation of this idea; the following program, `dfridr`, is based on his algorithm, modified by an improved termination criterion. Input to the routine is a function  $f$  (called `func`), a position  $x$ , and a *largest* stepsize  $h$  (more analogous to what we have called  $x_c$  above than to what we have called  $h$ ). Output is the returned value of the derivative and an estimate of its error, `err`.

```
template<class T>
Doub dfridr(T &func, const Doub x, const Doub h, Doub &err)
>Returns the derivative of a function func at a point x by Ridders' method of polynomial extrapolation. The value h is input as an estimated initial stepsize; it need not be small, but rather should be an increment in x over which func changes substantially. An estimate of the error in the derivative is returned as err.
{
    const Int ntab=10;                                Sets maximum size of tableau.
    const Doub con=1.4, con2=(con*con);              Stepsize decreased by CON at each iteration.
    const Doub big=numeric_limits<Doub>::max();      Return when error is SAFE worse than the
    const Doub safe=2.0;                            best so far.
    Int i,j;
    Doub errt,fac,hh,ans;
    MatDoub a(ntab,ntab);
    if (h == 0.0) throw("h must be nonzero in dfridr.");
    hh=h;
    a[0][0]=(func(x+hh)-func(x-hh))/(2.0*hh);
    err=big;
    for (i=1;i<ntab;i++) {
        Successive columns in the Neville tableau will go to smaller stepsizes and higher orders of
        extrapolation.
        hh /= con;
        a[0][i]=(func(x+hh)-func(x-hh))/(2.0*hh);      Try new, smaller stepsize.
        fac=con2;
```

```

for (j=1;j<=i;j++) {           Compute extrapolations of various orders, requiring
    a[j][i]=(a[j-1][i]*fac-a[j-1][i-1])/(fac-1.0);   no new function eval-
    fac=con2*fac;                                         uations.
    errt=MAX(abs(a[j][i]-a[j-1][i]),abs(a[j][i]-a[j-1][i-1]));
    The error strategy is to compare each new extrapolation to one order lower, both
    at the present stepsize and the previous one.
    if (errt <= err) {           If error is decreased, save the improved answer.
        err=errt;
        ans=a[j][i];
    }
}
if (abs(a[i][i]-a[i-1][i-1]) >= safe*err) break;
If higher order is worse by a significant factor SAFE, then quit early.
}
return ans;
}

```

In `dfridr`, the number of evaluations of `func` is typically 6 to 12, but is allowed to be as great as  $2 \times \text{NTAB}$ . As a function of input  $h$ , it is typical for the accuracy to get *better* as  $h$  is made larger, until a sudden point is reached where nonsensical extrapolation produces an early return with a large error. You should therefore choose a fairly large value for  $h$  but monitor the returned value `err`, decreasing  $h$  if it is not small. For functions whose characteristic  $x$  scale is of order unity, we typically take  $h$  to be a few tenths.

Besides Ridders' method, there are other possible techniques. If your function is fairly smooth, and you know that you will want to evaluate its derivative many times at arbitrary points in some interval, then it makes sense to construct a Chebyshev polynomial approximation to the function in that interval, and to evaluate the derivative directly from the resulting Chebyshev coefficients. This method is described in §5.8 – §5.9, following.

Another technique applies when the function consists of data that is tabulated at equally spaced intervals, and perhaps also noisy. One might then want, at each point, to least-squares *fit* a polynomial of some degree  $M$ , using an additional number  $n_L$  of points to the left and some number  $n_R$  of points to the right of each desired  $x$  value. The estimated derivative is then the derivative of the resulting fitted polynomial. A very efficient way to do this construction is via Savitzky-Golay smoothing filters, which will be discussed later, in §14.9. There we will give a routine for getting filter coefficients that not only construct the fitting polynomial but, in the accumulation of a single sum of data points times filter coefficients, evaluate it as well. In fact, the routine given, `savgo1`, has an argument `ld` that determines which derivative of the fitted polynomial is evaluated. For the first derivative, the appropriate setting is `ld=1`, and the value of the derivative is the accumulated sum divided by the sampling interval  $h$ .

#### CITED REFERENCES AND FURTHER READING:

- Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; reprinted 1996 (Philadelphia: S.I.A.M.), §5.4 – §5.6.[1]
- Ridders, C.J.F. 1982, “Accurate computation of  $F'(x)$  and  $F'(x)F''(x)$ ,” *Advances in Engineering Software*, vol. 4, no. 2, pp. 75–76.[2]

## 5.8 Chebyshev Approximation

The Chebyshev polynomial of degree  $n$  is denoted  $T_n(x)$  and is given by the explicit formula

$$T_n(x) = \cos(n \arccos x) \quad (5.8.1)$$

This may look trigonometric at first glance (and there is in fact a close relation between the Chebyshev polynomials and the discrete Fourier transform); however, (5.8.1) can be combined with trigonometric identities to yield explicit expressions for  $T_n(x)$  (see Figure 5.8.1):

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\dots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned} \quad (5.8.2)$$

(There also exist inverse formulas for the powers of  $x$  in terms of the  $T_n$ 's — see, e.g., [1].)

The Chebyshev polynomials are orthogonal in the interval  $[-1, 1]$  over a weight  $(1 - x^2)^{-1/2}$ . In particular,

$$\int_{-1}^1 \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}} dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases} \quad (5.8.3)$$

The polynomial  $T_n(x)$  has  $n$  zeros in the interval  $[-1, 1]$ , and they are located at the points

$$x = \cos\left(\frac{\pi(k + \frac{1}{2})}{n}\right) \quad k = 0, 1, \dots, n-1 \quad (5.8.4)$$

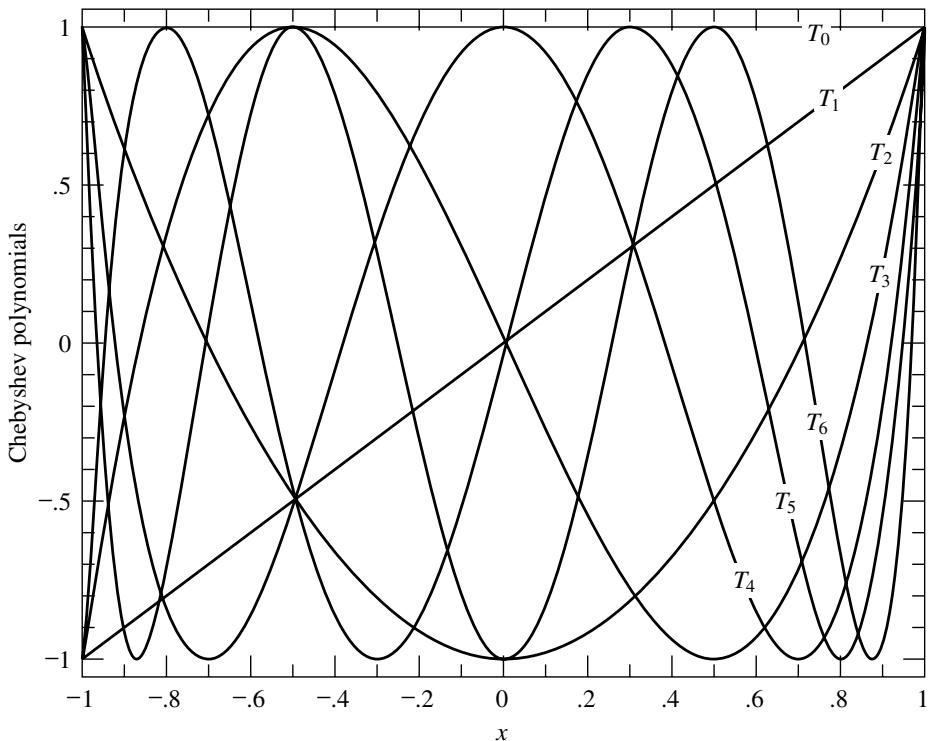
In this same interval there are  $n + 1$  extrema (maxima and minima), located at

$$x = \cos\left(\frac{\pi k}{n}\right) \quad k = 0, 1, \dots, n \quad (5.8.5)$$

At all of the maxima  $T_n(x) = 1$ , while at all of the minima  $T_n(x) = -1$ ; it is precisely this property that makes the Chebyshev polynomials so useful in polynomial approximation of functions.

The Chebyshev polynomials satisfy a discrete orthogonality relation as well as the continuous one (5.8.3): If  $x_k$  ( $k = 0, \dots, m-1$ ) are the  $m$  zeros of  $T_m(x)$  given by (5.8.4), and if  $i, j < m$ , then

$$\sum_{k=0}^{m-1} T_i(x_k)T_j(x_k) = \begin{cases} 0 & i \neq j \\ m/2 & i = j \neq 0 \\ m & i = j = 0 \end{cases} \quad (5.8.6)$$



**Figure 5.8.1.** Chebyshev polynomials  $T_0(x)$  through  $T_6(x)$ . Note that  $T_j$  has  $j$  roots in the interval  $(-1, 1)$  and that all the polynomials are bounded between  $\pm 1$ .

It is not too difficult to combine equations (5.8.1), (5.8.4), and (5.8.6) to prove the following theorem: If  $f(x)$  is an arbitrary function in the interval  $[-1, 1]$ , and if  $N$  coefficients  $c_j$ ,  $j = 0, \dots, N - 1$ , are defined by

$$\begin{aligned} c_j &= \frac{2}{N} \sum_{k=0}^{N-1} f(x_k) T_j(x_k) \\ &= \frac{2}{N} \sum_{k=0}^{N-1} f \left[ \cos \left( \frac{\pi(k + \frac{1}{2})}{N} \right) \right] \cos \left( \frac{\pi j(k + \frac{1}{2})}{N} \right) \end{aligned} \quad (5.8.7)$$

then the approximation formula

$$f(x) \approx \left[ \sum_{k=0}^{N-1} c_k T_k(x) \right] - \frac{1}{2} c_0 \quad (5.8.8)$$

is *exact* for  $x$  equal to all of the  $N$  zeros of  $T_N(x)$ .

For a fixed  $N$ , equation (5.8.8) is a polynomial in  $x$  that approximates the function  $f(x)$  in the interval  $[-1, 1]$  (where all the zeros of  $T_N(x)$  are located). Why is this particular approximating polynomial better than any other one, exact on some other set of  $N$  points? The answer is *not* that (5.8.8) is necessarily more accurate

than some other approximating polynomial of the same order  $N$  (for some specified definition of “accurate”), but rather that (5.8.8) can be truncated to a polynomial of *lower* degree  $m \ll N$  in a very graceful way, one that *does* yield the “most accurate” approximation of degree  $m$  (in a sense that can be made precise). Suppose  $N$  is so large that (5.8.8) is virtually a perfect approximation of  $f(x)$ . Now consider the truncated approximation

$$f(x) \approx \left[ \sum_{k=0}^{m-1} c_k T_k(x) \right] - \frac{1}{2} c_0 \quad (5.8.9)$$

with the same  $c_j$ ’s, computed from (5.8.7). Since the  $T_k(x)$ ’s are all bounded between  $\pm 1$ , the difference between (5.8.9) and (5.8.8) can be no larger than the sum of the neglected  $c_k$ ’s ( $k = m, \dots, N-1$ ). In fact, if the  $c_k$ ’s are rapidly decreasing (which is the typical case), then the error is dominated by  $c_m T_m(x)$ , an oscillatory function with  $m+1$  equal extrema distributed smoothly over the interval  $[-1, 1]$ . This smooth spreading out of the error is a very important property: The Chebyshev approximation (5.8.9) is very nearly the same polynomial as that holy grail of approximating polynomials the *minimax polynomial*, which (among all polynomials of the same degree) has the smallest maximum deviation from the true function  $f(x)$ . The minimax polynomial is very difficult to find; the Chebyshev approximating polynomial is almost identical and is very easy to compute!

So, given some (perhaps difficult) means of computing the function  $f(x)$ , we now need algorithms for implementing (5.8.7) and (after inspection of the resulting  $c_k$ ’s and choice of a truncating value  $m$ ) evaluating (5.8.9). The latter equation then becomes an easy way of computing  $f(x)$  for all subsequent time.

The first of these tasks is straightforward. A generalization of equation (5.8.7) that is here implemented is to allow the range of approximation to be between two arbitrary limits  $a$  and  $b$ , instead of just  $-1$  to  $1$ . This is effected by a change of variable

$$y \equiv \frac{x - \frac{1}{2}(b+a)}{\frac{1}{2}(b-a)} \quad (5.8.10)$$

and by the approximation of  $f(x)$  by a Chebyshev polynomial in  $y$ .

It will be convenient for us to group a number of functions related to Chebyshev polynomials into a single object, even though discussion of their specifics is spread out over §5.8 – §5.11:

```
struct Chebyshev {
    Object for Chebyshev approximation and related methods.
    Int n,m;                                Number of total, and truncated, coefficients.
    VecDoub c;                               Doub a,b;          Approximation interval.

    Chebyshev(Doub func(Doub), Doub aa, Doub bb, Int nn);
    Constructor. Approximate the function func in the interval [aa,bb] with nn terms.
    Chebyshev(VecDoub &cc, Doub aa, Doub bb)
        : n(cc.size()), m(n), c(cc), a(aa), b(bb) {}
    Constructor from previously computed coefficients.
    Int setm(Doub thresh) {while (m>1 && abs(c[m-1])<thresh) m--; return m;}
    Set m, the number of coefficients after truncating to an error level thresh, and return the
    value set.
```

chebyshev.h

```

Doub eval(Doub x, Int m);
inline Doub operator() (Doub x) {return eval(x,m);}
Return a value for the Chebyshev fit, either using the stored m or else overriding it.

Chebyshev derivative();           See §5.9.
Chebyshev integral();
```

**VecDoub polycofs(Int m);** See §5.10.  
**inline VecDoub polycofs() {return polycofs(m);}**  
**Chebyshev(VecDoub &pc);** See §5.11.

};

The first constructor, the one with an arbitrary function `func` as its first argument, calculates and saves `nn` Chebyshev coefficients that approximate `func` in the range `aa` to `bb`. (You can ignore for now the second constructor, which simply makes a Chebyshev object from already-calculated data.) Let us also note the method `setm`, which provides a quick way to truncate the Chebyshev series by (in effect) deleting, from the right, all coefficients smaller in magnitude than some threshold `thresh`.

**chebyshev.h**

```

Chebyshev::Chebyshev(Doub func(Doub), Doub aa, Doub bb, Int nn=50)
    : n(nn), m(nn), c(n), a(aa), b(bb)
```

Chebyshev fit: Given a function `func`, lower and upper limits of the interval  $[a,b]$ , compute and save `nn` coefficients of the Chebyshev approximation such that  $\text{func}(x) \approx [\sum_{k=0}^{nn-1} c_k T_k(y)] - c_0/2$ , where  $y$  and  $x$  are related by (5.8.10). This routine is intended to be called with moderately large `n` (e.g., 30 or 50), the array of `c`'s subsequently to be truncated at the smaller value `m` such that  $c_m$  and subsequent elements are negligible.

```

{
    const Doub pi=3.141592653589793;
    Int k,j;
    Doub fac,bpa,bma,y,sum;
    VecDoub f(n);
    bma=0.5*(b-a);
    bpa=0.5*(b+a);
    for (k=0;k<n;k++) {           We evaluate the function at the n points required
        y=cos(pi*(k+0.5)/n);      by (5.8.7).
        f[k]=func(y*bma+bpa);
    }
    fac=2.0/n;
    for (j=0;j<n;j++) {           Now evaluate (5.8.7).
        sum=0.0;
        for (k=0;k<n;k++)
            sum += f[k]*cos(pi*j*(k+0.5)/n);
        c[j]=fac*sum;
    }
}
```

If you find that the constructor's execution time is dominated by the calculation of  $N^2$  cosines, rather than by the  $N$  evaluations of your function, then you should look ahead to §12.3, especially equation (12.4.16), which shows how fast cosine transform methods can be used to evaluate equation (5.8.7).

Now that we have the Chebyshev coefficients, how do we evaluate the approximation? One could use the recurrence relation of equation (5.8.2) to generate values for  $T_k(x)$  from  $T_0 = 1, T_1 = x$ , while also accumulating the sum of (5.8.9). It is better to use Clenshaw's recurrence formula (§5.4), effecting the two processes simultaneously. Applied to the Chebyshev series (5.8.9), the recurrence is

$$\begin{aligned}d_{m+1} &\equiv d_m \equiv 0 \\d_j &= 2x d_{j+1} - d_{j+2} + c_j \quad j = m-1, m-2, \dots, 1 \\f(x) &\equiv d_0 = x d_1 - d_2 + \frac{1}{2} c_0\end{aligned}\tag{5.8.11}$$

```
Doub Chebyshev::eval(Doub x, Int m)
```

chebyshev.h

Chebyshev evaluation: The Chebyshev polynomial  $\sum_{k=0}^{m-1} c_k T_k(y) - c_0/2$  is evaluated at a point  $y = [x - (b + a)/2]/[(b - a)/2]$ , and the result is returned as the function value.

```
{ Doub d=0.0, dd=0.0, sv, y, y2;
Int j;
if ((x-a)*(x-b) > 0.0) throw("x not in range in Chebyshev::eval");
y2=2.0*(y=(2.0*x-a-b)/(b-a)); Change of variable.
for (j=m-1;j>0;j--) { Clenshaw's recurrence.
    sv=d;
    d=y2*d-dd+c[j];
    dd=sv;
}
return y*d-dd+0.5*c[0]; Last step is different.
}
```

The method `eval` has an argument for specifying how many leading coefficients `m` should be used in the evaluation. If you simply want to use a stored value of `m` that was set by a previous call to `setm` (or, by hand, by you), then you can use the `Chebyshev` object as a functor. For example,

```
Chebyshev approxfunc(func,0.,1.,50);
approxfunc.setm(1.e-8);
...
y = approxfunc(x);
```

If we are approximating an *even* function on the interval  $[-1, 1]$ , its expansion will involve only even Chebyshev polynomials. It is wasteful to construct a `Chebyshev` object with all the odd coefficients zero [2]. Instead, using the half-angle identity for the cosine in equation (5.8.1), we get the relation

$$T_{2n}(x) = T_n(2x^2 - 1)\tag{5.8.12}$$

Thus we can construct a more efficient `Chebyshev` object for even functions simply by replacing the function's argument  $x$  by  $2x^2 - 1$ , and likewise when we evaluate the Chebyshev approximation.

An odd function will have an expansion involving only odd Chebyshev polynomials. It is best to rewrite it as an expansion for the function  $f(x)/x$ , which involves only even Chebyshev polynomials. This has the added benefit of giving accurate values for  $f(x)/x$  near  $x = 0$ . Don't try to construct the series by evaluating  $f(x)/x$  numerically, however. Rather, the coefficients  $c'_n$  for  $f(x)/x$  can be found from those for  $f(x)$  by recurrence:

$$\begin{aligned}c'_{N+1} &= 0 \\c'_{n-1} &= 2c_n - c'_{n+1}, \quad n = N-1, N-3, \dots\end{aligned}\tag{5.8.13}$$

Equation (5.8.13) follows from the recurrence relation in equation (5.8.2).

If you insist on evaluating an odd Chebyshev series, the efficient way is to once again to replace  $x$  by  $y = 2x^2 - 1$  as the argument of your function. Now, however,

you must also change the last formula in equation (5.8.11) to be

$$f(x) = x[(2y - 1)d_1 - d_2 + c_0] \quad (5.8.14)$$

and change the corresponding line in eval.

### 5.8.1 Chebyshev and Exponential Convergence

Since first mentioning *truncation error* in §1.1, we have seen many examples of algorithms with an adjustable order, say  $M$ , such that the truncation error decreases as the  $M$ th power of something. Examples include most of the interpolation methods in Chapter 3 and most of the quadrature methods in Chapter 4. In these examples there is also another parameter,  $N$ , which is the number of points at which a function will be evaluated.

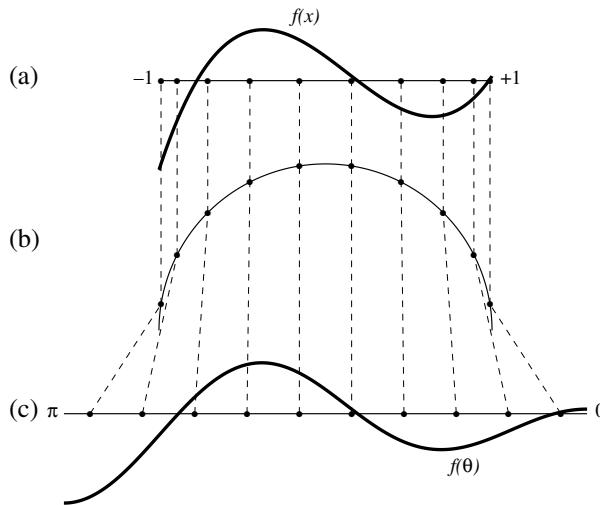
We have many times warned that “higher order does not necessarily give higher accuracy.” That remains good advice when  $N$  is held fixed while  $M$  is increased. However, a recently emerging theme in many areas of scientific computation is the use of methods that allow, in very special cases,  $M$  and  $N$  to be increased *together*, with the result that errors not only do decrease with higher order, but decrease exponentially!

The common thread in almost all of these relatively new methods is the remarkable fact that *infinitely smooth* functions become *exponentially* well determined by  $N$  sample points as  $N$  is increased. Thus, mere power-law convergence may be just a consequence of either (i) functions that are not smooth enough, or (ii) endpoint effects.

We already saw several examples of this in Chapter 4. In §4.1 we pointed out that high-order quadrature rules can have interior weights of unity, just like the trapezoidal rule; all of the “high-orderness” is obtained by a proper treatment near the boundaries. In §4.5 we further saw that variable transformations that push the boundaries off to infinity produce rapidly converging quadrature algorithms. In §4.5.1 we in fact proved exponential convergence, as a consequence of the Euler-Maclaurin formula. Then in §4.6 we remarked on the fact that the convergence of Gaussian quadratures could be exponentially rapid (an example, in the language above, of increasing  $M$  and  $N$  simultaneously).

Chebyshev approximation can be exponentially convergent for a different (though related) reason: Smooth *periodic* functions avoid endpoint effects by not having endpoints at all! Chebyshev approximation can be viewed as mapping the  $x$  interval  $[-1, 1]$  onto the angular interval  $[0, \pi]$  (cf. equations 5.8.4 and 5.8.5) in such a way that any infinitely smooth function on the interval  $[-1, 1]$  becomes an infinitely smooth, even, periodic function on  $[0, 2\pi]$ . Figure 5.8.2 shows the idea geometrically. By projecting the abscissas onto a semicircle, a half-period is produced. The other half-period is obtained by reflection, or could be imagined as the result of projecting the function onto an identical lower semicircle. The zeros of the Chebyshev polynomial, or nodes of a Chebyshev approximation, are equally spaced on the circle, where the Chebyshev polynomial itself is a cosine function (cf. equation 5.8.1). This illustrates the close connection between Chebyshev approximation and periodic functions on the circle; in Chapter 12, we will apply the discrete Fourier transform to such functions in an almost equivalent way (§12.4.2).

The reason that Chebyshev works so well (and also why Gaussian quadratures work so well) is thus seen to be intimately related to the special way that the the



**Figure 5.8.2.** Geometrical construction showing how Chebyshev approximation is related to periodic functions. A smooth function on the interval is plotted in (a). In (b), the abscissas are mapped to a semicircle. In (c), the semicircle is unrolled. Because of the semicircle's vertical tangents, the function is now nearly constant at the endpoints. In fact, if reflected into the interval  $[\pi, 2\pi]$ , it is a smooth, even, periodic function on  $[0, 2\pi]$ .

sample points are bunched up near the endpoints of the interval. Any function that is bounded on the interval will have a convergent Chebyshev approximation as  $N \rightarrow \infty$ , even if there are nearby poles in the complex plane. For functions that are not infinitely smooth, the actual rate of convergence depends on the smoothness of the function: the more derivatives that are bounded, the greater the convergence rate. For the special case of a  $C^\infty$  function, the convergence is exponential. In §3.0, in connection with polynomial interpolation, we mentioned the other side of the coin: equally spaced samples on the interval are about the *worst* possible geometry and often lead to ill-conditioned problems.

Use of the sampling theorem (§4.5, §6.9, §12.1, §13.11) is often closely associated with exponentially convergent methods. We will return to many of the concepts of exponentially convergent methods when we discuss spectral methods for partial differential equations in §20.7.

#### CITED REFERENCES AND FURTHER READING:

- Arfken, G. 1970, *Mathematical Methods for Physicists*, 2nd ed. (New York: Academic Press), p. 631.[1]
- Clenshaw, C.W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory (London: H.M. Stationery Office).[2]
- Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 8.
- Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §4.4.1, p. 104.
- Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.5.2, p. 334.
- Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §1.10, p. 39.

## 5.9 Derivatives or Integrals of a Chebyshev-Approximated Function

If you have obtained the Chebyshev coefficients that approximate a function in a certain range (e.g., from `chebft` in §5.8), then it is a simple matter to transform them to Chebyshev coefficients corresponding to the derivative or integral of the function. Having done this, you can evaluate the derivative or integral just as if it were a function that you had Chebyshev-fitted *ab initio*.

The relevant formulas are these: If  $c_i$ ,  $i = 0, \dots, m - 1$  are the coefficients that approximate a function  $f$  in equation (5.8.9),  $C_i$  are the coefficients that approximate the indefinite integral of  $f$ , and  $c'_i$  are the coefficients that approximate the derivative of  $f$ , then

$$C_i = \frac{c_{i-1} - c_{i+1}}{2i} \quad (i > 0) \quad (5.9.1)$$

$$c'_{i-1} = c'_{i+1} + 2ic_i \quad (i = m - 1, m - 2, \dots, 1) \quad (5.9.2)$$

Equation (5.9.1) is augmented by an arbitrary choice of  $C_0$ , corresponding to an arbitrary constant of integration. Equation (5.9.2), which is a recurrence, is started with the values  $c'_m = c'_{m-1} = 0$ , corresponding to no information about the  $m + 1$ st Chebyshev coefficient of the original function  $f$ .

Here are routines for implementing equations (5.9.1) and (5.9.2). Each returns a new `Chebyshev` object on which you can `setm`, call `eval`, or use directly as a functor.

`chebyshev.h`

`Chebyshev Chebyshev::derivative()`

Return a new `Chebyshev` object that approximates the derivative of the existing function over the same range  $[a,b]$ .

```
{
    Int j;
    Doub con;
    VecDoub cder(n);
    cder[n-1]=0.0;                                n-1 and n-2 are special cases.
    cder[n-2]=2*(n-1)*c[n-1];
    for (j=n-2;j>0;j--)
        cder[j-1]=cder[j+1]+2*j*c[j];            Equation (5.9.2).
    con=2.0/(b-a);
    for (j=0;j<n;j++) cder[j] *= con;           Normalize to the interval b-a.
    return Chebyshev(cder,a,b);
}
```

`chebyshev.h`

`Chebyshev Chebyshev::integral()`

Return a new `Chebyshev` object that approximates the indefinite integral of the existing function over the same range  $[a,b]$ . The constant of integration is set so that the integral vanishes at  $a$ .

```
{
    Int j;
    Doub sum=0.0,fac=1.0,con;
    VecDoub cint(n);
    con=0.25*(b-a);                                Factor that normalizes to the interval b-a.
    for (j=1;j<n-1;j++) {
        cint[j]=con*(c[j-1]-c[j+1])/j;          Equation (5.9.1).
        sum += fac*cint[j];
        fac = -fac;                               Accumulates the constant of integration.
    }
    cint[n-1]=con*c[n-2]/(n-1);                  Will equal ±1.
}
Special case of (5.9.1) for n-1.
```

```

sum += fac*cint[n-1];
cint[0]=2.0*sum;           Set the constant of integration.
return Chebyshev(cint,a,b);
}

```

### 5.9.1 Clenshaw-Curtis Quadrature

Since a smooth function's Chebyshev coefficients  $c_i$  decrease rapidly, generally exponentially, equation (5.9.1) is often quite efficient as the basis for a quadrature scheme. As described above, the `Chebyshev` object can be used to compute the integral  $\int_a^x f(x)dx$  when many different values of  $x$  in the range  $a \leq x \leq b$  are needed. If only the single definite integral  $\int_a^b f(x)dx$  is required, then instead use the simpler formula, derived from equation (5.9.1),

$$\int_a^b f(x)dx = (b-a) \left[ \frac{1}{2}c_0 - \frac{1}{3}c_2 - \frac{1}{15}c_4 - \cdots - \frac{1}{(2k+1)(2k-1)}c_{2k} - \cdots \right] \quad (5.9.3)$$

where the  $c_i$ 's are as returned by `chebft`. The series can be truncated when  $c_{2k}$  becomes negligible, and the first neglected term gives an error estimate.

This scheme is known as *Clenshaw-Curtis quadrature* [1]. It is often combined with an adaptive choice of  $N$ , the number of Chebyshev coefficients calculated via equation (5.8.7), which is also the number of function evaluations of  $f(x)$ . If a modest choice of  $N$  does not give a sufficiently small  $c_{2k}$  in equation (5.9.3), then a larger value is tried. In this adaptive case, it is even better to replace equation (5.8.7) by the so-called "trapezoidal" or Gauss-Lobatto (§4.6) variant,

$$c_j = \frac{2}{N} \sum_{k=0}^{N-1} f \left[ \cos \left( \frac{\pi k}{N} \right) \right] \cos \left( \frac{\pi j k}{N} \right) \quad j = 0, \dots, N-1 \quad (5.9.4)$$

where (N.B.! the two primes signify that the first and last terms in the sum are to be multiplied by  $1/2$ . If  $N$  is doubled in equation (5.9.4), then half of the new function evaluation points are identical to the old ones, allowing the previous function evaluations to be reused. This feature, plus the analytic weights and abscissas (cosine functions in 5.9.4), often give Clenshaw-Curtis quadrature an edge over high-order adaptive Gaussian quadrature (cf. §4.6.4), which the method otherwise resembles.

If your problem forces you to large values of  $N$ , you should be aware that equation (5.9.4) can be evaluated rapidly, and simultaneously for all the values of  $j$ , by a fast cosine transform. (See §12.3, especially equation 12.4.11. We already remarked that the nontrapezoidal form (5.8.7) can also be done by fast cosine methods, cf. equation 12.4.16.)

#### CITED REFERENCES AND FURTHER READING:

- Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), pp. 78–79.  
 Clenshaw, C.W., and Curtis, A.R. 1960, "A Method for Numerical Integration on an Automatic Computer," *Numerische Mathematik*, vol. 2, pp. 197–205.[1]

## 5.10 Polynomial Approximation from Chebyshev Coefficients

You may well ask after reading the preceding two sections: Must I store and evaluate my Chebyshev approximation as an array of Chebyshev coefficients for a transformed variable  $y$ ?

Can't I convert the  $c_k$ 's into actual polynomial coefficients in the original variable  $x$  and have an approximation of the following form?

$$f(x) \approx \sum_{k=0}^{m-1} g_k x^k, \quad a \leq x \leq b \quad (5.10.1)$$

Yes, you can do this (and we will give you the algorithm to do it), but we caution you against it: Evaluating equation (5.10.1), where the coefficient  $g$ 's reflect an underlying Chebyshev approximation, usually requires more significant figures than evaluation of the Chebyshev sum directly (as by `eval`). This is because the Chebyshev polynomials themselves exhibit a rather delicate cancellation: The leading coefficient of  $T_n(x)$ , for example, is  $2^{n-1}$ ; other coefficients of  $T_n(x)$  are even bigger; yet they all manage to combine into a polynomial that lies between  $\pm 1$ . Only when  $m$  is no larger than 7 or 8 should you contemplate writing a Chebyshev fit as a direct polynomial, and even in those cases you should be willing to tolerate two or so significant figures less accuracy than the roundoff limit of your machine.

You get the  $g$ 's in equation (5.10.1) in two steps. First, use the member function `polycofs` in `Chebyshev` to output a set of polynomial coefficients equivalent to the stored  $c_k$ 's (that is, with the range  $[a, b]$  scaled to  $[-1, 1]$ ). Second, use the routine `pcshft` to transform the coefficients so as to map the range back to  $[a, b]$ . The two required routines are listed here:

chebyshev.h

`VecDoub Chebyshev::polycofs(Int m)`

Polynomial coefficients from a Chebyshev fit. Given a coefficient array  $c[0..n-1]$ , this routine returns a coefficient array  $d[0..n-1]$  such that  $\sum_{k=0}^{n-1} d_k y^k = \sum_{k=0}^{n-1} c_k T_k(y) - c_0/2$ . The method is Clenshaw's recurrence (5.8.11), but now applied algebraically rather than arithmetically.

```
{
    Int k,j;
    Doub sv;
    VecDoub d(m),dd(m);
    for (j=0;j<m;j++) d[j]=dd[j]=0.0;
    d[0]=c[m-1];
    for (j=m-2;j>0;j--) {
        for (k=m-j;k>0;k--) {
            sv=d[k];
            d[k]=2.0*d[k-1]-dd[k];
            dd[k]=sv;
        }
        sv=d[0];
        d[0] = -dd[0]+c[j];
        dd[0]=sv;
    }
    for (j=m-1;j>0;j--) d[j]=d[j-1]-dd[j];
    d[0] = -dd[0]+0.5*c[0];
    return d;
}
```

pcshft.h

`void pcshft(Doub a, Doub b, VecDoub_Io &d)`

Polynomial coefficient shift. Given a coefficient array  $d[0..n-1]$ , this routine generates a coefficient array  $g[0..n-1]$  such that  $\sum_{k=0}^{n-1} d_k y^k = \sum_{k=0}^{n-1} g_k x^k$ , where  $x$  and  $y$  are related by (5.8.10), i.e., the interval  $-1 < y < 1$  is mapped to the interval  $a < x < b$ . The array  $g$  is returned in  $d$ .

```
{
    Int k,j,n=d.size();
    Doub cnst=2.0/(b-a), fac=cnst;
    for (j=1;j<n;j++) {           First we rescale by the factor const...
        d[j] *= fac;
        fac *= cnst;
    }
    cnst=0.5*(a+b);               ...which is then redefined as the desired shift.
```

```

for (j=0;j<=n-2;j++)
    for (k=n-2;k>=j;k--)
        d[k] -= cnst*d[k+1];
}

```

We accomplish the shift by synthetic division, a miracle  
of high-school algebra.

**CITED REFERENCES AND FURTHER READING:**

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), pp. 59, 182–183 [synthetic division].

## 5.11 Economization of Power Series

One particular application of Chebyshev methods, the *economization of power series*, is an occasionally useful technique, with a flavor of getting something for nothing.

Suppose that you know how to compute a function by the use of a convergent power series, for example,

$$f(x) \equiv \frac{1}{2} - \frac{x}{4} + \frac{x^2}{8} - \frac{x^3}{16} + \dots \quad (5.11.1)$$

(This function is actually just  $1/(x + 2)$ , but pretend you don't know that.) You might be doing a problem that requires evaluating the series many times in some particular interval, say  $[0, 1]$ . Everything is fine, except that the series requires a large number of terms before its error (approximated by the first neglected term, say) is tolerable. In our example, with  $x = 1$ , it takes about 30 terms before the first neglected term is  $< 10^{-9}$ .

Notice that because of the large exponent in  $x^{30}$ , the error is *much smaller* than  $10^{-9}$  everywhere in the interval except at the very largest values of  $x$ . This is the feature that allows “economization”: If we are willing to let the error elsewhere in the interval rise to about the same value that the first neglected term has at the extreme end of the interval, then we can replace the 30-term series by one that is significantly shorter.

Here are the steps for doing this:

1. Compute enough coefficients of the power series to get accurate function values everywhere in the range of interest.
2. Change variables from  $x$  to  $y$ , as in equation (5.8.10), to map the  $x$  interval into  $-1 \leq y \leq 1$ .
3. Find the Chebyshev series (like equation 5.8.8) that exactly equals your truncated power series.
4. Truncate this Chebyshev series to a smaller number of terms, using the coefficient of the first neglected Chebyshev polynomial as an estimate of the error.
5. Convert back to a polynomial in  $y$ .
6. Change variables back to  $x$ .

We already have tools for all of the steps, except for steps 2 and 3. Step 2 is exactly the inverse of the routine `pcshft` (§5.10), which mapped a polynomial from  $y$  (in the interval  $[-1, 1]$ ) to  $x$  (in the interval  $[a, b]$ ). But since equation (5.8.10) is a linear relation between  $x$  and  $y$ , one can also use `pcshft` for the inverse. The inverse of

`pcshft(a,b,d,n)`

turns out to be (you can check this)

```

void ipcshft(Doub a, Doub b, VecDoub_I0 &d) {
    pcshft(-(2.+b+a)/(b-a),(2.-b-a)/(b-a),d);
}

```

`pcshft.h`

Step 3 requires a new `Chebyshev` constructor, one that computes Chebyshev coefficients from a vector of polynomial coefficients. The following code accomplishes this. The algorithm is based on constructing the polynomial by the technique of §5.3 starting with the highest coefficient  $d[n-1]$  and using the recurrence of equation (5.8.2) written in the form

$$\begin{aligned} xT_0 &= T_1 \\ xT_n &= \frac{1}{2}(T_{n+1} + T_{n-1}), \quad n \geq 1. \end{aligned} \tag{5.11.2}$$

The only subtlety is to multiply the coefficient of  $T_0$  by 2 since it gets used with a factor 1/2 in equation (5.8.8).

`chebyshev.h`

```
    Chebyshev::Chebyshev(VecDoub &d)
        : n(d.size()), m(n), c(n), a(-1.), b(1.)
```

Inverse of routine `polycofs` in `Chebyshev`: Given an array of polynomial coefficients  $d[0..n-1]$ , construct an equivalent `Chebyshev` object.

```
{
    c[n-1]=d[n-1];
    c[n-2]=2.0*d[n-2];
    for (Int j=n-3;j>=0;j--) {
        c[j]=2.0*d[j]+c[j+2];
        for (Int i=j+1;i<n-2;i++) {
            c[i] = (c[i]+c[i+2])/2;
        }
        c[n-2] /= 2;
        c[n-1] /= 2;
    }
}
```

Putting them all together, steps 2 through 6 will look something like this (starting with a vector `powser` of power series coefficients):

```
ipcshft(a,b,powser);
Chebyshev cpowser(powser);
cpowser.setm(1.e-9);
VecDoub d=cpowser.polycofs();
pcshft(a,b,d);
```

In our example, by the way, the number of terms required for  $10^{-9}$  accuracy is reduced from 30 to 9. Replacing a 30-term polynomial with a 9-term polynomial without any loss of accuracy — that does seem to be getting something for nothing. Is there some magic in this technique? Not really. The 30-term polynomial defined a function  $f(x)$ . Equivalent to economizing the series, we could instead have evaluated  $f(x)$  at enough points to construct its Chebyshev approximation in the interval of interest, by the methods of §5.8. We would have obtained just the same lower-order polynomial. The principal lesson is that the rate of convergence of Chebyshev coefficients has nothing to do with the rate of convergence of power series coefficients; and it is the *former* that dictates the number of terms needed in a polynomial approximation. A function might have a *divergent* power series in some region of interest, but if the function itself is well-behaved, it will have perfectly good polynomial approximations. These can be found by the methods of §5.8, but *not* by economization of series. There is slightly less to economization of series than meets the eye.

#### CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), Chapter 12.

## 5.12 Padé Approximants

A *Padé approximant*, so called, is that rational function (of a specified order) whose power series expansion agrees with a given power series to the highest possible order. If the rational function is

$$R(x) \equiv \frac{\sum_{k=0}^M a_k x^k}{1 + \sum_{k=1}^N b_k x^k} \quad (5.12.1)$$

then  $R(x)$  is said to be a Padé approximant to the series

$$f(x) \equiv \sum_{k=0}^{\infty} c_k x^k \quad (5.12.2)$$

if

$$R(0) = f(0) \quad (5.12.3)$$

and also

$$\left. \frac{d^k}{dx^k} R(x) \right|_{x=0} = \left. \frac{d^k}{dx^k} f(x) \right|_{x=0}, \quad k = 1, 2, \dots, M + N \quad (5.12.4)$$

Equations (5.12.3) and (5.12.4) furnish  $M + N + 1$  equations for the unknowns  $a_0, \dots, a_M$  and  $b_1, \dots, b_N$ . The easiest way to see what these equations are is to equate (5.12.1) and (5.12.2), multiply both by the denominator of equation (5.12.1), and equate all powers of  $x$  that have either  $a$ 's or  $b$ 's in their coefficients. If we consider only the special case of a diagonal rational approximation,  $M = N$  (cf. §3.4), then we have  $a_0 = c_0$ , with the remaining  $a$ 's and  $b$ 's satisfying

$$\sum_{m=1}^N b_m c_{N-m+k} = -c_{N+k}, \quad k = 1, \dots, N \quad (5.12.5)$$

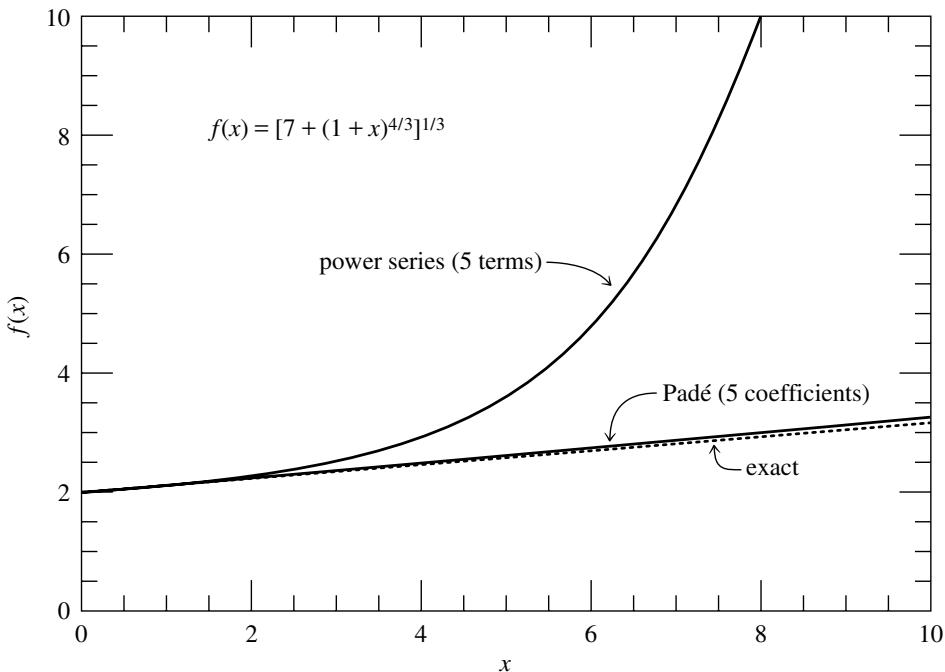
$$\sum_{m=0}^k b_m c_{k-m} = a_k, \quad k = 1, \dots, N \quad (5.12.6)$$

(note, in equation 5.12.1, that  $b_0 = 1$ ). To solve these, start with equations (5.12.5), which are a set of linear equations for all the unknown  $b$ 's. Although the set is in the form of a Toeplitz matrix (compare equation 2.8.8), experience shows that the equations are frequently close to singular, so that one should not solve them by the methods of §2.8, but rather by full *LU* decomposition. Additionally, it is a good idea to refine the solution by iterative improvement (method `mprove` in §2.5) [1].

Once the  $b$ 's are known, then equation (5.12.6) gives an explicit formula for the unknown  $a$ 's, completing the solution.

Padé approximants are typically used when there is some unknown underlying function  $f(x)$ . We suppose that you are able somehow to compute, perhaps by laborious analytic expansions, the values of  $f(x)$  and a few of its derivatives at  $x = 0$ :  $f(0), f'(0), f''(0)$ , and so on. These are of course the first few coefficients in the power series expansion of  $f(x)$ ; but they are not necessarily getting small, and you have no idea where (or whether) the power series is convergent.

By contrast with techniques like Chebyshev approximation (§5.8) or economization of power series (§5.11) that only condense the information that you already know about a function, Padé approximants can give you genuinely new information about your function's values.



**Figure 5.12.1.** The five-term power series expansion and the derived five-coefficient Padé approximant for a sample function  $f(x)$ . The full power series converges only for  $x < 1$ . Note that the Padé approximant maintains accuracy far outside the radius of convergence of the series.

It is sometimes quite mysterious how well this can work. (Like other mysteries in mathematics, it relates to *analyticity*.) An example will illustrate.

Imagine that, by extraordinary labors, you have ground out the first five terms in the power series expansion of an unknown function  $f(x)$ ,

$$f(x) \approx 2 + \frac{1}{9}x + \frac{1}{81}x^2 - \frac{49}{8748}x^3 + \frac{175}{78732}x^4 + \dots \quad (5.12.7)$$

(It is not really necessary that you know the coefficients in exact rational form — numerical values are just as good. We here write them as rationals to give you the impression that they derive from some side analytic calculation.) Equation (5.12.7) is plotted as the curve labeled “power series” in Figure 5.12.1. One sees that for  $x \gtrsim 4$  it is dominated by its largest, quartic, term.

We now take the five coefficients in equation (5.12.7) and run them through the routine `pade` listed below. It returns five rational coefficients, three  $a$ 's and two  $b$ 's, for use in equation (5.12.1) with  $M = N = 2$ . The curve in the figure labeled “Padé” plots the resulting rational function. Note that both solid curves derive from the *same* five original coefficient values.

To evaluate the results, we need *Deus ex machina* (a useful fellow, when he is available) to tell us that equation (5.12.7) is in fact the power series expansion of the function

$$f(x) = [7 + (1 + x)^{4/3}]^{1/3} \quad (5.12.8)$$

which is plotted as the dotted curve in the figure. This function has a branch point at  $x = -1$ , so its power series is convergent only in the range  $-1 < x < 1$ . In most of the range shown in the figure, the series is divergent, and the value of its truncation to five terms is rather meaningless. Nevertheless, those five terms, converted to a Padé approximant, give a remarkably good representation of the function up to at least  $x \sim 10$ .

Why does this work? Are there not other functions with the same first five terms in their power series but completely different behavior in the range (say)  $2 < x < 10$ ? Indeed there are. Padé approximation has the uncanny knack of picking the function *you had in mind* from among all the possibilities. *Except when it doesn't!* That is the downside of Padé approximation: It is uncontrolled. There is, in general, no way to tell how accurate it is, or how far out in  $x$  it can usefully be extended. It is a powerful, but in the end still mysterious, technique.

Here is the routine that returns a `Ratfn` rational function object that is the Padé approximant to a set of power series coefficients that you provide. Note that the routine is specialized to the case  $M = N$ . You can then use the `Ratfn` object directly as a functor, or else read out its coefficients by hand (§5.1).

`Ratfn pade(VecDoub_I &cof)`

Given `cof[0..2*n]`, the leading terms in the power series expansion of a function, solve the linear Padé equations to return a `Ratfn` object that embodies a diagonal rational function approximation to the same function.

```
{
    const Doub BIG=1.0e99;
    Int j,k,n=(cof.size()-1)/2;
    Doub sum;
    MatDoub q(n,n),qlu(n,n);
    VecInt indx(n);
    VecDoub x(n),y(n),num(n+1),denom(n+1);
    for (j=0;j<n;j++) {                                Set up matrix for solving.
        y[j]=cof[n+j+1];
        for (k=0;k<n;k++) q[j][k]=cof[j-k+n];
    }
    LUdcmp lu(q);                                     Solve by LU decomposition and backsubstitution,
    lu.solve(y,x);                                    with iterative improvement.
    for (j=0;j<4;j++) lu.mprove(y,x);
    for (k=0;k<n;k++) {                                Calculate the remaining coefficients.
        for (sum=cof[k+1],j=0;j<=k;j++) sum -= x[j]*cof[k-j];
        y[k]=sum;
    }
    num[0] = cof[0];
    denom[0] = 1.;
    for (j=0;j<n;j++) {                                Copy answers to output.
        num[j+1]=y[j];
        denom[j+1] = -x[j];
    }
    return Ratfn(num,denom);
}
```

`pade.h`

#### CITED REFERENCES AND FURTHER READING:

- Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), p. 14.
- Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 2.
- Graves-Morris, P.R. 1979, in *Padé Approximation and Its Applications*, Lecture Notes in Mathematics, vol. 765, L. Wuytack, ed. (Berlin: Springer).[1]

## 5.13 Rational Chebyshev Approximation

In §5.8 and §5.10 we learned how to find good polynomial approximations to a given function  $f(x)$  in a given interval  $a \leq x \leq b$ . Here, we want to generalize the task to find

good approximations that are rational functions (see §5.1). The reason for doing so is that, for some functions and some intervals, the optimal rational function approximation is able to achieve substantially higher accuracy than the optimal polynomial approximation with the same number of coefficients. This must be weighed against the fact that finding a rational function approximation is not as straightforward as finding a polynomial approximation, which, as we saw, could be done elegantly via Chebyshev polynomials.

Let the desired rational function  $R(x)$  have a numerator of degree  $m$  and denominator of degree  $k$ . Then we have

$$R(x) \equiv \frac{p_0 + p_1 x + \cdots + p_m x^m}{1 + q_1 x + \cdots + q_k x^k} \approx f(x) \quad \text{for } a \leq x \leq b \quad (5.13.1)$$

The unknown quantities that we need to find are  $p_0, \dots, p_m$  and  $q_1, \dots, q_k$ , that is,  $m + k + 1$  quantities in all. Let  $r(x)$  denote the deviation of  $R(x)$  from  $f(x)$ , and let  $r$  denote its maximum absolute value,

$$r(x) \equiv R(x) - f(x) \quad r \equiv \max_{a \leq x \leq b} |r(x)| \quad (5.13.2)$$

The ideal *minimax* solution would be that choice of  $p$ 's and  $q$ 's that minimizes  $r$ . Obviously there is *some* minimax solution, since  $r$  is bounded below by zero. How can we find it, or a reasonable approximation to it?

A first hint is furnished by the following fundamental theorem: If  $R(x)$  is nondegenerate (has no common polynomial factors in numerator and denominator), then there is a unique choice of  $p$ 's and  $q$ 's that minimizes  $r$ ; for this choice,  $r(x)$  has  $m + k + 2$  extrema in  $a \leq x \leq b$ , *all of magnitude  $r$  and with alternating sign*. (We have omitted some technical assumptions in this theorem. See Ralston [1] for a precise statement.) We thus learn that the situation with rational functions is quite analogous to that for minimax polynomials: In §5.8 we saw that the error term of an  $n$ th-order approximation, with  $n + 1$  Chebyshev coefficients, was generally dominated by the first neglected Chebyshev term, namely  $T_{n+1}$ , which itself has  $n + 2$  extrema of equal magnitude and alternating sign. So, here, the number of rational coefficients,  $m + k + 1$ , plays the same role of the number of polynomial coefficients,  $n + 1$ .

A different way to see why  $r(x)$  should have  $m + k + 2$  extrema is to note that  $R(x)$  can be made exactly equal to  $f(x)$  at any  $m + k + 1$  points  $x_i$ . Multiplying equation (5.13.1) by its denominator gives the equations

$$p_0 + p_1 x_i + \cdots + p_m x_i^m = f(x_i)(1 + q_1 x_i + \cdots + q_k x_i^k) \quad i = 0, 1, \dots, m+k \quad (5.13.3)$$

This is a set of  $m + k + 1$  linear equations for the unknown  $p$ 's and  $q$ 's, which can be solved by standard methods (e.g., LU decomposition). If we choose the  $x_i$ 's to all be in the interval  $(a, b)$ , then there will generically be an extremum between each chosen  $x_i$  and  $x_{i+1}$ , plus also extrema where the function goes out of the interval at  $a$  and  $b$ , for a total of  $m + k + 2$  extrema. For arbitrary  $x_i$ 's, the extrema will not have the same magnitude. The theorem says that, for one particular choice of  $x_i$ 's, the magnitudes can be beaten down to the identical, minimal, value of  $r$ .

Instead of making  $f(x_i)$  and  $R(x_i)$  equal at the points  $x_i$ , one can instead force the residual  $r(x_i)$  to any desired values  $y_i$  by solving the linear equations

$$p_0 + p_1 x_i + \cdots + p_m x_i^m = [f(x_i) - y_i](1 + q_1 x_i + \cdots + q_k x_i^k) \quad i = 0, 1, \dots, m+k \quad (5.13.4)$$

In fact, if the  $x_i$ 's are chosen to be the extrema (not the zeros) of the minimax solution, then the equations satisfied will be

$$p_0 + p_1 x_i + \cdots + p_m x_i^m = [f(x_i) \pm r](1 + q_1 x_i + \cdots + q_k x_i^k) \quad i = 0, 1, \dots, m+k+1 \quad (5.13.5)$$

where the  $\pm$  alternates for the alternating extrema. Notice that equation (5.13.5) is satisfied at  $m + k + 2$  extrema, while equation (5.13.4) was satisfied only at  $m + k + 1$  arbitrary points. How can this be? The answer is that  $r$  in equation (5.13.5) is an additional unknown, so that

the number of both equations and unknowns is  $m + k + 2$ . True, the set is mildly nonlinear (in  $r$ ), but in general it is still perfectly soluble by methods that we will develop in Chapter 9.

We thus see that, given only the *locations* of the extrema of the minimax rational function, we can solve for its coefficients and maximum deviation. Additional theorems, leading up to the so-called *Remes algorithms* [1], tell how to converge to these locations by an iterative process. For example, here is a (slightly simplified) statement of *Remes' Second Algorithm*:

- (1) Find an initial rational function with  $m + k + 2$  extrema  $x_i$  (not having equal deviation).
- (2) Solve equation (5.13.5) for new rational coefficients and  $r$ .
- (3) Evaluate the resulting  $R(x)$  to find its actual extrema (which will not be the same as the guessed values).
- (4) Replace each guessed value with the nearest actual extremum of the same sign.
- (5) Go back to step 2 and iterate to convergence.

Under a broad set of assumptions, this method will converge. Ralston [1] fills in the necessary details, including how to find the initial set of  $x_i$ 's.

Up to this point, our discussion has been textbook standard. We now reveal ourselves as heretics. We don't much like the elegant Remes algorithm. Its two nested iterations (on  $r$  in the nonlinear set 5.13.5, and on the new sets of  $x_i$ 's) are finicky and require a lot of special logic for degenerate cases. Even more heretical, we doubt that compulsive searching for the *exactly best*, equal deviation approximation is worth the effort — except perhaps for those few people in the world whose business it is to find optimal approximations that get built into compilers and microcode.

When we use rational function approximation, the goal is usually much more pragmatic: Inside some inner loop we are evaluating some function a zillion times, and we want to speed up its evaluation. Almost never do we need this function to the last bit of machine accuracy. Suppose (heresy!) we use an approximation whose error has  $m + k + 2$  extrema whose deviations differ by a factor of 2. The theorems on which the Remes algorithms are based guarantee that the perfect minimax solution will have extrema somewhere within this factor of 2 range — forcing down the higher extrema will cause the lower ones to rise, until all are equal. So our “sloppy” approximation is in fact within a fraction of a least significant bit of the minimax one.

That is good enough for us, especially when we have available a very robust method for finding the so-called “sloppy” approximation. Such a method is the least-squares solution of overdetermined linear equations by singular value decomposition (§2.6 and §15.4). We proceed as follows: First, solve (in the least-squares sense) equation (5.13.3), not just for  $m + k + 1$  values of  $x_i$ , but for a significantly larger number of  $x_i$ 's, spaced approximately like the zeros of a high-order Chebyshev polynomial. This gives an initial guess for  $R(x)$ . Second, tabulate the resulting deviations, find the mean absolute deviation, call it  $r$ , and then solve (again in the least-squares sense) equation (5.13.5) with  $r$  fixed and the  $\pm$  chosen to be the sign of the observed deviation at each point  $x_i$ . Third, repeat the second step a few times.

You can spot some Remes orthodoxy lurking in our algorithm: The equations we solve are trying to bring the deviations not to zero, but rather to plus-or-minus some consistent value. However, we dispense with keeping track of actual extrema, and we solve only linear equations at each stage. One additional trick is to solve a *weighted* least-squares problem, where the weights are chosen to beat down the largest deviations fastest.

Here is a function implementing these ideas. Notice that the only calls to the function `fn` occur in the initial filling of the table `fs`. You could easily modify the code to do this filling outside of the routine. It is not even necessary that your abscissas `xs` be exactly the ones that we use, though the quality of the fit will deteriorate if you do not have several abscissas between each extremum of the (underlying) minimax solution. The function returns a `Ratfn` object that you can subsequently use as a functor, or from which you can extract the stored coefficients.

```
Ratfn ratlsq(Doub fn(const Doub), const Doub a, const Doub b, const Int mm,
               const Int kk, Doub &dev)
{
    const Int NPFAC=8,MAXIT=5;
    const Doub BIG=1.0e99,PI02=1.570796326794896619;
```

`ratlsq.h`

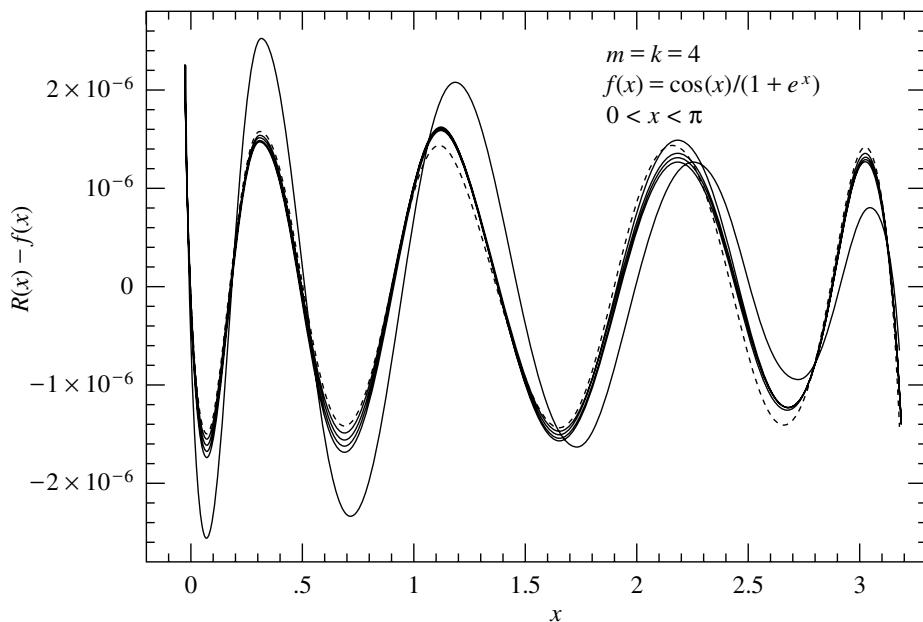
Returns a rational function approximation to the function `fn` in the interval  $(a, b)$ . Input quantities `mm` and `kk` specify the order of the numerator and denominator, respectively. The maximum absolute deviation of the approximation (insofar as is known) is returned as `dev`.

```

Int i,it,j,ncof=mm+kk+1,npt=NPFAC*ncof;
Number of points where function is evaluated, i.e., fineness of the mesh.
Doub devmax,e,hth,power,sum;
VecDoub bb(npt),coff(ncof),ee(npt),fs(npt),wt(npt),xs(npt);
MatDoub u(npt,ncof);
Ratfn ratbest(coff,mm+1,kk+1);
dev=BIG;
for (i=0;i<npt;i++) {                                Fill arrays with mesh abscissas and function val-
    if (i < (npt/2)-1) {                               ues.
        hth=PI02*i/(npt-1.0);                      At each end, use formula that minimizes round-
        xs[i]=a+(b-a)*SQR(sin(hth));      off sensitivity.
    } else {
        hth=PI02*(npt-i)/(npt-1.0);
        xs[i]=b-(b-a)*SQR(sin(hth));
    }
    fs[i]=fn(xs[i]);
    wt[i]=1.0;                                         In later iterations we will adjust these weights to
    ee[i]=1.0;                                         combat the largest deviations.
}
e=0.0;
for (it=0;it<MAXIT;it++) {                            Loop over iterations.
    for (i=0;i<npt;i++) {                            Set up the "design matrix" for the least-squares
        power=wt[i];                                 fit.
        bb[i]=power*(fs[i]+SIGN(e,ee[i]));
        Key idea here: Fit to  $fn(x) + e$  where the deviation is positive, to  $fn(x) - e$  where
        it is negative. Then  $e$  is supposed to become an approximation to the equal-ripple
        deviation.
        for (j=0;j<mm+1;j++) {
            u[i][j]=power;
            power *= xs[i];
        }
        power = -bb[i];
        for (j=mm+1;j<ncof;j++) {
            power *= xs[i];
            u[i][j]=power;
        }
    }
    SVD svd(u);                                     Singular value decomposition.
    svd.solve(bb,coff);
    In especially singular or difficult cases, one might here edit the singular values, replacing
    small values by zero in w[0..ncof-1].
    devmax=sum=0.0;
    Ratfn rat(coff,mm+1,kk+1);
    for (j=0;j<npt;j++) {                            Tabulate the deviations and revise the weights.
        ee[j]=rat(xs[j])-fs[j];
        wt[j]=abs(ee[j]);                           Use weighting to emphasize most deviant points.
        sum += wt[j];
        if (wt[j] > devmax) devmax=wt[j];
    }
    e=sum/npt;                                       Update  $e$  to be the mean absolute deviation.
    if (devmax <= dev) {                            Save only the best coefficient set found.
        ratbest = rat;
        dev=devmax;
    }
    cout << " ratlsq iteration= " << it;
    cout << " max error= " << setw(10) << devmax << endl;
}
return ratbest;
}

```

Figure 5.13.1 shows the discrepancies for the first five iterations of `ratlsq` when it is applied to find the  $m = k = 4$  rational fit to the function  $f(x) = \cos x / (1 + e^x)$  in the interval  $(0, \pi)$ . One sees that after the first iteration, the results are virtually as good as the



**Figure 5.13.1.** Solid curves show deviations  $r(x)$  for five successive iterations of the routine `ratlsq` for an arbitrary test problem. The algorithm does not converge to exactly the minimax solution (shown as the dotted curve). But, after one iteration, the discrepancy is a small fraction of the last significant bit of accuracy.

minimax solution. The iterations do not converge in the order that the figure suggests. In fact, it is the second iteration that is best (has smallest maximum deviation). The routine `ratlsq` accordingly returns the best of its iterations, not necessarily the last one; there is no advantage in doing more than five iterations.

#### CITED REFERENCES AND FURTHER READING:

Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), Chapter 13.[1]

## 5.14 Evaluation of Functions by Path Integration

In computer programming, the technique of choice is not necessarily the most efficient, or elegant, or fastest executing one. Instead, it may be the one that is quick to implement, general, and easy to check.

One sometimes needs only a few, or a few thousand, evaluations of a special function, perhaps a complex-valued function of a complex variable, that has many different parameters, or asymptotic regimes, or both. Use of the usual tricks (series, continued fractions, rational function approximations, recurrence relations, and so forth) may result in a patchwork program with tests and branches to different formulas. While such a program may be highly efficient in execution, it is often not the shortest way to the answer from a standing start.

A different technique of considerable generality is direct integration of a function's defining differential equation — an *ab initio* integration for each desired function value — along a path in the complex plane if necessary. While this may at first seem like swatting a fly with a golden brick, it turns out that when you already have the brick, and the fly is asleep right under it, all you have to do is let it fall!

As a specific example, let us consider the complex hypergeometric function  ${}_2F_1(a, b, c; z)$ , which is defined as the analytic continuation of the so-called hypergeometric series,

$$\begin{aligned} {}_2F_1(a, b, c; z) = 1 + \frac{ab}{c} \frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)} \frac{z^2}{2!} + \dots \\ + \frac{a(a+1)\dots(a+j-1)b(b+1)\dots(b+j-1)}{c(c+1)\dots(c+j-1)} \frac{z^j}{j!} + \dots \end{aligned} \quad (5.14.1)$$

The series converges only within the unit circle  $|z| < 1$  (see [1]), but one's interest in the function is often not confined to this region.

The hypergeometric function  ${}_2F_1$  is a solution (in fact *the* solution that is regular at the origin) of the hypergeometric differential equation, which we can write as

$$z(1-z)F'' = abF - [c - (a+b+1)z]F' \quad (5.14.2)$$

Here prime denotes  $d/dz$ . One can see that the equation has regular singular points at  $z = 0, 1$ , and  $\infty$ . Since the desired solution is regular at  $z = 0$ , the values  $1$  and  $\infty$  will in general be branch points. If we want  ${}_2F_1$  to be a single-valued function, we must have a branch cut connecting these two points. A conventional position for this cut is along the positive real axis from  $1$  to  $\infty$ , though we may wish to keep open the possibility of altering this choice for some applications.

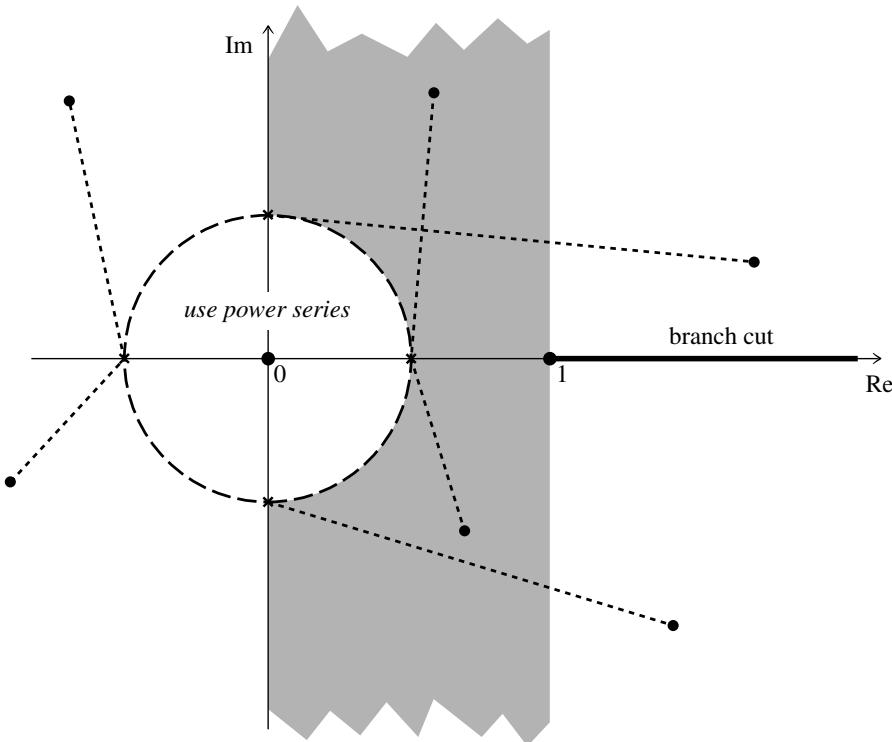
Our golden brick consists of a collection of routines for the integration of sets of ordinary differential equations, which we will develop in detail later, in Chapter 17. For now, we need only a high-level, “black-box” routine that integrates such a set from initial conditions at one value of a (real) independent variable to final conditions at some other value of the independent variable, while automatically adjusting its internal stepsize to maintain some specified accuracy. That routine is called `Odeint` and, in one particular invocation, it calculates its individual steps with a sophisticated Bulirsch-Stoer technique.

Suppose that we know values for  $F$  and its derivative  $F'$  at some value  $z_0$ , and that we want to find  $F$  at some other point  $z_1$  in the complex plane. The straight-line path connecting these two points is parametrized by

$$z(s) = z_0 + s(z_1 - z_0) \quad (5.14.3)$$

with  $s$  a real parameter. The differential equation (5.14.2) can now be written as a set of two first-order equations,

$$\begin{aligned} \frac{dF}{ds} &= (z_1 - z_0)F' \\ \frac{dF'}{ds} &= (z_1 - z_0) \left( \frac{abF - [c - (a+b+1)z]F'}{z(1-z)} \right) \end{aligned} \quad (5.14.4)$$



**Figure 5.14.1.** Complex plane showing the singular points of the hypergeometric function, its branch cut, and some integration paths from the circle  $|z| = 1/2$  (where the power series converges rapidly) to other points in the plane.

to be integrated from  $s = 0$  to  $s = 1$ . Here  $F$  and  $F'$  are to be viewed as two independent complex variables. The fact that prime means  $d/dz$  can be ignored; it will emerge as a consequence of the first equation in (5.14.4). Moreover, the real and imaginary parts of equation (5.14.4) define a set of four *real* differential equations, with independent variable  $s$ . The complex arithmetic on the right-hand side can be viewed as mere shorthand for how the four components are to be coupled. It is precisely this point of view that gets passed to the routine `Odeint`, since it knows nothing of either complex functions or complex independent variables.

It remains only to decide where to start, and what path to take in the complex plane, to get to an arbitrary point  $z$ . This is where consideration of the function's singularities, and the adopted branch cut, enter. Figure 5.14.1 shows the strategy that we adopt. For  $|z| \leq 1/2$ , the series in equation (5.14.1) will in general converge rapidly, and it makes sense to use it directly. Otherwise, we integrate along a straight-line path from one of the starting points  $(\pm 1/2, 0)$  or  $(0, \pm 1/2)$ . The former choices are natural for  $0 < \text{Re}(z) < 1$  and  $\text{Re}(z) < 0$ , respectively. The latter choices are used for  $\text{Re}(z) > 1$ , above and below the branch cut; the purpose of starting away from the real axis in these cases is to avoid passing too close to the singularity at  $z = 1$  (see Figure 5.14.1). The location of the branch cut is *defined* by the fact that our adopted strategy never integrates across the real axis for  $\text{Re}(z) > 1$ .

An implementation of this algorithm is given in §6.13 as the routine `hypgeo`.

A number of variants on the procedure described thus far are possible and easy to program. If successively called values of  $z$  are close together (with identical values of  $a$ ,  $b$ , and  $c$ ), then you can save the state vector  $(F, F')$  and the corresponding value of  $z$  on each call, and use these as starting values for the next call. The incremental integration may then take only one or two steps. Avoid integrating across the branch cut unintentionally: The function value will be “correct,” but not the one you want.

Alternatively, you may wish to integrate to some position  $z$  by a dog-leg path that *does* cross the real axis  $\text{Re}(z) > 1$ , as a means of *moving* the branch cut. For example, in some cases you might want to integrate from  $(0, 1/2)$  to  $(3/2, 1/2)$ , and go from there to any point with  $\text{Re}(z) > 1$  — with either sign of  $\text{Im}z$ . (If you are, for example, finding roots of a function by an iterative method, you do not want the integration for nearby values to take different paths around a branch point. If it does, your root-finder will see discontinuous function values and will likely not converge correctly!)

In any case, be aware that a loss of numerical accuracy can result if you integrate through a region of large function value on your way to a final answer where the function value is small. (For the hypergeometric function, a particular case of this is when  $a$  and  $b$  are both large and positive, with  $c$  and  $x \gtrsim 1$ .) In such cases, you’ll need to find a better dog-leg path.

The general technique of evaluating a function by integrating its differential equation in the complex plane can also be applied to other special functions. For example, the complex Bessel function, Airy function, Coulomb wave function, and Weber function are all special cases of the *confluent hypergeometric function*, with a differential equation similar to the one used above (see, e.g., [1] §13.6, for a table of special cases). The confluent hypergeometric function has no singularities at finite  $z$ : That makes it easy to integrate. However, its essential singularity at infinity means that it can have, along some paths and for some parameters, highly oscillatory or exponentially decreasing behavior: That makes it hard to integrate. Some case-by-case judgment (or experimentation) is therefore required.

#### CITED REFERENCES AND FURTHER READING:

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>.[1]

# Special Functions

## 6.0 Introduction

There is nothing particularly special about a *special function*, except that some person in authority or a textbook writer (not the same thing!) has decided to bestow the moniker. Special functions are sometimes called *higher transcendental functions* (higher than what?) or *functions of mathematical physics* (but they occur in other fields also) or *functions that satisfy certain frequently occurring second-order differential equations* (but not all special functions do). One might simply call them “useful functions” and let it go at that. The choice of which functions to include in this chapter is highly arbitrary.

Commercially available program libraries contain many special function routines that are intended for users who will have no idea what goes on inside them. Such state-of-the-art black boxes are often very messy things, full of branches to completely different methods depending on the value of the calling arguments. Black boxes have, or should have, careful control of accuracy, to some stated uniform precision in all regimes.

We will not be quite so fastidious in our examples, in part because we want to illustrate techniques from Chapter 5, and in part because we *want* you to understand what goes on in the routines presented. Some of our routines have an accuracy parameter that can be made as small as desired, while others (especially those involving polynomial fits) give only a certain stated accuracy, one that we believe is serviceable (usually, but not always, close to double precision). We do *not* certify that the routines are perfect black boxes. We do hope that, if you ever encounter trouble in a routine, you will be able to diagnose and correct the problem on the basis of the information that we have given.

In short, the special function routines of this chapter are meant to be used — we use them all the time — but we also want you to learn from their inner workings.

### CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>.

- Andrews, G.E., Askey, R., and Roy, R. 1999, *Special Functions* (Cambridge, UK: Cambridge University Press).
- Thompson, W.J. 1997, *Atlas for Computing Mathematical Functions* (New York: Wiley-Interscience).
- Spanier, J., and Oldham, K.B. 1987, *An Atlas of Functions* (Washington: Hemisphere Pub. Corp.).
- Wolfram, S. 2003, *The Mathematica Book*, 5th ed. (Champaign, IL: Wolfram Media).
- Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley).
- Hastings, C. 1955, *Approximations for Digital Computers* (Princeton: Princeton University Press).
- Luke, Y.L. 1975, *Mathematical Functions and Their Approximations* (New York: Academic Press).

## 6.1 Gamma Function, Beta Function, Factorials, Binomial Coefficients

The gamma function is defined by the integral

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt \quad (6.1.1)$$

When the argument  $z$  is an integer, the gamma function is just the familiar factorial function, but offset by 1:

$$n! = \Gamma(n + 1) \quad (6.1.2)$$

The gamma function satisfies the recurrence relation

$$\Gamma(z + 1) = z \Gamma(z) \quad (6.1.3)$$

If the function is known for arguments  $z > 1$  or, more generally, in the half complex plane  $\text{Re}(z) > 1$ , it can be obtained for  $z < 1$  or  $\text{Re}(z) < 1$  by the reflection formula

$$\Gamma(1 - z) = \frac{\pi}{\Gamma(z) \sin(\pi z)} = \frac{\pi z}{\Gamma(1 + z) \sin(\pi z)} \quad (6.1.4)$$

Notice that  $\Gamma(z)$  has a pole at  $z = 0$  and at all negative integer values of  $z$ .

There are a variety of methods in use for calculating the function  $\Gamma(z)$  numerically, but none is quite as neat as the approximation derived by Lanczos [1]. This scheme is entirely specific to the gamma function, seemingly plucked from thin air. We will not attempt to derive the approximation, but only state the resulting formula: For certain choices of rational  $\gamma$  and integer  $N$ , and for certain coefficients  $c_1, c_2, \dots, c_N$ , the gamma function is given by

$$\begin{aligned} \Gamma(z + 1) &= (z + \gamma + \frac{1}{2})^{z+\frac{1}{2}} e^{-(z+\gamma+\frac{1}{2})} \\ &\times \sqrt{2\pi} \left[ c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \epsilon \right] \quad (z > 0) \end{aligned} \quad (6.1.5)$$

You can see that this is a sort of take-off on Stirling's approximation, but with a series of corrections that take into account the first few poles in the left complex plane. The constant  $c_0$  is very nearly equal to 1. The error term is parametrized by

$\epsilon$ . For  $N = 14$ , and a certain set of  $c$ 's and  $\gamma$  (calculated by P. Godfrey), the error is smaller than  $|\epsilon| < 10^{-15}$ . Even more impressive is the fact that, with these same constants, the formula (6.1.5) applies for the *complex* gamma function, *everywhere in the half complex plane*  $\operatorname{Re} z > 0$ , achieving almost the same accuracy as on the real line.

It is better to implement  $\ln \Gamma(x)$  than  $\Gamma(x)$ , since the latter will overflow at quite modest values of  $x$ . Often the gamma function is used in calculations where the large values of  $\Gamma(x)$  are divided by other large numbers, with the result being a perfectly ordinary value. Such operations would normally be coded as subtraction of logarithms. With (6.1.5) in hand, we can compute the logarithm of the gamma function with two calls to a logarithm and a few dozen arithmetic operations. This makes it not much more difficult than other built-in functions that we take for granted, such as  $\sin x$  or  $e^x$ :

```
Doub gammln(const Doub xx) {
    Returns the value  $\ln[\Gamma(xx)]$  for  $xx > 0$ .
    Int j;
    Doub x,tmp,y,ser;
    static const Doub cof[14]={57.1562356658629235,-59.5979603554754912,
        14.1360979747417471,-0.491913816097620199,.339946499848118887e-4,
        .465236289270485756e-4,-.983744753048795646e-4,.158088703224912494e-3,
        -.210264441724104883e-3,.217439618115212643e-3,-.164318106536763890e-3,
        .844182239838527433e-4,-.261908384015814087e-4,.368991826595316234e-5};
    if (xx <= 0) throw("bad arg in gammln");
    y=x==xx;
    tmp = x+5.242187500000000000;           Rational 671/128.
    tmp = (x+0.5)*log(tmp)-tmp;
    ser = 0.99999999999997092;
    for (j=0;j<14;j++) ser += cof[j]/++y;
    return tmp+log(2.5066282746310005*ser/x);
}
```

gamma.h

How shall we write a routine for the factorial function  $n!$ ? Generally the factorial function will be called for small integer values, and in most applications the same integer value will be called for many times. It is obviously inefficient to call  $\exp(\text{gammln}(n+1))$  for each required factorial. Better is to initialize a static table on the first call, and do a fast lookup on subsequent calls. The fixed size 171 for the table is because  $170!$  is representable as an IEEE double precision value, but  $171!$  overflows. It is also sometimes useful to know that factorials up to  $22!$  have exact double precision representations (52 bits of mantissa, not counting powers of two that are absorbed into the exponent), while  $23!$  and above are represented only approximately.

```
Doub factrl(const Int n) {
    Returns the value  $n!$  as a floating-point number.
    static VecDoub a(171);
    static Bool init=true;
    if (init) {
        init = false;
        a[0] = 1.;
        for (Int i=1;i<171;i++) a[i] = i*a[i-1];
    }
    if (n < 0 || n > 170) throw("factrl out of range");
    return a[n];
}
```

gamma.h

More useful in practice is a function returning the log of a factorial, which doesn't have overflow issues. The size of the table of logarithms is whatever you can afford in space and initialization time. The value  $\text{NTOP} = 2000$  should be increased if your integer arguments are often larger.

```
gamma.h Doub factln(const Int n) {
    Returns ln(n!).
    static const Int NTOP=2000;
    static VecDoub a(NTOP);
    static Bool init=true;
    if (init) {
        init = false;
        for (Int i=0;i<NTOP;i++) a[i] = gammln(i+1.);
    }
    if (n < 0) throw("negative arg in factln");
    if (n < NTOP) return a[n];
    return gammln(n+1.);           Out of range of table.
}
```

The binomial coefficient is defined by

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad 0 \leq k \leq n \quad (6.1.6)$$

A routine that takes advantage of the tables stored in `factrl` and `factln` is

```
gamma.h Doub bico(const Int n, const Int k) {
    Returns the binomial coefficient  $\binom{n}{k}$  as a floating-point number.
    if (n<0 || k<0 || k>n) throw("bad args in bico");
    if (n<171) return floor(0.5*factrl(n)/(factrl(k)*factrl(n-k)));
    return floor(0.5+exp(factln(n)-factln(k)-factln(n-k)));
    The floor function cleans up roundoff error for smaller values of n and k.
}
```

If your problem requires a series of related binomial coefficients, a good idea is to use recurrence relations, for example,

$$\begin{aligned} \binom{n+1}{k} &= \frac{n+1}{n-k+1} \binom{n}{k} = \binom{n}{k} + \binom{n}{k-1} \\ \binom{n}{k+1} &= \frac{n-k}{k+1} \binom{n}{k} \end{aligned} \quad (6.1.7)$$

Finally, turning away from the combinatorial functions with integer-valued arguments, we come to the beta function,

$$B(z, w) = B(w, z) = \int_0^1 t^{z-1} (1-t)^{w-1} dt \quad (6.1.8)$$

which is related to the gamma function by

$$B(z, w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \quad (6.1.9)$$

hence

```
Doub beta(const Doub z, const Doub w) {
    Returns the value of the beta function  $B(z, w)$ .
    return exp(gammln(z)+gammln(w)-gammln(z+w));
}
```

gamma.h

**CITED REFERENCES AND FURTHER READING:**

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, Chapter 6.

Lanczos, C. 1964, “A Precision Approximation of the Gamma Function,” *SIAM Journal on Numerical Analysis*, ser. B, vol. 1, pp. 86–96.[1]

## 6.2 Incomplete Gamma Function and Error Function

The incomplete gamma function is defined by

$$P(a, x) \equiv \frac{\gamma(a, x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt \quad (a > 0) \quad (6.2.1)$$

It has the limiting values

$$P(a, 0) = 0 \quad \text{and} \quad P(a, \infty) = 1 \quad (6.2.2)$$

The incomplete gamma function  $P(a, x)$  is monotonic and (for  $a$  greater than one or so) rises from “near-zero” to “near-unity” in a range of  $x$  centered on about  $a - 1$ , and of width about  $\sqrt{a}$  (see Figure 6.2.1).

The complement of  $P(a, x)$  is also confusingly called an incomplete gamma function,

$$Q(a, x) \equiv 1 - P(a, x) \equiv \frac{\Gamma(a, x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_x^\infty e^{-t} t^{a-1} dt \quad (a > 0) \quad (6.2.3)$$

It has the limiting values

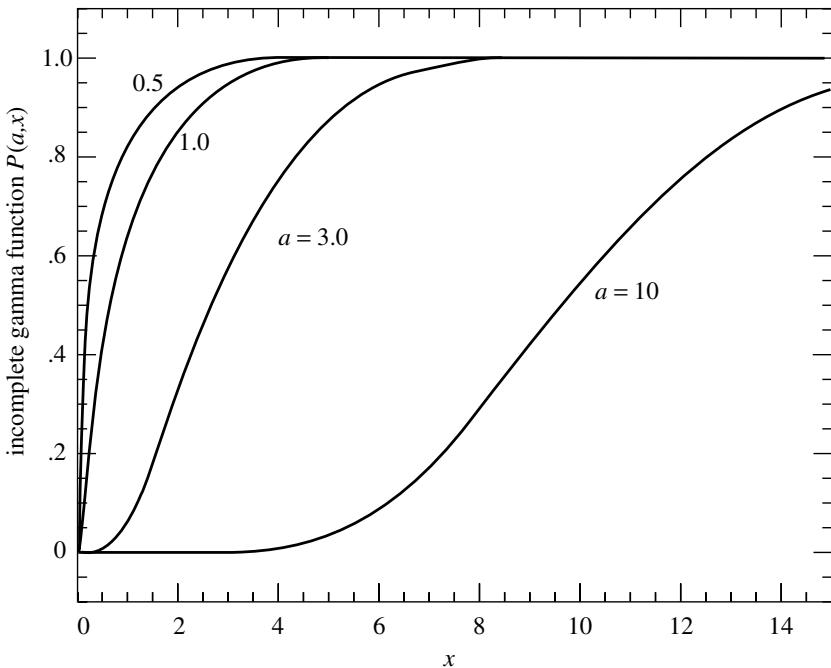
$$Q(a, 0) = 1 \quad \text{and} \quad Q(a, \infty) = 0 \quad (6.2.4)$$

The notations  $P(a, x)$ ,  $\gamma(a, x)$ , and  $\Gamma(a, x)$  are standard; the notation  $Q(a, x)$  is specific to this book.

There is a series development for  $\gamma(a, x)$  as follows:

$$\gamma(a, x) = e^{-x} x^a \sum_{n=0}^{\infty} \frac{\Gamma(a)}{\Gamma(a + 1 + n)} x^n \quad (6.2.5)$$

One does not actually need to compute a new  $\Gamma(a + 1 + n)$  for each  $n$ ; one rather uses equation (6.1.3) and the previous coefficient.



**Figure 6.2.1.** The incomplete gamma function  $P(a, x)$  for four values of  $a$ .

A continued fraction development for  $\Gamma(a, x)$  is

$$\Gamma(a, x) = e^{-x} x^a \left( \frac{1}{x +} \frac{1-a}{1+} \frac{1}{x+} \frac{2-a}{1+} \frac{2}{x+} \dots \right) \quad (x > 0) \quad (6.2.6)$$

It is computationally better to use the even part of (6.2.6), which converges twice as fast (see §5.2):

$$\Gamma(a, x) = e^{-x} x^a \left( \frac{1}{x+1-a-} \frac{1 \cdot (1-a)}{x+3-a-} \frac{2 \cdot (2-a)}{x+5-a-} \dots \right) \quad (x > 0) \quad (6.2.7)$$

It turns out that (6.2.5) converges rapidly for  $x$  less than about  $a + 1$ , while (6.2.6) or (6.2.7) converges rapidly for  $x$  greater than about  $a + 1$ . In these respective regimes each requires at most a few times  $\sqrt{a}$  terms to converge, and this many only near  $x = a$ , where the incomplete gamma functions are varying most rapidly. For moderate values of  $a$ , less than 100, say, (6.2.5) and (6.2.7) together allow evaluation of the function for all  $x$ . An extra dividend is that we never need to compute a function value near zero by subtracting two nearly equal numbers.

Some applications require  $P(a, x)$  and  $Q(a, x)$  for much larger values of  $a$ , where both the series and the continued fraction are inefficient. In this regime, however, the integrand in equation (6.2.1) falls off sharply in both directions from its peak, within a few times  $\sqrt{a}$ . An efficient procedure is to evaluate the integral directly, with a single step of high-order Gauss-Legendre quadrature (§4.6) extending from  $x$  just far enough into the nearest tail to achieve negligible values of the integrand. Actually it is “half a step,” because we need the dense abscissas only near  $x$ , not far out on the tail where the integrand is effectively zero.

We package the various incomplete gamma parts into an object `Gamma`. The only persistent state is the value `gln`, which is set to  $\Gamma(a)$  for the most recent call to  $P(a, x)$  or  $Q(a, x)$ . This is useful when you need a different normalization convention, for example  $\gamma(a, x)$  or  $\Gamma(a, x)$  in equations (6.2.1) or (6.2.3).

```
struct Gamma : Gauleg18 {
Object for incomplete gamma function. Gauleg18 provides coefficients for Gauss-Legendre quadrature.
    static const Int ASWITCH=100;           When to switch to quadrature method.
    static const Doub EPS;                 See end of struct for initializations.
    static const Doub FPMIN;
    Doub gln;

Doub gammmp(const Doub a, const Doub x) {
Returns the incomplete gamma function  $P(a, x)$ .
    if (x < 0.0 || a <= 0.0) throw("bad args in gammmp");
    if (x == 0.0) return 0.0;
    else if ((Int)a >= ASWITCH) return gammppapprox(a,x,1); Quadrature.
    else if (x < a+1.0) return gser(a,x);      Use the series representation.
    else return 1.0-gcf(a,x);                  Use the continued fraction representation.
}

Doub gammq(const Doub a, const Doub x) {
Returns the incomplete gamma function  $Q(a, x) \equiv 1 - P(a, x)$ .
    if (x < 0.0 || a <= 0.0) throw("bad args in gammq");
    if (x == 0.0) return 1.0;
    else if ((Int)a >= ASWITCH) return gammppapprox(a,x,0); Quadrature.
    else if (x < a+1.0) return 1.0-gser(a,x);      Use the series representation.
    else return gcf(a,x);                      Use the continued fraction representation.
}

Doub gser(const Doub a, const Doub x) {
Returns the incomplete gamma function  $P(a, x)$  evaluated by its series representation.
Also sets  $\ln \Gamma(a)$  as gln. User should not call directly.
    Doub sum,del,ap;
    gln=gammln(a);
    ap=a;
    del=sum=1.0/a;
    for (;;) {
        ++ap;
        del *= x/ap;
        sum += del;
        if (fabs(del) < fabs(sum)*EPS) {
            return sum*exp(-x+a*log(x)-gln);
        }
    }
}

Doub gcf(const Doub a, const Doub x) {
Returns the incomplete gamma function  $Q(a, x)$  evaluated by its continued fraction representation.
Also sets  $\ln \Gamma(a)$  as gln. User should not call directly.
    Int i;
    Doub an,b,c,d,del,h;
    gln=gammln(a);
    b=x+1.0-a;
    c=1.0/FPMIN;
    d=1.0/b;
    h=d;
    for (i=1;;i++) {
        an = -i*(i-a);
        b += 2.0;
        d=an*d+b;
        Set up for evaluating continued fraction
        by modified Lentz's method (§5.2)
        with  $b_0 = 0$ .
        Iterate to convergence.
    }
}
```

```

        if (fabs(d) < FPMIN) d=FPMIN;
        c=b+an/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        del=d*c;
        h *= del;
        if (fabs(del-1.0) <= EPS) break;
    }
    return exp(-x+a*log(x)-gln)*h;           Put factors in front.
}

Doub gammapprox(Doub a, Doub x, Int psig) {
Incomplete gamma by quadrature. Returns  $P(a, x)$  or  $Q(a, x)$ , when psig is 1 or 0,
respectively. User should not call directly.
    Int j;
    Doub xu,t,sum,ans;
    Doub a1 = a-1.0, lna1 = log(a1), sqrta1 = sqrt(a1);
    gln = gammeln(a);
    Set how far to integrate into the tail:
    if (x > a1) xu = MAX(a1 + 11.5*sqrt(a1), x + 6.0*sqrt(a1));
    else xu = MAX(0.,MIN(a1 - 7.5*sqrt(a1), x - 5.0*sqrt(a1)));
    sum = 0;
    for (j=0;j<ngau;j++) {                           Gauss-Legendre.
        t = x + (xu-x)*y[j];
        sum += w[j]*exp(-(t-a1)+a1*(log(t)-lna1));
    }
    ans = sum*(xu-x)*exp(a1*(lna1-1.)-gln);
    return (psig?(ans>0.0? 1.0-ans:-ans):(ans>=0.0? ans:1.0+ans));
}

Doub invgammp(Doub p, Doub a);
Inverse function on  $x$  of  $P(a, x)$ . See §6.2.1.

};

const Doub Gamma::EPS = numeric_limits<Doub>::epsilon();
const Doub Gamma::FPMIN = numeric_limits<Doub>::min()/EPS;

```

Remember that since `Gamma` is an object, you have to declare an instance of it before you can use its member functions. We habitually write

```
Gamma gam;
```

as a global declaration, and then call `gam.gammp` or `gam.gammq` as needed. The structure `Gauleg18` just contains the abscissas and weights for the Gauss-Legendre quadrature.

`ncgammabeta.h`

```

struct Gauleg18 {
Abscissas and weights for Gauss-Legendre quadrature.
    static const Int ngau = 18;
    static const Doub y[18];
    static const Doub w[18];
};
const Doub Gauleg18::y[18] = {0.0021695375159141994,
0.011413521097787704,0.027972308950302116,0.051727015600492421,
0.082502225484340941, 0.12007019910960293,0.16415283300752470,
0.21442376986779355, 0.27051082840644336, 0.33199876341447887,
0.39843234186401943, 0.46931971407375483, 0.54413605556657973,
0.62232745288031077, 0.70331500465597174, 0.78649910768313447,
0.87126389619061517, 0.95698180152629142};
const Doub Gauleg18::w[18] = {0.0055657196642445571,
0.012915947284065419,0.020181515297735382,0.027298621498568734,
0.034213810770299537,0.040875750923643261,0.047235083490265582,
0.053244713977759692,0.058860144245324798,0.064039797355015485,

```

```
0.068745323835736408, 0.072941885005653087, 0.076598410645870640,
0.079687828912071670, 0.082187266704339706, 0.084078218979661945,
0.085346685739338721, 0.085983275670394821};
```

### 6.2.1 Inverse Incomplete Gamma Function

In many statistical applications one needs the inverse of the incomplete gamma function, that is, the value  $x$  such that  $P(a, x) = p$ , for a given value  $0 \leq p \leq 1$ . Newton's method works well if we can devise a good-enough initial guess. In fact, this is a good place to use Halley's method (see §9.4), since the second derivative (that is, the first derivative of the integrand) is easy to compute.

For  $a > 1$ , we use an initial guess that derives from §26.2.22 and §26.4.17 in reference [1]. For  $a \leq 1$ , we first roughly approximate  $P_a \equiv P(a, 1)$ :

$$P_a \equiv P(a, 1) \approx 0.253a + 0.12a^2, \quad 0 \leq a \leq 1 \quad (6.2.8)$$

and then solve for  $x$  in one or the other of the (rough) approximations:

$$P(a, x) \approx \begin{cases} P_a x^a, & x < 1 \\ P_a + (1 - P_a)(1 - e^{1-x}), & x \geq 1 \end{cases} \quad (6.2.9)$$

An implementation is

```
Doub Gamma::invgammp(Doub p, Doub a) {
    Returns x such that P(a,x) = p for an argument p between 0 and 1.
    Int j;
    Doub x,err,t,u,pp,lna1,afac,a1=a-1;
    const Doub EPS=1.e-8;                                Accuracy is the square of EPS.
    gln=gammaln(a);
    if (a <= 0.) throw("a must be pos in invgammap");
    if (p >= 1.) return MAX(100.,a + 100.*sqrt(a));
    if (p <= 0.) return 0.0;
    if (a > 1.) {                                       Initial guess based on reference [1].
        lna1=log(a1);
        afac = exp(a1*(lna1-1.)-gln);
        pp = (p < 0.5)? p : 1. - p;
        t = sqrt(-2.*log(pp));
        x = (2.30753+t*0.27061)/(1.+t*(0.99229+t*0.04481)) - t;
        if (p < 0.5) x = -x;
        x = MAX(1.e-3,a*pow(1.-1./(9.*a)-x/(3.*sqrt(a)),3));
    } else {                                         Initial guess based on equations (6.2.8)
        t = 1.0 - a*(0.253+a*0.12);                  and (6.2.9).
        if (p < t) x = pow(p/t,1./a);
        else x = 1.-log(1.-(p-t)/(1.-t));
    }
    for (j=0;j<12;j++) {
        if (x <= 0.0) return 0.0;                      x too small to compute accurately.
        err = gammp(a,x) - p;
        if (a > 1.) t = afac*exp(-(x-a1)+a1*(log(x)-lna1));
        else t = exp(-x+a1*log(x)-gln);
        u = err/t;
        x -= (t = u/(1.-0.5*MIN(1.,u*((a-1.)/x - 1.)));) ;      Halley's method.
        if (x <= 0.) x = 0.5*(x + t);          Halve old value if x tries to go negative.
        if (fabs(t) < EPS*x ) break;
    }
    return x;
}
```

incgammabeta.h

## 6.2.2 Error Function

The error function and complementary error function are special cases of the incomplete gamma function and are obtained moderately efficiently by the above procedures. Their definitions are

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad (6.2.10)$$

and

$$\operatorname{erfc}(x) \equiv 1 - \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt \quad (6.2.11)$$

The functions have the following limiting values and symmetries:

$$\operatorname{erf}(0) = 0 \quad \operatorname{erf}(\infty) = 1 \quad \operatorname{erf}(-x) = -\operatorname{erf}(x) \quad (6.2.12)$$

$$\operatorname{erfc}(0) = 1 \quad \operatorname{erfc}(\infty) = 0 \quad \operatorname{erfc}(-x) = 2 - \operatorname{erfc}(x) \quad (6.2.13)$$

They are related to the incomplete gamma functions by

$$\operatorname{erf}(x) = P\left(\frac{1}{2}, x^2\right) \quad (x \geq 0) \quad (6.2.14)$$

and

$$\operatorname{erfc}(x) = Q\left(\frac{1}{2}, x^2\right) \quad (x \geq 0) \quad (6.2.15)$$

A faster calculation takes advantage of an approximation of the form

$$\operatorname{erfc}(z) \approx t \exp[-z^2 + \mathcal{P}(t)], \quad z > 0 \quad (6.2.16)$$

where

$$t \equiv \frac{2}{2+z} \quad (6.2.17)$$

and  $\mathcal{P}(t)$  is a polynomial for  $0 \leq t \leq 1$  that can be found by Chebyshev methods (§5.8). As with Gamma, implementation is by an object that also includes the inverse function, here an inverse for both erf and erfc. Halley's method is again used for the inverses (as suggested by P.J. Acklam).

```
erf.h struct Erf {
Object for error function and related functions.
    static const Int ncof=28;
    static const Doub cof[28];           Initialization at end of struct.

    inline Doub erf(Doub x) {
        Return erf(x) for any x.
        if (x >= 0.) return 1.0 - erfccheb(x);
        else return erfccheb(-x) - 1.0;
    }

    inline Doub erfc(Doub x) {
        Return erfc(x) for any x.
        if (x >= 0.) return erfccheb(x);
        else return 2.0 - erfccheb(-x);
    }
}
```

```

Doub erfccheb(Doub z){
Evaluate equation (6.2.16) using stored Chebyshev coefficients. User should not call directly.
    Int j;
    Doub t,ty,tmp,d=0.,dd=0.;
    if (z < 0.) throw("erfccheb requires nonnegative argument");
    t = 2./(2.+z);
    ty = 4.*t - 2.;
    for (j=ncof-1;j>0;j--) {
        tmp = d;
        d = ty*d - dd + cof[j];
        dd = tmp;
    }
    return t*exp(-z*z + 0.5*(cof[0] + ty*d) - dd);
}

Doub inverfc(Doub p) {
Inverse of complementary error function. Returns  $x$  such that  $\text{erfc}(x) = p$  for argument  $p$  between 0 and 2.
    Doub x,err,t,pp;
    if (p >= 2.0) return -100.;           Return arbitrary large pos or neg value.
    if (p <= 0.0) return 100.;
    pp = (p < 1.0)? p : 2. - p;
    t = sqrt(-2.*log(pp/2.));           Initial guess:
    x = -0.70711*((2.30753+t*0.27061)/(1.+t*(0.99229+t*0.04481)) - t);
    for (Int j=0;j<2;j++) {
        err = erfc(x) - pp;
        x += err/(1.12837916709551257*exp(-SQR(x))-x*err); Halley.
    }
    return (p < 1.0? x : -x);
}

inline Doub inverf(Doub p) {return inverfc(1.-p);}
Inverse of the error function. Returns  $x$  such that  $\text{erf}(x) = p$  for argument  $p$  between -1 and 1.

};

const Doub Erf::cof[28] = {-1.3026537197817094, 6.4196979235649026e-1,
1.9476473204185836e-2,-9.561514786808631e-3,-9.46595344482036e-4,
3.66839497852761e-4,4.2523324806907e-5,-2.0278578112534e-5,
-1.624290004647e-6,1.3036558355580e-6,1.5626441722e-8,-8.5238095915e-8,
6.529054439e-9,5.059343495e-9,-9.91364156e-10,-2.27365122e-10,
9.6467911e-11, 2.394038e-12,-6.886027e-12,8.94487e-13, 3.13092e-13,
-1.12708e-13,3.81e-16,7.106e-15,-1.523e-15,-9.4e-17,1.21e-16,-2.8e-17};

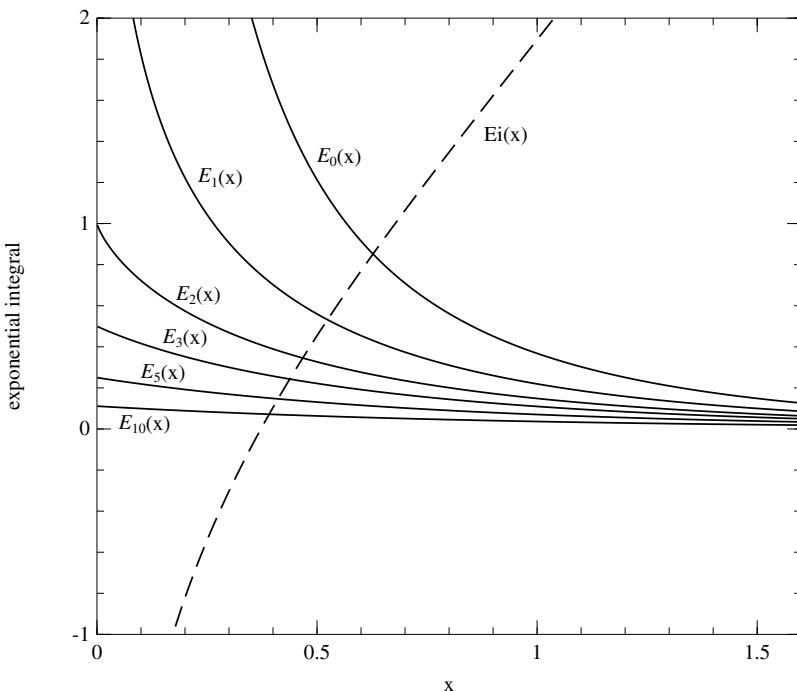
```

A lower-order Chebyshev approximation produces a very concise routine, though with only about single precision accuracy:

```

Doub erfcc(const Doub x)                                erf.h
Returns the complementary error function  $\text{erfc}(x)$  with fractional error everywhere less than
 $1.2 \times 10^{-7}$ .
{
    Doub t,z=abs(x),ans;
    t=2./(2.+z);
    ans=t*exp(-z*z-1.26551223+t*(1.00002368+t*(0.37409196+t*(0.09678418+
        t*(-0.18628806+t*(0.27886807+t*(-1.13520398+t*(1.48851587+
            t*(-0.82215223+t*0.17087277)))))));
    return (x >= 0.0 ? ans : 2.0-ans);
}

```



**Figure 6.3.1.** Exponential integrals  $E_n(x)$  for  $n = 0, 1, 2, 3, 5$ , and  $10$ , and the exponential integral  $Ei(x)$ .

#### CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, Chapters 6, 7, and 26.[1]

Pearson, K. (ed.) 1951, *Tables of the Incomplete Gamma Function* (Cambridge, UK: Cambridge University Press).

## 6.3 Exponential Integrals

The standard definition of the exponential integral is

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} dt, \quad x > 0, \quad n = 0, 1, \dots \quad (6.3.1)$$

The function defined by the principal value of the integral

$$Ei(x) = - \int_{-\infty}^{\infty} \frac{e^{-t}}{t} dt = \int_{-\infty}^x \frac{e^t}{t} dt, \quad x > 0 \quad (6.3.2)$$

is also called an exponential integral. Note that  $Ei(-x)$  is related to  $-E_1(x)$  by analytic continuation. Figure 6.3.1 plots these functions for representative values of their parameters.

The function  $E_n(x)$  is a special case of the incomplete gamma function

$$E_n(x) = x^{n-1} \Gamma(1-n, x) \quad (6.3.3)$$

We can therefore use a similar strategy for evaluating it. The continued fraction — just equation (6.2.6) rewritten — converges for all  $x > 0$ :

$$E_n(x) = e^{-x} \left( \frac{1}{x+} \frac{n}{1+} \frac{1}{x+} \frac{n+1}{1+} \frac{2}{x+} \dots \right) \quad (6.3.4)$$

We use it in its more rapidly converging even form,

$$E_n(x) = e^{-x} \left( \frac{1}{x+n-} \frac{1 \cdot n}{x+n+2-} \frac{2(n+1)}{x+n+4-} \dots \right) \quad (6.3.5)$$

The continued fraction only really converges fast enough to be useful for  $x \gtrsim 1$ . For  $0 < x \lesssim 1$ , we can use the series representation

$$E_n(x) = \frac{(-x)^{n-1}}{(n-1)!} [-\ln x + \psi(n)] - \sum_{\substack{m=0 \\ m \neq n-1}}^{\infty} \frac{(-x)^m}{(m-n+1)m!} \quad (6.3.6)$$

The quantity  $\psi(n)$  here is the digamma function, given for integer arguments by

$$\psi(1) = -\gamma, \quad \psi(n) = -\gamma + \sum_{m=1}^{n-1} \frac{1}{m} \quad (6.3.7)$$

where  $\gamma = 0.5772156649\dots$  is Euler's constant. We evaluate the expression (6.3.6) in order of ascending powers of  $x$ :

$$\begin{aligned} E_n(x) = & - \left[ \frac{1}{(1-n)} - \frac{x}{(2-n) \cdot 1} + \frac{x^2}{(3-n)(1 \cdot 2)} - \dots + \frac{(-x)^{n-2}}{(-1)(n-2)!} \right] \\ & + \frac{(-x)^{n-1}}{(n-1)!} [-\ln x + \psi(n)] - \left[ \frac{(-x)^n}{1 \cdot n!} + \frac{(-x)^{n+1}}{2 \cdot (n+1)!} + \dots \right] \end{aligned} \quad (6.3.8)$$

The first square bracket is omitted when  $n = 1$ . This method of evaluation has the advantage that, for large  $n$ , the series converges before reaching the term containing  $\psi(n)$ . Accordingly, one needs an algorithm for evaluating  $\psi(n)$  only for small  $n$ ,  $n \lesssim 20 - 40$ . We use equation (6.3.7), although a table lookup would improve efficiency slightly.

Amos [1] presents a careful discussion of the truncation error in evaluating equation (6.3.8) and gives a fairly elaborate termination criterion. We have found that simply stopping when the last term added is smaller than the required tolerance works about as well.

Two special cases have to be handled separately:

$$\begin{aligned} E_0(x) &= \frac{e^{-x}}{x} \\ E_n(0) &= \frac{1}{n-1}, \quad n > 1 \end{aligned} \quad (6.3.9)$$

The routine `expint` allows fast evaluation of  $E_n(x)$  to any accuracy EPS within the reach of your machine's precision for floating-point numbers. The only modification required for increased accuracy is to supply Euler's constant with enough significant digits. Wrench [2] can provide you with the first 328 digits if necessary!

```
expint.h Doub expint(const Int n, const Doub x)
Evaluates the exponential integral  $E_n(x)$ .
{
    static const Int MAXIT=100;
    static const Doub EULER=0.577215664901533,
        EPS=numeric_limits<Doub>::epsilon(),
        BIG=numeric_limits<Doub>::max()*EPS;
    Here MAXIT is the maximum allowed number of iterations; EULER is Euler's constant
     $\gamma$ ; EPS is the desired relative error, not smaller than the machine precision; BIG is a
    number near the largest representable floating-point number.

    Int i,ii,nm1=n-1;
    Doub a,b,c,d,del,fact,h,psi,ans;
    if (n < 0 || x < 0.0 || (x==0.0 && (n==0 || n==1)))
        throw("bad arguments in expint");
    if (n == 0) ans=exp(-x)/x;                                Special case.
    else {
        if (x == 0.0) ans=1.0/nm1;                            Another special case.
        else {
            if (x > 1.0) {                                    Lentz's algorithm (§5.2).
                b=x+n;
                c=BIG;
                d=1.0/b;
                h=d;
                for (i=1;i<=MAXIT;i++) {
                    a = -i*(nm1+i);
                    b += 2.0;
                    d=1.0/(a*d+b);                          Denominators cannot be zero.
                    c=b+a/c;
                    del=c*d;
                    h *= del;
                    if (abs(del-1.0) <= EPS) {
                        ans=h*exp(-x);
                        return ans;
                    }
                }
                throw("continued fraction failed in expint");
            } else {                                         Evaluate series.
                ans = (nm1!=0 ? 1.0/nm1 : -log(x)-EULER);      Set first term.
                fact=1.0;
                for (i=1;i<=MAXIT;i++) {
                    fact *= -x/i;
                    if (i != nm1) del = -fact/(i-nm1);
                    else {
                        psi = -EULER;                         Compute  $\psi(n)$ .
                        for (ii=1;ii<=nm1;ii++) psi += 1.0/ii;
                        del=fact*(-log(x)+psi);
                    }
                    ans += del;
                    if (abs(del) < abs(ans)*EPS) return ans;
                }
                throw("series failed in expint");
            }
        }
    }
    return ans;
}
```

A good algorithm for evaluating Ei is to use the power series for small  $x$  and the asymptotic series for large  $x$ . The power series is

$$\text{Ei}(x) = \gamma + \ln x + \frac{x}{1 \cdot 1!} + \frac{x^2}{2 \cdot 2!} + \dots \quad (6.3.10)$$

where  $\gamma$  is Euler's constant. The asymptotic expansion is

$$\text{Ei}(x) \sim \frac{e^x}{x} \left( 1 + \frac{1!}{x} + \frac{2!}{x^2} + \dots \right) \quad (6.3.11)$$

The lower limit for the use of the asymptotic expansion is approximately  $|\ln \text{EPS}|$ , where EPS is the required relative error.

```
Doub ei(const Doub x) { expint.h
    Computes the exponential integral Ei(x) for x > 0.
    static const Int MAXIT=100;
    static const Doub EULER=0.577215664901533,
        EPS=numeric_limits<Doub>::epsilon(),
        FPMIN=numeric_limits<Doub>::min()/EPS;
    Here MAXIT is the maximum number of iterations allowed; EULER is Euler's constant γ; EPS
    is the relative error, or absolute error near the zero of Ei at x = 0.3725; FPMIN is a number
    close to the smallest representable floating-point number.
    Int k;
    Doub fact,prev,sum,term;
    if (x <= 0.0) throw("Bad argument in ei");
    if (x < FPMIN) return log(x)+EULER;           Special case: Avoid failure of convergence
                                                    test because of underflow.
    if (x <= -log(EPS)) {
        sum=0.0;                                     Use power series.
        fact=1.0;
        for (k=1;k<=MAXIT;k++) {
            fact *= x/k;
            term=fact/k;
            sum += term;
            if (term < EPS*sum) break;
        }
        if (k > MAXIT) throw("Series failed in ei");
        return sum+log(x)+EULER;
    } else {                                         Use asymptotic series.
        sum=0.0;                                     Start with second term.
        term=1.0;
        for (k=1;k<=MAXIT;k++) {
            prev=term;
            term *= k/x;
            if (term < EPS) break;
        Since final sum is greater than one, term itself approximates the relative error.
            if (term < prev) sum += term;           Still converging: Add new term.
            else {
                sum -= prev;                      Diverging: Subtract previous term and
                break;                           exit.
            }
        }
        return exp(x)*(1.0+sum)/x;
    }
}
```

#### CITED REFERENCES AND FURTHER READING:

Stegun, I.A., and Zucker, R. 1974, "Automatic Computing Methods for Special Functions. II. The Exponential Integral  $E_n(x)$ ," *Journal of Research of the National Bureau of Standards*,

- vol. 78B, pp. 199–216; 1976, “Automatic Computing Methods for Special Functions. III. The Sine, Cosine, Exponential Integrals, and Related Functions,” *op. cit.*, vol. 80B, pp. 291–311.
- Amos D.E. 1980, “Computation of Exponential Integrals,” *ACM Transactions on Mathematical Software*, vol. 6, pp. 365–377[1]; also vol. 6, pp. 420–428.
- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, Chapter 5.
- Wrench J.W. 1952, “A New Calculation of Euler’s Constant,” *Mathematical Tables and Other Aids to Computation*, vol. 6, p. 255.[2]

## 6.4 Incomplete Beta Function

The incomplete beta function is defined by

$$I_x(a, b) \equiv \frac{B_x(a, b)}{B(a, b)} \equiv \frac{1}{B(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} dt \quad (a, b > 0) \quad (6.4.1)$$

It has the limiting values

$$I_0(a, b) = 0 \quad I_1(a, b) = 1 \quad (6.4.2)$$

and the symmetry relation

$$I_x(a, b) = 1 - I_{1-x}(b, a) \quad (6.4.3)$$

If  $a$  and  $b$  are both rather greater than one, then  $I_x(a, b)$  rises from “near-zero” to “near-unity” quite sharply at about  $x = a/(a+b)$ . Figure 6.4.1 plots the function for several pairs  $(a, b)$ .

The incomplete beta function has a series expansion

$$I_x(a, b) = \frac{x^a (1-x)^b}{a B(a, b)} \left[ 1 + \sum_{n=0}^{\infty} \frac{B(a+1, n+1)}{B(a+b, n+1)} x^{n+1} \right] \quad (6.4.4)$$

but this does not prove to be very useful in its numerical evaluation. (Note, however, that the beta functions in the coefficients can be evaluated for each value of  $n$  with just the previous value and a few multiplies, using equations 6.1.9 and 6.1.3.)

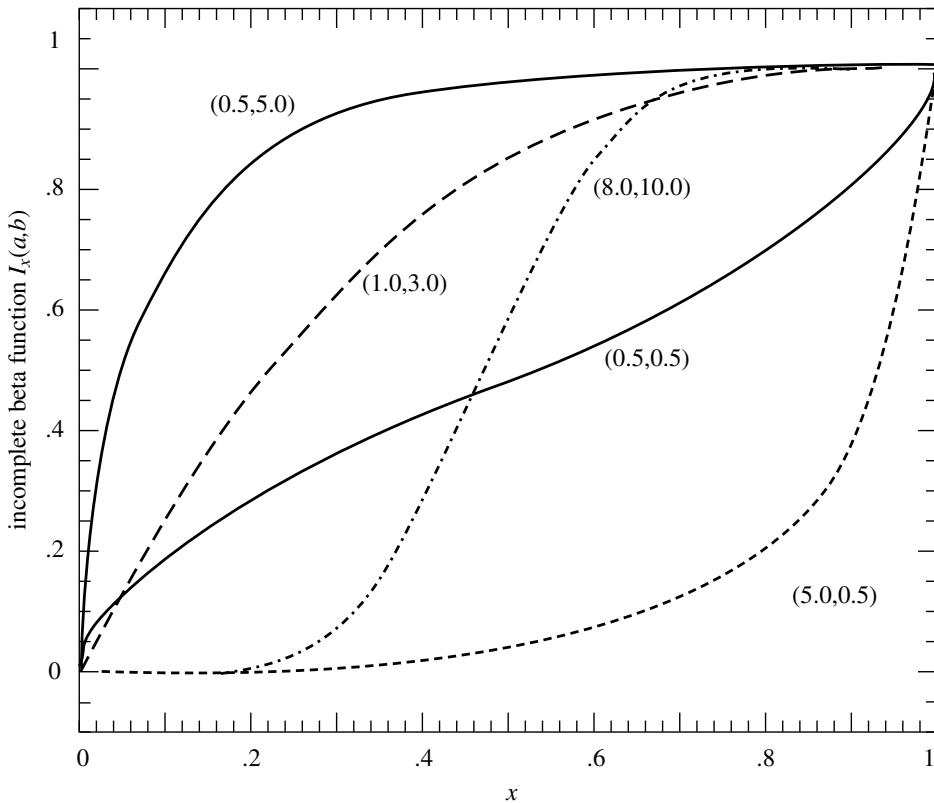
The continued fraction representation proves to be much more useful:

$$I_x(a, b) = \frac{x^a (1-x)^b}{a B(a, b)} \left[ \frac{1}{1} \frac{d_1}{1} \frac{d_2}{1} \dots \right] \quad (6.4.5)$$

where

$$\begin{aligned} d_{2m+1} &= -\frac{(a+m)(a+b+m)x}{(a+2m)(a+2m+1)} \\ d_{2m} &= \frac{m(b-m)x}{(a+2m-1)(a+2m)} \end{aligned} \quad (6.4.6)$$

This continued fraction converges rapidly for  $x < (a+1)/(a+b+2)$ , except when  $a$  and  $b$  are both large, when it can take  $O(\sqrt{\min(a, b)})$  iterations. For  $x >$



**Figure 6.4.1.** The incomplete beta function  $I_x(a,b)$  for five different pairs of  $(a,b)$ . Notice that the pairs  $(0.5, 5.0)$  and  $(5.0, 0.5)$  are symmetrically related as indicated in equation (6.4.3).

$(a+1)/(a+b+2)$  we can just use the symmetry relation (6.4.3) to obtain an equivalent computation in which the convergence is again rapid. Our computational strategy is thus very similar to that used in **Gamma**: We use the continued fraction except when  $a$  and  $b$  are both large, in which case we do a single step of high-order Gauss-Legendre quadrature.

Also as in **Gamma**, we code an inverse function using Halley's method. When  $a$  and  $b$  are both  $\geq 1$ , the initial guess comes from §26.5.22 in reference [1]. When either is less than 1, the guess comes from first crudely approximating

$$\int_0^1 t^{a-1} (1-t)^{b-1} dt \approx \frac{1}{a} \left( \frac{a}{a+b} \right)^a + \frac{1}{b} \left( \frac{b}{a+b} \right)^b \equiv S \quad (6.4.7)$$

which comes from breaking the integral at  $t = a/(a+b)$  and ignoring one factor in the integrand on each side of the break. We then write

$$I_x(a,b) \approx \begin{cases} x^a / (S a) & x \leq a/(a+b) \\ (1-x)^b / (S b) & x > a/(a+b) \end{cases} \quad (6.4.8)$$

and solve for  $x$  in the respective regimes. While crude, this is good enough to get well within the basin of convergence in all cases.

ncgammabeta.h

```
struct Beta : Gauleg18 {
```

Object for incomplete beta function. Gauleg18 provides coefficients for Gauss-Legendre quadrature.

```
    static const Int SWITCH=3000;           When to switch to quadrature method.
    static const Doub EPS, FPMIN;           See end of struct for initializations.
```

```
Doub betai(const Doub a, const Doub b, const Doub x) {
```

Returns incomplete beta function  $I_x(a, b)$  for positive  $a$  and  $b$ , and  $x$  between 0 and 1.

```
    Doub bt;
    if (a <= 0.0 || b <= 0.0) throw("Bad a or b in routine betai");
    if (x < 0.0 || x > 1.0) throw("Bad x in routine betai");
    if (x == 0.0 || x == 1.0) return x;
    if (a > SWITCH && b > SWITCH) return betaiapprox(a,b,x);
    bt=exp(gammln(a+b)-gammln(a)-gammln(b)+a*log(x)+b*log(1.0-x));
    if (x < (a+1.0)/(a+b+2.0)) return bt*betacf(a,b,x)/a;
    else return 1.0-bt*betacf(b,a,1.0-x)/b;
}
```

```
Doub betacf(const Doub a, const Doub b, const Doub x) {
```

Evaluates continued fraction for incomplete beta function by modified Lentz's method (§5.2). User should not call directly.

```
    Int m,m2;
    Doub aa,c,d,del,h,qab,qam,qap;
    qab=a+b;
    qap=a+1.0;
    qam=a-1.0;
    c=1.0;
    d=1.0-qab*x/qap;
    if (fabs(d) < FPMIN) d=FPMIN;
    d=1.0/d;
    h=d;
    for (m=1;m<10000;m++) {
        m2=2*m;
        aa=m*(b-m)*x/((qam+m2)*(a+m2));
        d=1.0+aa*d;
        if (fabs(d) < FPMIN) d=FPMIN;
        c=1.0+aa/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        h *= d*c;
        aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
        d=1.0+aa*d;
        if (fabs(d) < FPMIN) d=FPMIN;
        c=1.0+aa/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        del=d*c;
        h *= del;
        if (fabs(del-1.0) <= EPS) break;  Are we done?
    }
    return h;
}
```

These q's will be used in factors that occur in the coefficients (6.4.6).

First step of Lentz's method.

One step (the even one) of the recurrence.

Next step of the recurrence (the odd one).

```
Doub betaiapprox(Doub a, Doub b, Doub x) {
```

Incomplete beta by quadrature. Returns  $I_x(a, b)$ . User should not call directly.

```
    Int j;
    Doub xu,t,sum,ans;
    Doub a1 = a-1.0, b1 = b-1.0, mu = a/(a+b);
    Doub lnmu=log(mu),lnmuc=log(1.-mu);
    t = sqrt(a*b/(SQR(a+b)*(a+b+1.0)));
    if (x > a/(a+b)) {                                Set how far to integrate into the tail:
        if (x >= 1.0) return 1.0;
        xu = MIN(1.,MAX(mu + 10.*t, x + 5.0*t));
    } else {
```

```

        if (x <= 0.0) return 0.0;
        xu = MAX(0.,MIN(mu - 10.*t, x - 5.0*t));
    }
    sum = 0;
    for (j=0;j<18;j++)           Gauss-Legendre.
        t = x + (xu-x)*y[j];
        sum += w[j]*exp(a1*(log(t)-lnmu)+b1*(log(1-t)-lnmuc));
    }
    ans = sum*(xu-x)*exp(a1*lnmu-gammln(a)+b1*lnmuc-gammln(b)+gammln(a+b));
    return ans>0.0? 1.0-ans : -ans;
}

Doub invbetai(Doub p, Doub a, Doub b) {
Inverse of incomplete beta function. Returns  $x$  such that  $I_x(a,b) = p$  for argument  $p$  between 0 and 1.
    const Doub EPS = 1.e-8;
    Doub pp,t,u,err,x,al,h,w,afac,a1=a-1.,b1=b-1.;
    Int j;
    if (p <= 0.) return 0.;
    else if (p >= 1.) return 1.;
    else if (a >= 1. && b >= 1.) {           Set initial guess. See text.
        pp = (p < 0.5)? p : 1. - p;
        t = sqrt(-2.*log(pp));
        x = (2.30753+t*0.27061)/(1.+t*(0.99229+t*0.04481)) - t;
        if (p < 0.5) x = -x;
        al = (SQR(x)-3.)/6.;
        h = 2./(1./(2.*a-1.)+1./(2.*b-1.));
        w = (x*sqrt(al+h)/h)-(1./(2.*b-1)-1./(2.*a-1.))*(al+5./6.-2./(3.*h));
        x = a/(a+b*exp(2.*w));
    } else {
        Doub lna = log(a/(a+b)), lnb = log(b/(a+b));
        t = exp(alna)/a;
        u = exp(lnb)/b;
        w = t + u;
        if (p < t/w) x = pow(a*w*p,1./a);
        else x = 1. - pow(b*w*(1.-p),1./b);
    }
    afac = -gammln(a)-gammln(b)+gammln(a+b);
    for (j=0;j<10;j++) {
        if (x == 0. || x == 1.) return x;      a or b too small for accurate calcu-
        err = betai(a,b,x) - p;             lation.
        t = exp(a1*log(x)+b1*log(1.-x) + afac);       Halley:
        u = err/t;
        x -= (t = u/(1.-0.5*MIN(1.,u*(a1/x - b1/(1.-x)))));       Bisect if x tries to go neg or > 1.
        if (x <= 0.) x = 0.5*(x + t);
        if (x >= 1.) x = 0.5*(x + t + 1.);
        if (fabs(t) < EPS*x && j > 0) break;
    }
    return x;
}

};

const Doub Beta::EPS = numeric_limits<Doub>::epsilon();
const Doub Beta::FPMIN = numeric_limits<Doub>::min()/EPS;

```

**CITED REFERENCES AND FURTHER READING:**

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, Chapters 6 and 26.[1]
- Pearson, E., and Johnson, N. 1968, *Tables of the Incomplete Beta Function* (Cambridge, UK: Cambridge University Press).

## 6.5 Bessel Functions of Integer Order

This section presents practical algorithms for computing various kinds of Bessel functions of integer order. In §6.6 we deal with fractional order. Actually, the more complicated routines for fractional order work fine for integer order too. For integer order, however, the routines in this section are simpler and faster.

For any real  $\nu$ , the Bessel function  $J_\nu(x)$  can be defined by the series representation

$$J_\nu(x) = \left(\frac{1}{2}x\right)^\nu \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!\Gamma(\nu+k+1)} \quad (6.5.1)$$

The series converges for all  $x$ , but it is not computationally very useful for  $x \gg 1$ .

For  $\nu$  *not* an integer, the Bessel function  $Y_\nu(x)$  is given by

$$Y_\nu(x) = \frac{J_\nu(x) \cos(\nu\pi) - J_{-\nu}(x)}{\sin(\nu\pi)} \quad (6.5.2)$$

The right-hand side goes to the correct limiting value  $Y_n(x)$  as  $\nu$  goes to some integer  $n$ , but this is also not computationally useful.

For arguments  $x < \nu$ , both Bessel functions look qualitatively like simple power laws, with the asymptotic forms for  $0 < x \ll \nu$

$$\begin{aligned} J_\nu(x) &\sim \frac{1}{\Gamma(\nu+1)} \left(\frac{1}{2}x\right)^\nu & \nu \geq 0 \\ Y_0(x) &\sim \frac{2}{\pi} \ln(x) \\ Y_\nu(x) &\sim -\frac{\Gamma(\nu)}{\pi} \left(\frac{1}{2}x\right)^{-\nu} & \nu > 0 \end{aligned} \quad (6.5.3)$$

For  $x > \nu$ , both Bessel functions look qualitatively like sine or cosine waves whose amplitude decays as  $x^{-1/2}$ . The asymptotic forms for  $x \gg \nu$  are

$$\begin{aligned} J_\nu(x) &\sim \sqrt{\frac{2}{\pi x}} \cos\left(x - \frac{1}{2}\nu\pi - \frac{1}{4}\pi\right) \\ Y_\nu(x) &\sim \sqrt{\frac{2}{\pi x}} \sin\left(x - \frac{1}{2}\nu\pi - \frac{1}{4}\pi\right) \end{aligned} \quad (6.5.4)$$

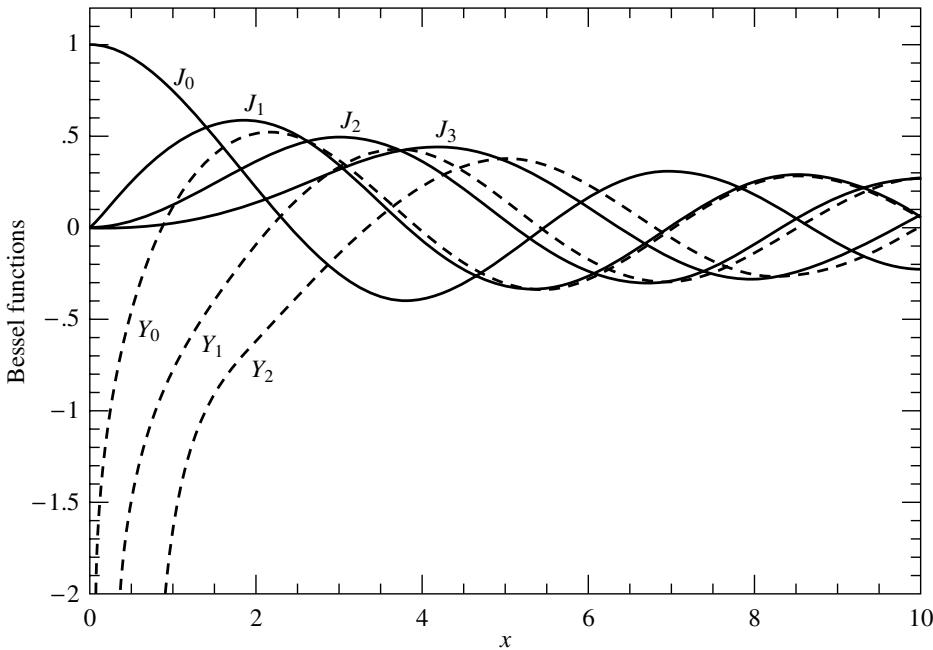
In the transition region where  $x \sim \nu$ , the typical amplitudes of the Bessel functions are on the order

$$\begin{aligned} J_\nu(\nu) &\sim \frac{2^{1/3}}{3^{2/3}\Gamma(\frac{2}{3})} \frac{1}{\nu^{1/3}} \sim \frac{0.4473}{\nu^{1/3}} \\ Y_\nu(\nu) &\sim -\frac{2^{1/3}}{3^{1/6}\Gamma(\frac{2}{3})} \frac{1}{\nu^{1/3}} \sim -\frac{0.7748}{\nu^{1/3}} \end{aligned} \quad (6.5.5)$$

which holds asymptotically for large  $\nu$ . Figure 6.5.1 plots the first few Bessel functions of each kind.

The Bessel functions satisfy the recurrence relations

$$J_{n+1}(x) = \frac{2n}{x} J_n(x) - J_{n-1}(x) \quad (6.5.6)$$



**Figure 6.5.1.** Bessel functions  $J_0(x)$  through  $J_3(x)$  and  $Y_0(x)$  through  $Y_2(x)$ .

and

$$Y_{n+1}(x) = \frac{2n}{x} Y_n(x) - Y_{n-1}(x) \quad (6.5.7)$$

As already mentioned in §5.4, only the second of these, (6.5.7), is stable in the direction of increasing  $n$  for  $x < n$ . The reason that (6.5.6) is unstable in the direction of increasing  $n$  is simply that it is *the same recurrence* as (6.5.7): A small amount of “polluting”  $Y_n$  introduced by roundoff error will quickly come to swamp the desired  $J_n$ , according to equation (6.5.3).

A practical strategy for computing the Bessel functions of integer order divides into two tasks: first, how to compute  $J_0$ ,  $J_1$ ,  $Y_0$ , and  $Y_1$ ; and second, how to use the recurrence relations stably to find other  $J$ ’s and  $Y$ ’s. We treat the first task first.

For  $x$  between zero and some arbitrary value (we will use the value 8), approximate  $J_0(x)$  and  $J_1(x)$  by rational functions in  $x$ . Likewise approximate by rational functions the “regular part” of  $Y_0(x)$  and  $Y_1(x)$ , defined as

$$Y_0(x) - \frac{2}{\pi} J_0(x) \ln(x) \quad \text{and} \quad Y_1(x) - \frac{2}{\pi} \left[ J_1(x) \ln(x) - \frac{1}{x} \right] \quad (6.5.8)$$

For  $8 < x < \infty$ , use the approximating forms ( $n = 0, 1$ )

$$J_n(x) = \sqrt{\frac{2}{\pi x}} \left[ P_n \left( \frac{8}{x} \right) \cos(X_n) - Q_n \left( \frac{8}{x} \right) \sin(X_n) \right] \quad (6.5.9)$$

$$Y_n(x) = \sqrt{\frac{2}{\pi x}} \left[ P_n \left( \frac{8}{x} \right) \sin(X_n) + Q_n \left( \frac{8}{x} \right) \cos(X_n) \right] \quad (6.5.10)$$

where

$$X_n \equiv x - \frac{2n+1}{4}\pi \quad (6.5.11)$$

and where  $P_0, P_1, Q_0$ , and  $Q_1$  are each polynomials in their arguments, for  $0 < 8/x < 1$ . The  $P$ 's are even polynomials, the  $Q$ 's odd.

In the routines below, the various coefficients were calculated in multiple precision so as to achieve full double precision in the relative error. (In the neighborhood of the zeros of the functions, it is the absolute error that is double precision.) However, because of roundoff, evaluating the approximations can lead to a loss of up to two significant digits.

One additional twist: The rational approximation for  $0 < x < 8$  is actually computed in the form [1]

$$J_0(x) = (x^2 - x_0^2)(x^2 - x_1^2) \frac{r(x^2)}{s(x^2)} \quad (6.5.12)$$

and similarly for  $J_1, Y_0$  and  $Y_1$ . Here  $x_0$  and  $x_1$  are the two zeros of  $J_0$  in the interval, and  $r$  and  $s$  are polynomials. The polynomial  $r(x^2)$  has alternating signs. Writing it in terms of  $64 - x^2$  makes all the signs the same and reduces roundoff error. For the approximations (6.5.9) and (6.5.10), our coefficients are similar but not identical to those given by Hart [2].

The functions  $J_0, J_1, Y_0$ , and  $Y_1$  share a lot of code, so we package them as a single object `Bessjy`. The routines for higher  $J_n$  and  $Y_n$  are also member functions, with implementations discussed below. All the numerical coefficients are declared in `Bessjy` but defined (as a long list of constants) separately; the listing is in a Webnote [3].

```
bessel.h struct Bessjy {
    static const Doub xj00,xj10,xj01,xj11,twoopi,pio4;
    static const Doub j0r[7],j0s[7],j0pn[5],j0pd[5],j0qn[5],j0qd[5];
    static const Doub j1r[7],j1s[7],j1pn[5],j1pd[5],j1qn[5],j1qd[5];
    static const Doub y0r[9],y0s[9],y0pn[5],y0pd[5],y0qn[5],y0qd[5];
    static const Doub y1r[8],y1s[8],y1pn[5],y1pd[5],y1qn[5],y1qd[5];
    Doub nump,denp,numq,denq,y,z,ax,xx;

    Doub j0(const Doub x) {
        Returns the Bessel function  $J_0(x)$  for any real  $x$ .
        if ((ax=abs(x)) < 8.0) {           Direct rational function fit.
            rat(x,j0r,j0s,6);
            return nump*(y-xj00)*(y-xj10)/denp;
        } else {                           Fitting function (6.5.9).
            asp(j0pn,j0pd,j0qn,j0qd,1.);
            return sqrt(twoopi/ax)*(cos(xx)*nump/denp-z*sin(xx)*numq/denq);
        }
    }

    Doub j1(const Doub x) {
        Returns the Bessel function  $J_1(x)$  for any real  $x$ .
        if ((ax=abs(x)) < 8.0) {           Direct rational approximation.
            rat(x,j1r,j1s,6);
            return x*nump*(y-xj01)*(y-xj11)/denp;
        } else {                           Fitting function (6.5.9).
            asp(j1pn,j1pd,j1qn,j1qd,3.);
            Doub ans=sqrt(twoopi/ax)*(cos(xx)*nump/denp-z*sin(xx)*numq/denq);
            return x > 0.0 ? ans : -ans;
        }
    }
}
```

```

Doub y0(const Doub x) {
    Returns the Bessel function  $Y_0(x)$  for positive x.
    if (x < 8.0) {                                     Rational function approximation of (6.5.8).
        Doub j0x = j0(x);
        rat(x,y0r,y0s,8);
        return nump/denp+twoopi*j0x*log(x);
    } else {                                         Fitting function (6.5.10).
        ax=x;
        asp(y0pn,y0pd,y0qn,y0qd,1.);
        return sqrt(twoopi/x)*(sin(xx)*nump/denp+z*cos(xx)*numq/denq);
    }
}

Doub y1(const Doub x) {
    Returns the Bessel function  $Y_1(x)$  for positive x.
    if (x < 8.0) {                                     Rational function approximation of (6.5.8).
        Doub j1x = j1(x);
        rat(x,y1r,y1s,7);
        return x*nump/denp+twoopi*(j1x*log(x)-1.0/x);
    } else {                                         Fitting function (6.5.10).
        ax=x;
        asp(y1pn,y1pd,y1qn,y1qd,3.);
        return sqrt(twoopi/x)*(sin(xx)*nump/denp+z*cos(xx)*numq/denq);
    }
}

Doub jn(const Int n, const Doub x);
Returns the Bessel function  $J_n(x)$  for any real x and integer  $n \geq 0$ .

Doub yn(const Int n, const Doub x);
Returns the Bessel function  $Y_n(x)$  for any positive x and integer  $n \geq 0$ .

void rat(const Doub x, const Doub *r, const Doub *s, const Int n) {
Common code: Evaluates rational approximation.
    y = **x;
    z=64.0-y;
    nump=r[n];
    denp=s[n];
    for (Int i=n-1;i>=0;i--) {
        nump=nump*z+r[i];
        denp=denp*y+s[i];
    }
}

void asp(const Doub *pn, const Doub *pd, const Doub *qn, const Doub *qd,
Common code: Evaluates asymptotic approximation.
    const Doub fac) {
    z=8.0/ax;
    y=z*z;
    xx=ax-fac*pi04;
    nump=pn[4];
    denp=pd[4];
    numq=qn[4];
    denq=qd[4];
    for (Int i=3;i>=0;i--) {
        nump=nump*y+pn[i];
        denp=denp*y+pd[i];
        numq=numq*y+qn[i];
        denq=denq*y+qd[i];
    }
};

};

```

We now turn to the second task, namely, how to use the recurrence formulas (6.5.6) and (6.5.7) to get the Bessel functions  $J_n(x)$  and  $Y_n(x)$  for  $n \geq 2$ . The latter of these is straightforward, since its upward recurrence is always stable:

```
bessel.h Doub Bessjy::yn(const Int n, const Doub x)
Returns the Bessel function  $Y_n(x)$  for any positive  $x$  and integer  $n \geq 0$ .
{
    Int j;
    Doub by,bym,byp,tox;
    if (n==0) return y0(x);
    if (n==1) return y1(x);
    tox=2.0/x;
    by=y1(x);                                Starting values for the recurrence.
    bym=y0(x);
    for (j=1;j<n;j++) {                      Recurrence (6.5.7).
        byp=j*tox*by-bym;
        bym=by;
        by=byp;
    }
    return by;
}
```

The cost of this algorithm is the calls to  $y1$  and  $y0$  (which generate a call to each of  $j1$  and  $j0$ ), plus  $O(n)$  operations in the recurrence.

For  $J_n(x)$ , things are a bit more complicated. We can start the recurrence upward on  $n$  from  $J_0$  and  $J_1$ , but it will remain stable only while  $n$  does not exceed  $x$ . This is, however, just fine for calls with large  $x$  and small  $n$ , a case that occurs frequently in practice.

The harder case to provide for is that with  $x < n$ . The best thing to do here is to use Miller's algorithm (see discussion preceding equation 5.4.16), applying the recurrence *downward* from some arbitrary starting value and making use of the upward-unstable nature of the recurrence to put us *onto* the correct solution. When we finally arrive at  $J_0$  or  $J_1$  we are able to normalize the solution with the sum (5.4.16) accumulated along the way.

The only subtlety is in deciding at how large an  $n$  we need start the downward recurrence so as to obtain a desired accuracy by the time we reach the  $n$  that we really want. If you play with the asymptotic forms (6.5.3) and (6.5.5), you should be able to convince yourself that the answer is to start larger than the desired  $n$  by an additive amount of order  $[\text{constant} \times n]^{1/2}$ , where the square root of the constant is, very roughly, the number of significant figures of accuracy.

The above considerations lead to the following function.

```
bessel.h Doub Bessjy::jn(const Int n, const Doub x)
Returns the Bessel function  $J_n(x)$  for any real  $x$  and integer  $n \geq 0$ .
{
    const Doub ACC=160.0;                      ACC determines accuracy.
    const Int IEXP=numeric_limits<Doub>::max_exponent/2;
    Bool jsum;
    Int j,k,m;
    Doub ax,bj,bjm,bjp,dum,sum,tox,ans;
    if (n==0) return j0(x);
    if (n==1) return j1(x);
    ax=abs(x);
    if (ax*ax <= 8.0*numeric_limits<Doub>::min()) return 0.0;
    else if (ax > Doub(n)) {                  Upwards recurrence from  $J_0$  and  $J_1$ .
        tox=2.0/ax;
```

```

bjm=j0(ax);
bj=j1(ax);
for (j=1;j<n;j++) {
    bjp=j*t0x*bj-bjm;
    bjm=bj;
    bj=bjp;
}
ans=bj;
} else {
    tox=2.0/ax;
    m=2*((n+Int(sqrt(ACC*n)))/2);
    jsum=false;
    bjp=ans=sum=0.0;
    bj=1.0;
    for (j=m;j>0;j--) {
        bjm=j*t0x*bj-bjp;
        bjp=bj;
        bj=bjm;
        dum=frexp(bj,&k);
        if (k > IEXP) {
            bj=ldexp(bj,-IEXP);
            bjp=ldexp(bjp,-IEXP);
            ans=ldexp(ans,-IEXP);
            sum=ldexp(sum,-IEXP);
        }
        if (jsum) sum += bj;
        jsum=!jsum;
        if (j == n) ans=bjp;
    }
    sum=2.0*sum-bj;
    ans /= sum;
}
return x < 0.0 && (n & 1) ? -ans : ans;
}

```

Downward recurrence from an even  $m$  here computed.

$jsum$  will alternate between `false` and `true`; when it is `true`, we accumulate in `sum` the even terms in (5.4.16).

The downward recurrence.

Renormalize to prevent overflows.

Accumulate the sum.

Change `false` to `true` or vice versa.

Save the unnormalized answer.

Compute (5.4.16)  
and use it to normalize the answer.

The function `ldexp`, used above, is a standard C and C++ library function for scaling the binary exponent of a number.

### 6.5.1 Modified Bessel Functions of Integer Order

The modified Bessel functions  $I_n(x)$  and  $K_n(x)$  are equivalent to the usual Bessel functions  $J_n$  and  $Y_n$  evaluated for purely imaginary arguments. In detail, the relationship is

$$\begin{aligned} I_n(x) &= (-i)^n J_n(ix) \\ K_n(x) &= \frac{\pi}{2} i^{n+1} [J_n(ix) + i Y_n(ix)] \end{aligned} \quad (6.5.13)$$

The particular choice of prefactor and of the linear combination of  $J_n$  and  $Y_n$  to form  $K_n$  are simply choices that make the functions real-valued for real arguments  $x$ .

For small arguments  $x \ll n$ , both  $I_n(x)$  and  $K_n(x)$  become, asymptotically, simple powers of their arguments

$$\begin{aligned} I_n(x) &\approx \frac{1}{n!} \left(\frac{x}{2}\right)^n \quad n \geq 0 \\ K_0(x) &\approx -\ln(x) \\ K_n(x) &\approx \frac{(n-1)!}{2} \left(\frac{x}{2}\right)^{-n} \quad n > 0 \end{aligned} \quad (6.5.14)$$

These expressions are virtually identical to those for  $J_n(x)$  and  $Y_n(x)$  in this region, except for the factor of  $-2/\pi$  difference between  $Y_n(x)$  and  $K_n(x)$ . In the region  $x \gg n$ , however, the modified functions have quite different behavior than the Bessel functions,

$$\begin{aligned} I_n(x) &\approx \frac{1}{\sqrt{2\pi x}} \exp(x) \\ K_n(x) &\approx \frac{\pi}{\sqrt{2\pi x}} \exp(-x) \end{aligned} \quad (6.5.15)$$

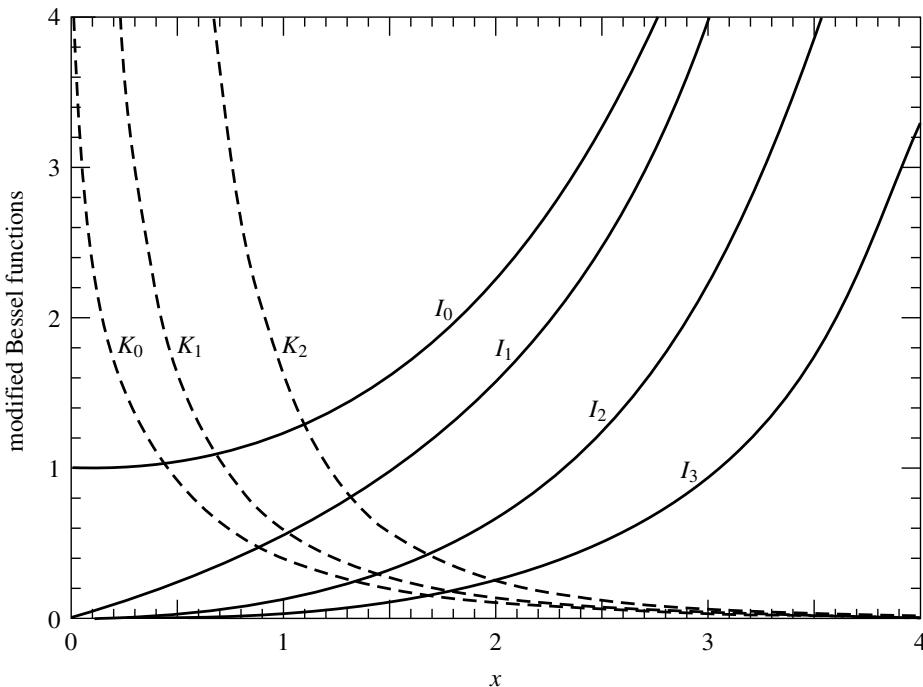
The modified functions evidently have exponential rather than sinusoidal behavior for large arguments (see Figure 6.5.2). Rational approximations analogous to those for the  $J$  and  $Y$  Bessel functions are efficient for computing  $I_0$ ,  $I_1$ ,  $K_0$ , and  $K_1$ . The corresponding routines are packaged as an object `Bessik`. The routines are similar to those in [1], although different in detail. (All the constants are again listed in a Webnote [3].)

```
bessel.h struct Bessik {
    static const Doub i0p[14],i0q[5],i0pp[5],i0qq[6];
    static const Doub i1p[14],i1q[5],i1pp[5],i1qq[6];
    static const Doub k0pi[5],k0qi[3],k0p[5],k0q[3],k0pp[8],k0qq[8];
    static const Doub k1pi[5],k1qi[3],k1p[5],k1q[3],k1pp[8],k1qq[8];
    Doub y,z,ax,term;

    Doub i0(const Doub x) {
        Returns the modified Bessel function  $I_0(x)$  for any real  $x$ .
        if ((ax=abs(x)) < 15.0) {           Rational approximation.
            y = x*x;
            return poly(i0p,13,y)/poly(i0q,4,225.-y);
        } else {                           Rational approximation with  $e^x/\sqrt{x}$  factored out.
            z=1.0-15.0/ax;
            return exp(ax)*poly(i0pp,4,z)/(poly(i0qq,5,z)*sqrt(ax));
        }
    }

    Doub i1(const Doub x) {
        Returns the modified Bessel function  $I_1(x)$  for any real  $x$ .
        if ((ax=abs(x)) < 15.0) {           Rational approximation.
            y=x*x;
            return x*poly(i1p,13,y)/poly(i1q,4,225.-y);
        } else {                           Rational approximation with  $e^x/\sqrt{x}$  factored out.
            z=1.0-15.0/ax;
            Doub ans=exp(ax)*poly(i1pp,4,z)/(poly(i1qq,5,z)*sqrt(ax));
            return x > 0.0 ? ans : -ans;
        }
    }

    Doub k0(const Doub x) {
        Returns the modified Bessel function  $K_0(x)$  for positive real  $x$ .
        if (x <= 1.0) {                   Use two rational approximations.
            z=x*x;
            term = poly(k0pi,4,z)*log(x)/poly(k0qi,2,1.-z);
            return poly(k0p,4,z)/poly(k0q,2,1.-z)-term;
        } else {                           Rational approximation with  $e^{-x}/\sqrt{x}$  factored
            z=1.0/x;                      out.
            return exp(-x)*poly(k0pp,7,z)/(poly(k0qq,7,z)*sqrt(x));
        }
    }
}
```



**Figure 6.5.2.** Modified Bessel functions  $I_0(x)$  through  $I_3(x)$ , and  $K_0(x)$  through  $K_2(x)$ .

```

Doub k1(const Doub x) {
    Returns the modified Bessel function  $K_1(x)$  for positive real  $x$ .
    if ( $x \leq 1.0$ ) {                                     Use two rational approximations.
        z=x*x;
        term = poly(k1pi,4,z)*log(x)/poly(k1qi,2,1.-z);
        return x*(poly(k1p,4,z)/poly(k1q,2,1.-z)+term)+1./x;
    } else {                                              Rational approximation with  $e^{-x}/\sqrt{x}$  factored
        z=1.0/x;                                         out.
        return exp(-x)*poly(k1pp,7,z)/(poly(k1qq,7,z)*sqrt(x));
    }
}

Doub in(const Int n, const Doub x);
    Returns the modified Bessel function  $I_n(x)$  for any real  $x$  and  $n \geq 0$ .

Doub kn(const Int n, const Doub x);
    Returns the modified Bessel function  $K_n(x)$  for positive  $x$  and  $n \geq 0$ .

inline Doub poly(const Doub *cof, const Int n, const Doub x) {
    Common code: Evaluate a polynomial.
    Doub ans = cof[1];
    for (Int i=n-1;i>=0;i--) ans = ans*x+cof[i];
    return ans;
}
;
```

The recurrence relation for  $I_n(x)$  and  $K_n(x)$  is the same as that for  $J_n(x)$  and  $Y_n(x)$  provided that  $ix$  is substituted for  $x$ . This has the effect of changing a sign in the relation,

$$\begin{aligned} I_{n+1}(x) &= -\left(\frac{2n}{x}\right) I_n(x) + I_{n-1}(x) \\ K_{n+1}(x) &= +\left(\frac{2n}{x}\right) K_n(x) + K_{n-1}(x) \end{aligned} \quad (6.5.16)$$

These relations are always *unstable* for upward recurrence. For  $K_n$ , itself growing, this presents no problem. The implementation is

```
bessel.h Doub Bessik::kn(const Int n, const Doub x)
Returns the modified Bessel function  $K_n(x)$  for positive  $x$  and  $n \geq 0$ .
{
    Int j;
    Doub bkm,bkp,tox;
    if (n==0) return k0(x);
    if (n==1) return k1(x);
    tox=2.0/x;
    bkm=k0(x);                                Upward recurrence for all x...
    bk=k1(x);
    for (j=1;j<n;j++) {                      ...and here it is.
        bkp=bkm+j*tox*bk;
        bkm=bk;
        bk=bkp;
    }
    return bk;
}
```

For  $I_n$ , the strategy of downward recursion is required once again, and the starting point for the recursion may be chosen in the same manner as for the routine `Bessjy::jn`. The only fundamental difference is that the normalization formula for  $I_n(x)$  has an alternating minus sign in successive terms, which again arises from the substitution of  $ix$  for  $x$  in the formula used previously for  $J_n$ :

$$1 = I_0(x) - 2I_2(x) + 2I_4(x) - 2I_6(x) + \dots \quad (6.5.17)$$

In fact, we prefer simply to normalize with a call to `i0`.

```
bessel.h Doub Bessik::in(const Int n, const Doub x)
Returns the modified Bessel function  $I_n(x)$  for any real  $x$  and  $n \geq 0$ .
{
    const Doub ACC=200.0;                      ACC determines accuracy.
    const Int IEXP=numeric_limits<Doub>::max_exponent/2;
    Int j,k;
    Doub bi,bim,bip,dum,tox,ans;
    if (n==0) return i0(x);
    if (n==1) return i1(x);
    if (x*x <= 8.0*numeric_limits<Doub>::min()) return 0.0;
    else {
        tox=2.0/abs(x);
        bip=ans=0.0;
        bi=1.0;
        for (j=2*(n+Int(sqrt(ACC*n)));j>0;j--) {      Downward recurrence.
            bim=bip+j*tox*bi;
            bip=bi;
            bi=bim;
            dum=frexp(bi,&k);
            if (k > IEXP) {                            Renormalize to prevent overflows.
                ans=ldexp(ans,-IEXP);
                bi=ldexp(bi,-IEXP);
                bip=ldexp(bip,-IEXP);
            }
        }
    }
}
```

```

        }
        if (j == n) ans=bip;
    }
    ans *= i0(x)/bi;                                Normalize with bessio.
    return x < 0.0 && (n & 1) ? -ans : ans;
}

```

The function `1dexp`, used above, is a standard C and C++ library function for scaling the binary exponent of a number.

#### CITED REFERENCES AND FURTHER READING:

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, Chapter 9.
- Carrier, G.F., Krook, M. and Pearson, C.E. 1966, *Functions of a Complex Variable* (New York: McGraw-Hill), pp. 220ff.
- SPECFUN*, 2007+, at <http://www.netlib.org/specfun>.[1]
- Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley), §6.8, p. 141.[2]
- Numerical Recipes Software 2007, “Coefficients Used in the Bessjy and Bessik Objects,” *Numerical Recipes Webnote No. 7*, at <http://www.nr.com/webnotes?7> [3]

## 6.6 Bessel Functions of Fractional Order, Airy Functions, Spherical Bessel Functions

Many algorithms have been proposed for computing Bessel functions of fractional order numerically. Most of them are, in fact, not very good in practice. The routines given here are rather complicated, but they can be recommended wholeheartedly.

### 6.6.1 Ordinary Bessel Functions

The basic idea is *Steed's method*, which was originally developed [1] for Coulomb wave functions. The method calculates  $J_\nu$ ,  $J'_\nu$ ,  $Y_\nu$ , and  $Y'_\nu$  simultaneously, and so involves four relations among these functions. Three of the relations come from two continued fractions, one of which is complex. The fourth is provided by the Wronskian relation

$$W \equiv J_\nu Y'_\nu - Y_\nu J'_\nu = \frac{2}{\pi x} \quad (6.6.1)$$

The first continued fraction, CF1, is defined by

$$\begin{aligned} f_\nu \equiv \frac{J'_\nu}{J_\nu} &= \frac{\nu}{x} - \frac{J_{\nu+1}}{J_\nu} \\ &= \frac{\nu}{x} - \frac{1}{2(\nu+1)/x - \frac{1}{2(\nu+2)/x - \dots}} \end{aligned} \quad (6.6.2)$$

You can easily derive it from the three-term recurrence relation for Bessel functions: Start with equation (6.5.6) and use equation (5.4.18). Forward evaluation of the continued fraction by one of the methods of §5.2 is essentially equivalent to backward recurrence of the recurrence relation. The rate of convergence of CF1 is determined by the position of the *turning point*  $x_{tp} = \sqrt{\nu(\nu+1)} \approx \nu$ , beyond which the Bessel functions become oscillatory. If  $x \lesssim x_{tp}$ ,

convergence is very rapid. If  $x \gtrsim x_{\text{tp}}$ , then each iteration of the continued fraction effectively increases  $v$  by one until  $x \lesssim x_{\text{tp}}$ ; thereafter rapid convergence sets in. Thus the number of iterations of CF1 is of order  $x$  for large  $x$ . In the routine `besseljy` we set the maximum allowed number of iterations to 10,000. For larger  $x$ , you can use the usual asymptotic expressions for Bessel functions.

One can show that the sign of  $J_v$  is the same as the sign of the denominator of CF1 once it has converged.

The complex continued fraction CF2 is defined by

$$p + iq \equiv \frac{J'_v + iY'_v}{J_v + iY_v} = -\frac{1}{2x} + i + \frac{i}{x} \frac{(1/2)^2 - v^2}{2(x+i) +} \frac{(3/2)^2 - v^2}{2(x+2i) +} \dots \quad (6.6.3)$$

(We sketch the derivation of CF2 in the analogous case of modified Bessel functions in the next subsection.) This continued fraction converges rapidly for  $x \gtrsim x_{\text{tp}}$ , while convergence fails as  $x \rightarrow 0$ . We have to adopt a special method for small  $x$ , which we describe below. For  $x$  not too small, we can ensure that  $x \gtrsim x_{\text{tp}}$  by a stable recurrence of  $J_v$  and  $J'_v$  downward to a value  $v = \mu \lesssim x$ , thus yielding the ratio  $f_\mu$  at this lower value of  $v$ . This is the stable direction for the recurrence relation. The initial values for the recurrence are

$$J_v = \text{arbitrary}, \quad J'_v = f_v J_v, \quad (6.6.4)$$

with the sign of the arbitrary initial value of  $J_v$  chosen to be the sign of the denominator of CF1. Choosing the initial value of  $J_v$  very small minimizes the possibility of overflow during the recurrence. The recurrence relations are

$$\begin{aligned} J_{v-1} &= \frac{v}{x} J_v + J'_v \\ J'_{v-1} &= \frac{v-1}{x} J_{v-1} - J_v \end{aligned} \quad (6.6.5)$$

Once CF2 has been evaluated at  $v = \mu$ , then with the Wronskian (6.6.1) we have enough relations to solve for all four quantities. The formulas are simplified by introducing the quantity

$$\gamma \equiv \frac{p - f_\mu}{q} \quad (6.6.6)$$

Then

$$J_\mu = \pm \left( \frac{W}{q + \gamma(p - f_\mu)} \right)^{1/2} \quad (6.6.7)$$

$$J'_\mu = f_\mu J_\mu \quad (6.6.8)$$

$$Y_\mu = \gamma J_\mu \quad (6.6.9)$$

$$Y'_\mu = Y_\mu \left( p + \frac{q}{\gamma} \right) \quad (6.6.10)$$

The sign of  $J_\mu$  in (6.6.7) is chosen to be the same as the sign of the initial  $J_v$  in (6.6.4).

Once all four functions have been determined at the value  $v = \mu$ , we can find them at the original value of  $v$ . For  $J_v$  and  $J'_v$ , simply scale the values in (6.6.4) by the ratio of (6.6.7) to the value found after applying the recurrence (6.6.5). The quantities  $Y_v$  and  $Y'_v$  can be found by starting with the values in (6.6.9) and (6.6.10) and using the stable upward recurrence

$$Y_{v+1} = \frac{2v}{x} Y_v - Y_{v-1} \quad (6.6.11)$$

together with the relation

$$Y'_v = \frac{v}{x} Y_v - Y_{v+1} \quad (6.6.12)$$

Now turn to the case of small  $x$ , when CF2 is not suitable. Temme [2] has given a good method of evaluating  $Y_v$  and  $Y_{v+1}$ , and hence  $Y'_v$  from (6.6.12), by series expansions that

accurately handle the singularity as  $x \rightarrow 0$ . The expansions work only for  $|\nu| \leq 1/2$ , and so now the recurrence (6.6.5) is used to evaluate  $f_\nu$  at a value  $\nu = \mu$  in this interval. Then one calculates  $J_\mu$  from

$$J_\mu = \frac{W}{Y'_\mu - Y_\mu f_\mu} \quad (6.6.13)$$

and  $J'_\mu$  from (6.6.8). The values at the original value of  $\nu$  are determined by scaling as before, and the  $Y$ 's are recurred up as before.

Temme's series are

$$Y_\nu = - \sum_{k=0}^{\infty} c_k g_k \quad Y_{\nu+1} = - \frac{2}{x} \sum_{k=0}^{\infty} c_k h_k \quad (6.6.14)$$

Here

$$c_k = \frac{(-x^2/4)^k}{k!} \quad (6.6.15)$$

while the coefficients  $g_k$  and  $h_k$  are defined in terms of quantities  $p_k$ ,  $q_k$ , and  $f_k$  that can be found by recursion:

$$\begin{aligned} g_k &= f_k + \frac{2}{\nu} \sin^2 \left( \frac{\nu\pi}{2} \right) q_k \\ h_k &= -k g_k + p_k \\ p_k &= \frac{p_{k-1}}{k - \nu} \\ q_k &= \frac{q_{k-1}}{k + \nu} \\ f_k &= \frac{k f_{k-1} + p_{k-1} + q_{k-1}}{k^2 - \nu^2} \end{aligned} \quad (6.6.16)$$

The initial values for the recurrences are

$$\begin{aligned} p_0 &= \frac{1}{\pi} \left( \frac{x}{2} \right)^{-\nu} \Gamma(1 + \nu) \\ q_0 &= \frac{1}{\pi} \left( \frac{x}{2} \right)^\nu \Gamma(1 - \nu) \\ f_0 &= \frac{2}{\pi} \frac{\nu\pi}{\sin \nu\pi} \left[ \cosh \sigma \Gamma_1(\nu) + \frac{\sinh \sigma}{\sigma} \ln \left( \frac{2}{x} \right) \Gamma_2(\nu) \right] \end{aligned} \quad (6.6.17)$$

with

$$\begin{aligned} \sigma &= \nu \ln \left( \frac{2}{x} \right) \\ \Gamma_1(\nu) &= \frac{1}{2\nu} \left[ \frac{1}{\Gamma(1 - \nu)} - \frac{1}{\Gamma(1 + \nu)} \right] \\ \Gamma_2(\nu) &= \frac{1}{2} \left[ \frac{1}{\Gamma(1 - \nu)} + \frac{1}{\Gamma(1 + \nu)} \right] \end{aligned} \quad (6.6.18)$$

The whole point of writing the formulas in this way is that the potential problems as  $\nu \rightarrow 0$  can be controlled by evaluating  $\nu\pi/\sin \nu\pi$ ,  $\sinh \sigma/\sigma$ , and  $\Gamma_1$  carefully. In particular, Temme gives Chebyshev expansions for  $\Gamma_1(\nu)$  and  $\Gamma_2(\nu)$ . We have rearranged his expansion for  $\Gamma_1$  to be explicitly an even series in  $\nu$  for more efficient evaluation, as explained in §5.8.

Because  $J_\nu$ ,  $Y_\nu$ ,  $J'_\nu$ , and  $Y'_\nu$  are all calculated simultaneously, a single void function sets them all. You then grab those that you need directly from the object. Alternatively, the functions `jnu` and `ynu` can be used. (We've omitted similar helper functions for the derivatives, but you can easily add them.) The object `Bessel` contains various other methods that will be discussed below.

The routines assume  $\nu \geq 0$ . For negative  $\nu$  you can use the reflection formulas

$$\begin{aligned} J_{-\nu} &= \cos \nu\pi J_\nu - \sin \nu\pi Y_\nu \\ Y_{-\nu} &= \sin \nu\pi J_\nu + \cos \nu\pi Y_\nu \end{aligned} \quad (6.6.19)$$

The routine also assumes  $x > 0$ . For  $x < 0$ , the functions are in general complex but expressible in terms of functions with  $x > 0$ . For  $x = 0$ ,  $Y_\nu$  is singular. The complex arithmetic is carried out explicitly with real variables.

besselfrac.h

```
struct Bessel {
    Object for Bessel functions of arbitrary order  $\nu$ , and related functions.
    static const Int NUSE1=7, NUSE2=8;
    static const Doub c1[NUSE1],c2[NUSE2];
    Doub xo,nuo;                                Saved  $x$  and  $\nu$  from last call.
    Doub jo,yo,jpo,ypo;                          Set by besseljy.
    Doub io,ko,ipo,kpo;                          Set by besselik.
    Doub aio,bio,aipo,bipo;                      Set by airy.
    Doub sphjo,sphyo,sphjpo,sphypo;            Set by sphbes.
    Int sphno;

    Bessel() : xo(9.99e99), nuo(9.99e99), sphno(-9999) {}
    Default constructor. No arguments.

    void besseljy(const Doub nu, const Doub x);
    Calculate Bessel functions  $J_\nu(x)$  and  $Y_\nu(x)$  and their derivatives.
    void besselik(const Doub nu, const Doub x);
    Calculate Bessel functions  $I_\nu(x)$  and  $K_\nu(x)$  and their derivatives.

    Doub jnu(const Doub nu, const Doub x) {
        Simple interface returning  $J_\nu(x)$ .
        if (nu != nuo || x != xo) besseljy(nu,x);
        return jo;
    }
    Doub ynu(const Doub nu, const Doub x) {
        Simple interface returning  $Y_\nu(x)$ .
        if (nu != nuo || x != xo) besseljy(nu,x);
        return yo;
    }
    Doub inu(const Doub nu, const Doub x) {
        Simple interface returning  $I_\nu(x)$ .
        if (nu != nuo || x != xo) besselik(nu,x);
        return io;
    }
    Doub knu(const Doub nu, const Doub x) {
        Simple interface returning  $K_\nu(x)$ .
        if (nu != nuo || x != xo) besselik(nu,x);
        return ko;
    }

    void airy(const Doub x);
    Calculate Airy functions  $Ai(x)$  and  $Bi(x)$  and their derivatives.
    Doub airy_ai(const Doub x);
    Simple interface returning  $Ai(x)$ .
    Doub airy_bi(const Doub x);
    Simple interface returning  $Bi(x)$ .

    void sphbes(const Int n, const Doub x);
    Calculate spherical Bessel functions  $j_n(x)$  and  $y_n(x)$  and their derivatives.
    Doub sphbesj(const Int n, const Doub x);
    Simple interface returning  $j_n(x)$ .
    Doub sphbesy(const Int n, const Doub x);
    Simple interface returning  $y_n(x)$ .
```

```

inline Doub chebev(const Doub *c, const Int m, const Doub x) {
    Utility used by besseljy and besselik, evaluates Chebyshev series.
    Doub d=0.0,dd=0.0,sv;
    Int j;
    for (j=m-1;j>0;j--) {
        sv=d;
        d=2.*x*d-dd+c[j];
        dd=sv;
    }
    return x*d-dd+0.5*c[0];
};

const Doub Bessel::c1[7] = {-1.142022680371168e0,6.5165112670737e-3,
    3.087090173086e-4,-3.4706269649e-6,6.9437664e-9,3.67795e-11,
    -1.356e-13};
const Doub Bessel::c2[8] = {1.843740587300905e0,-7.68528408447867e-2,
    1.2719271366546e-3,-4.9717367042e-6,-3.31261198e-8,2.423096e-10,
    -1.702e-13,-1.49e-15};

```

The code listing for `Bessel::besseljy` is in a Webnote [4].

## 6.6.2 Modified Bessel Functions

Steed's method does not work for modified Bessel functions because in this case CF2 is purely imaginary and we have only three relations among the four functions. Temme [3] has given a normalization condition that provides the fourth relation.

The Wronskian relation is

$$W \equiv I_\nu K'_\nu - K_\nu I'_\nu = -\frac{1}{x} \quad (6.6.20)$$

The continued fraction CF1 becomes

$$f_\nu \equiv \frac{I'_\nu}{I_\nu} = \frac{\nu}{x} + \frac{1}{2(\nu+1)/x +} \frac{1}{2(\nu+2)/x +} \dots \quad (6.6.21)$$

To get CF2 and the normalization condition in a convenient form, consider the sequence of confluent hypergeometric functions

$$z_n(x) = U(\nu + 1/2 + n, 2\nu + 1, 2x) \quad (6.6.22)$$

for fixed  $\nu$ . Then

$$K_\nu(x) = \pi^{1/2} (2x)^\nu e^{-x} z_0(x) \quad (6.6.23)$$

$$\frac{K_{\nu+1}(x)}{K_\nu(x)} = \frac{1}{x} \left[ \nu + \frac{1}{2} + x + \left( \nu^2 - \frac{1}{4} \right) \frac{z_1}{z_0} \right] \quad (6.6.24)$$

Equation (6.6.23) is the standard expression for  $K_\nu$  in terms of a confluent hypergeometric function, while equation (6.6.24) follows from relations between contiguous confluent hypergeometric functions (equations 13.4.16 and 13.4.18 in Ref. [5]). Now the functions  $z_n$  satisfy the three-term recurrence relation (equation 13.4.15 in Ref. [5])

$$z_{n-1}(x) = b_n z_n(x) + a_{n+1} z_{n+1} \quad (6.6.25)$$

with

$$\begin{aligned} b_n &= 2(n+x) \\ a_{n+1} &= -[(n+1/2)^2 - \nu^2] \end{aligned} \quad (6.6.26)$$

Following the steps leading to equation (5.4.18), we get the continued fraction CF2

$$\frac{z_1}{z_0} = \frac{1}{b_1 +} \frac{a_2}{b_2 +} \dots \quad (6.6.27)$$

from which (6.6.24) gives  $K_{\nu+1}/K_\nu$  and thus  $K'_\nu/K_\nu$ .

Temme's normalization condition is that

$$\sum_{n=0}^{\infty} C_n z_n = \left(\frac{1}{2x}\right)^{\nu+1/2} \quad (6.6.28)$$

where

$$C_n = \frac{(-1)^n}{n!} \frac{\Gamma(\nu + 1/2 + n)}{\Gamma(\nu + 1/2 - n)} \quad (6.6.29)$$

Note that the  $C_n$ 's can be determined by recursion:

$$C_0 = 1, \quad C_{n+1} = -\frac{a_{n+1}}{n+1} C_n \quad (6.6.30)$$

We use the condition (6.6.28) by finding

$$S = \sum_{n=1}^{\infty} C_n \frac{z_n}{z_0} \quad (6.6.31)$$

Then

$$z_0 = \left(\frac{1}{2x}\right)^{\nu+1/2} \frac{1}{1+S} \quad (6.6.32)$$

and (6.6.23) gives  $K_\nu$ .

Thompson and Barnett [6] have given a clever method of doing the sum (6.6.31) simultaneously with the forward evaluation of the continued fraction CF2. Suppose the continued fraction is being evaluated as

$$\frac{z_1}{z_0} = \sum_{n=0}^{\infty} \Delta h_n \quad (6.6.33)$$

where the increments  $\Delta h_n$  are being found by, e.g., Steed's algorithm or the modified Lentz's algorithm of §5.2. Then the approximation to  $S$  keeping the first  $N$  terms can be found as

$$S_N = \sum_{n=1}^N Q_n \Delta h_n \quad (6.6.34)$$

Here

$$Q_n = \sum_{k=1}^n C_k q_k \quad (6.6.35)$$

and  $q_k$  is found by recursion from

$$q_{k+1} = (q_{k-1} - b_k q_k)/a_{k+1} \quad (6.6.36)$$

starting with  $q_0 = 0$ ,  $q_1 = 1$ . For the case at hand, approximately three times as many terms are needed to get  $S$  to converge as are needed simply for CF2 to converge.

To find  $K_\nu$  and  $K_{\nu+1}$  for small  $x$  we use series analogous to (6.6.14):

$$K_\nu = \sum_{k=0}^{\infty} c_k f_k \quad K_{\nu+1} = \frac{2}{x} \sum_{k=0}^{\infty} c_k h_k \quad (6.6.37)$$

Here

$$\begin{aligned} c_k &= \frac{(x^2/4)^k}{k!} \\ h_k &= -kf_k + p_k \\ p_k &= \frac{pk-1}{k-\nu} \\ q_k &= \frac{qk-1}{k+\nu} \\ f_k &= \frac{kf_{k-1} + pk-1 + qk-1}{k^2 - \nu^2} \end{aligned} \quad (6.6.38)$$

The initial values for the recurrences are

$$\begin{aligned} p_0 &= \frac{1}{2} \left(\frac{x}{2}\right)^{-\nu} \Gamma(1+\nu) \\ q_0 &= \frac{1}{2} \left(\frac{x}{2}\right)^\nu \Gamma(1-\nu) \\ f_0 &= \frac{\nu\pi}{\sin\nu\pi} \left[ \cosh\sigma\Gamma_1(\nu) + \frac{\sinh\sigma}{\sigma} \ln\left(\frac{2}{x}\right) \Gamma_2(\nu) \right] \end{aligned} \quad (6.6.39)$$

Both the series for small  $x$ , and CF2 and the normalization relation (6.6.28) require  $|\nu| \leq 1/2$ . In both cases, therefore, we recurse  $I_\nu$  down to a value  $\nu = \mu$  in this interval, find  $K_\mu$  there, and recurse  $K_\nu$  back up to the original value of  $\nu$ .

The routine assumes  $\nu \geq 0$ . For negative  $\nu$  use the reflection formulas

$$\begin{aligned} I_{-\nu} &= I_\nu + \frac{2}{\pi} \sin(\nu\pi) K_\nu \\ K_{-\nu} &= K_\nu \end{aligned} \quad (6.6.40)$$

Note that for large  $x$ ,  $I_\nu \sim e^x$  and  $K_\nu \sim e^{-x}$ , and so these functions will overflow or underflow. It is often desirable to be able to compute the scaled quantities  $e^{-x}I_\nu$  and  $e^xK_\nu$ . Simply omitting the factor  $e^{-x}$  in equation (6.6.23) will ensure that all four quantities will have the appropriate scaling. If you also want to scale the four quantities for small  $x$  when the series in equation (6.6.37) are used, you must multiply each series by  $e^x$ .

As with `besseljy`, you can either call the `void` function `besselik`, and then retrieve the function and/or derivative values from the object, or else just call `inu` or `knu`.

The code listing for `Bessel::besselik` is in a Webnote [4].

### 6.6.3 Airy Functions

For positive  $x$ , the Airy functions are defined by

$$\text{Ai}(x) = \frac{1}{\pi} \sqrt{\frac{x}{3}} K_{1/3}(z) \quad (6.6.41)$$

$$\text{Bi}(x) = \sqrt{\frac{x}{3}} [I_{1/3}(z) + I_{-1/3}(z)] \quad (6.6.42)$$

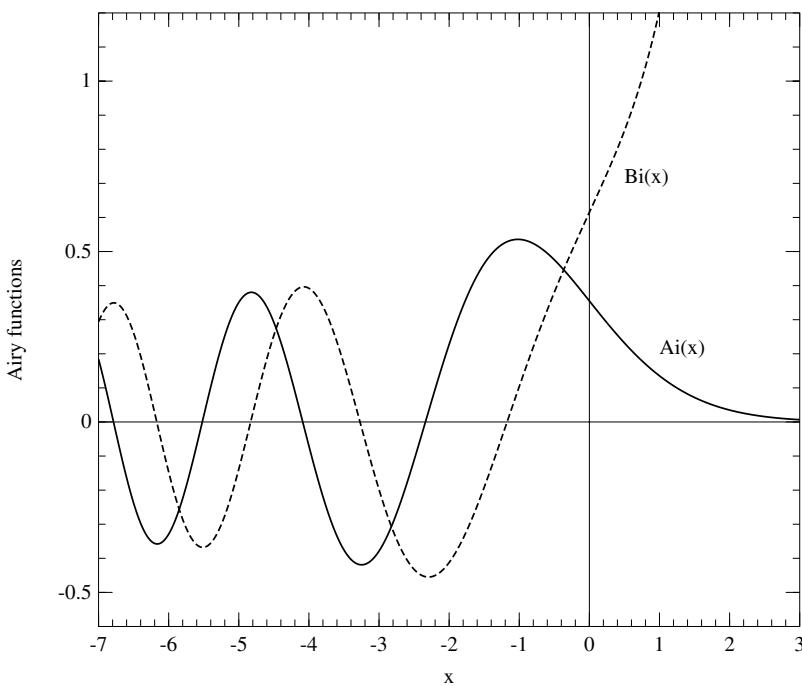
where

$$z = \frac{2}{3}x^{3/2} \quad (6.6.43)$$

By using the reflection formula (6.6.40), we can convert (6.6.42) into the computationally more useful form

$$\text{Bi}(x) = \sqrt{x} \left[ \frac{2}{\sqrt{3}} I_{1/3}(z) + \frac{1}{\pi} K_{1/3}(z) \right] \quad (6.6.44)$$

so that `Ai` and `Bi` can be evaluated with a single call to `besselik`.



**Figure 6.6.1.** Airy functions  $\text{Ai}(x)$  and  $\text{Bi}(x)$ .

The derivatives should not be evaluated by simply differentiating the above expressions because of possible subtraction errors near  $x = 0$ . Instead, use the equivalent expressions

$$\begin{aligned}\text{Ai}'(x) &= -\frac{x}{\pi\sqrt{3}}K_{2/3}(z) \\ \text{Bi}'(x) &= x\left[\frac{2}{\sqrt{3}}I_{2/3}(z) + \frac{1}{\pi}K_{2/3}(z)\right]\end{aligned}\tag{6.6.45}$$

The corresponding formulas for negative arguments are

$$\begin{aligned}\text{Ai}(-x) &= \frac{\sqrt{x}}{2}\left[J_{1/3}(z) - \frac{1}{\sqrt{3}}Y_{1/3}(z)\right] \\ \text{Bi}(-x) &= -\frac{\sqrt{x}}{2}\left[\frac{1}{\sqrt{3}}J_{1/3}(z) + Y_{1/3}(z)\right] \\ \text{Ai}'(-x) &= \frac{x}{2}\left[J_{2/3}(z) + \frac{1}{\sqrt{3}}Y_{2/3}(z)\right] \\ \text{Bi}'(-x) &= \frac{x}{2}\left[\frac{1}{\sqrt{3}}J_{2/3}(z) - Y_{2/3}(z)\right]\end{aligned}\tag{6.6.46}$$

besselfrac.h

```
void Bessel::airy(const Doub x) {
    Sets aio, bio, aipo, and bipo, respectively, to the Airy functions Ai(x), Bi(x) and their
    derivatives Ai'(x), Bi'(x).
    static const Doub PI=3.141592653589793238,
        ONOVRT=0.577350269189626,THR=1./3.,TWOTHR=2.*THR;
    Doub absx,rootx,z;
    absx=abs(x);
    rootx=sqrt(absx);
    z=TWOTHR*absx*rootx;
```

```

if (x > 0.0) {
    besselik(THR,z);
    aio = rootx*ONOVRT*ko/PI;
    bio = rootx*(ko/PI+2.0*ONOVRT*io);
    besselik(TWOTHR,z);
    aipo = -x*ONOVRT*ko/PI;
    bipo = x*(ko/PI+2.0*ONOVRT*io);
} else if (x < 0.0) {
    besseljy(THR,z);
    aio = 0.5*rootx*(jo-ONOVRT*yo);
    bio = -0.5*rootx*(yo+ONOVRT*jo);
    besseljy(TWOTHR,z);
    aipo = 0.5*absx*(ONOVRT*yo+jo);
    bipo = 0.5*absx*(ONOVRT*jo-yo);
} else {                                Case x = 0.
    aio=0.355028053887817;
    bio=aio/ONOVRT;
    aipo = -0.258819403792807;
    bipo = -aipo/ONOVRT;
}
}

Doub Bessel::airy_ai(const Doub x) {
Simple interface returning Ai(x).
    if (x != xo) airy(x);
    return aio;
}
Doub Bessel::airy_bi(const Doub x) {
Simple interface returning Bi(x).
    if (x != xo) airy(x);
    return bio;
}

```

## 6.6.4 Spherical Bessel Functions

For integer  $n$ , spherical Bessel functions are defined by

$$\begin{aligned} j_n(x) &= \sqrt{\frac{\pi}{2x}} J_{n+\frac{1}{2}}(x) \\ y_n(x) &= \sqrt{\frac{\pi}{2x}} Y_{n+\frac{1}{2}}(x) \end{aligned} \quad (6.6.47)$$

They can be evaluated by a call to `besseljy`, and the derivatives can safely be found from the derivatives of equation (6.6.47).

Note that in the continued fraction CF2 in (6.6.3) just the first term survives for  $\nu = 1/2$ . Thus one can make a very simple algorithm for spherical Bessel functions along the lines of `besseljy` by always recursing  $j_n$  down to  $n = 0$ , setting  $p$  and  $q$  from the first term in CF2, and then recursing  $y_n$  up. No special series is required near  $x = 0$ . However, `besseljy` is already so efficient that we have not bothered to provide an independent routine for spherical Bessels.

```

void Bessel::sphbes(const Int n, const Doub x) {                                besselfrac.h
Sets sphjo, sphyo, sphjpo, and sphypo, respectively, to the spherical Bessel functions  $j_n(x)$ ,
 $y_n(x)$ , and their derivatives  $j'_n(x)$ ,  $y'_n(x)$  for integer  $n$  (which is saved as sphno).
    const Doub RTPI02=1.253314137315500251;
    Doub factor,order;
    if (n < 0 || x <= 0.0) throw("bad arguments in sphbes");
    order=n+0.5;
    besseljy(order,x);
    factor=RTPI02/sqrt(x);
    sphjo=factor*jo;

```

```

sphyo=factor*yo;
sphjpo=factor*jpo-sphjo/(2.*x);
sphypo=factor*ypo-sphyo/(2.*x);
sphno = n;
}

Doub Bessel::sphbesj(const Int n, const Doub x) {
Simple interface returning  $j_n(x)$ .
    if (n != sphno || x != xo) sphbes(n,x);
    return sphjo;
}
Doub Bessel::sphbesy(const Int n, const Doub x) {
Simple interface returning  $y_n(x)$ .
    if (n != sphno || x != xo) sphbes(n,x);
    return sphyo;
}

```

**CITED REFERENCES AND FURTHER READING:**

- Barnett, A.R., Feng, D.H., Steed, J.W., and Goldfarb, L.J.B. 1974, "Coulomb Wave Functions for All Real  $\eta$  and  $\rho$ ," *Computer Physics Communications*, vol. 8, pp. 377–395.[1]
- Temme, N.M. 1976, "On the Numerical Evaluation of the Ordinary Bessel Function of the Second Kind," *Journal of Computational Physics*, vol. 21, pp. 343–350[2]; 1975, *op. cit.*, vol. 19, pp. 324–337.[3]
- Numerical Recipes Software 2007, "Bessel Function Implementations," *Numerical Recipes Web-note No. 8*, at <http://www.nr.com/webnotes?8> [4]
- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, Chapter 10.[5]
- Thompson, I.J., and Barnett, A.R. 1987, "Modified Bessel Functions  $I_\nu(z)$  and  $K_\nu(z)$  of Real Order and Complex Argument, to Selected Accuracy," *Computer Physics Communications*, vol. 47, pp. 245–257.[6]
- Barnett, A.R. 1981, "An Algorithm for Regular and Irregular Coulomb and Bessel functions of Real Order to Machine Accuracy," *Computer Physics Communications*, vol. 21, pp. 297–314.
- Thompson, I.J., and Barnett, A.R. 1986, "Coulomb and Bessel Functions of Complex Arguments and Order," *Journal of Computational Physics*, vol. 64, pp. 490–509.

## 6.7 Spherical Harmonics

Spherical harmonics occur in a large variety of physical problems, for example, whenever a wave equation, or Laplace's equation, is solved by separation of variables in spherical coordinates. The spherical harmonic  $Y_{lm}(\theta, \phi)$ ,  $-l \leq m \leq l$ , is a function of the two coordinates  $\theta, \phi$  on the surface of a sphere.

The spherical harmonics are orthogonal for different  $l$  and  $m$ , and they are normalized so that their integrated square over the sphere is unity:

$$\int_0^{2\pi} d\phi \int_{-1}^1 d(\cos \theta) Y_{l'm'}^*(\theta, \phi) Y_{lm}(\theta, \phi) = \delta_{l'l} \delta_{m'm} \quad (6.7.1)$$

Here the asterisk denotes complex conjugation.

Mathematically, the spherical harmonics are related to *associated Legendre polynomials* by the equation

$$Y_{lm}(\theta, \phi) = \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}} P_l^m(\cos \theta) e^{im\phi} \quad (6.7.2)$$

By using the relation

$$Y_{l,-m}(\theta, \phi) = (-1)^m Y_{lm}^*(\theta, \phi) \quad (6.7.3)$$

we can always relate a spherical harmonic to an associated Legendre polynomial with  $m \geq 0$ . With  $x \equiv \cos \theta$ , these are defined in terms of the ordinary Legendre polynomials (cf. §4.6 and §5.4) by

$$P_l^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_l(x) \quad (6.7.4)$$

Be careful: There are alternative normalizations for the associated Legendre polynomials and alternative sign conventions.

The first few associated Legendre polynomials, and their corresponding normalized spherical harmonics, are

|                                    |  |
|------------------------------------|--|
| $P_0^0(x) = 1$                     | $Y_{00} = \sqrt{\frac{1}{4\pi}}$   |
| $P_1^1(x) = -(1-x^2)^{1/2}$        | $Y_{11} = -\sqrt{\frac{3}{8\pi}} \sin \theta e^{i\phi}$                    |
| $P_1^0(x) = x$                     | $Y_{10} = \sqrt{\frac{3}{4\pi}} \cos \theta$                               |
| $P_2^2(x) = 3(1-x^2)$              | $Y_{22} = \frac{1}{4} \sqrt{\frac{15}{2\pi}} \sin^2 \theta e^{2i\phi}$     |
| $P_2^1(x) = -3(1-x^2)^{1/2}x$      | $Y_{21} = -\sqrt{\frac{15}{8\pi}} \sin \theta \cos \theta e^{i\phi}$       |
| $P_2^0(x) = \frac{1}{2}(3x^2 - 1)$ | $Y_{20} = \sqrt{\frac{5}{4\pi}} (\frac{3}{2} \cos^2 \theta - \frac{1}{2})$ |

(6.7.5)

There are many bad ways to evaluate associated Legendre polynomials numerically. For example, there are explicit expressions, such as

$$\begin{aligned} P_l^m(x) &= \frac{(-1)^m (l+m)!}{2^m m! (l-m)!} (1-x^2)^{m/2} \left[ 1 - \frac{(l-m)(m+l+1)}{1!(m+1)} \left( \frac{1-x}{2} \right) \right. \\ &\quad \left. + \frac{(l-m)(l-m-1)(m+l+1)(m+l+2)}{2!(m+1)(m+2)} \left( \frac{1-x}{2} \right)^2 - \dots \right] \end{aligned} \quad (6.7.6)$$

where the polynomial continues up through the term in  $(1-x)^{l-m}$ . (See [1] for this and related formulas.) This is not a satisfactory method because evaluation of the polynomial involves delicate cancellations between successive terms, which alternate in sign. For large  $l$ , the individual terms in the polynomial become very much larger than their sum, and all accuracy is lost.

In practice, (6.7.6) can be used only in single precision (32-bit) for  $l$  up to 6 or 8, and in double precision (64-bit) for  $l$  up to 15 or 18, depending on the precision required for the answer. A more robust computational procedure is therefore desirable, as follows.

The associated Legendre functions satisfy numerous recurrence relations, tabulated in [1,2]. These are recurrences on  $l$  alone, on  $m$  alone, and on both  $l$  and  $m$  simultaneously. Most of the recurrences involving  $m$  are unstable, and so are dangerous for numerical work. The following recurrence on  $l$  is, however, stable (compare 5.4.1):

$$(l - m)P_l^m = x(2l - 1)P_{l-1}^m - (l + m - 1)P_{l-2}^m \quad (6.7.7)$$

Even this recurrence is useful only for moderate  $l$  and  $m$ , since the  $P_l^m$ 's themselves grow rapidly with  $l$  and quickly overflow. The spherical harmonics by contrast remain bounded — after all, they are normalized to unity (eq. 6.7.1). It is exactly the square-root factor in equation (6.7.2) that balances the divergence. So the right function to use in the recurrence relation is the renormalized associated Legendre function,

$$\tilde{P}_l^m = \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}} P_l^m \quad (6.7.8)$$

Then the recurrence relation (6.7.7) becomes

$$\tilde{P}_l^m = \sqrt{\frac{4l^2-1}{l^2-m^2}} \left[ x\tilde{P}_{l-1}^m - \sqrt{\frac{(l-1)^2-m^2}{4(l-1)^2-1}} \tilde{P}_{l-2}^m \right] \quad (6.7.9)$$

We start the recurrence with the closed-form expression for the  $l = m$  function,

$$\tilde{P}_m^m = (-1)^m \sqrt{\frac{2m+1}{4\pi(2m)!}} (2m-1)!! (1-x^2)^{m/2} \quad (6.7.10)$$

(The notation  $n!!$  denotes the product of all *odd* integers less than or equal to  $n$ .) Using (6.7.9) with  $l = m + 1$ , and setting  $\tilde{P}_{m-1}^m = 0$ , we find

$$\tilde{P}_{m+1}^m = x\sqrt{2m+3}\tilde{P}_m^m \quad (6.7.11)$$

Equations (6.7.10) and (6.7.11) provide the two starting values required for (6.7.9) for general  $l$ .

The function that implements this is

```
plegendre.h Doub plegendre(const Int l, const Int m, const Doub x) {
    Computes the renormalized associated Legendre polynomial  $\tilde{P}_l^m(x)$ , equation (6.7.8). Here  $m$  and  $l$  are integers satisfying  $0 \leq m \leq l$ , while  $x$  lies in the range  $-1 \leq x \leq 1$ .
    static const Doub PI=3.141592653589793;
    Int i, l;
    Doub fact, oldfact, pll, pmm, pmmp1, omx2;
    if (m < 0 || m > l || abs(x) > 1.0)
        throw("Bad arguments in routine plegendre");
    pmmp1=1.0; Compute  $\tilde{P}_m^m$ .
    if (m > 0) {
        omx2=(1.0-x)*(1.0+x);
        fact=1.0;
        for (i=1; i<=m; i++) {
            pmm *= omx2*fact/(fact+1.0);
            fact += 2.0;
        }
    }
    pmm=sqrt((2*m+1)*pmm/(4.0*PI));
```

```

    if (m & 1)
        pmm=-pmm;
    if (l == m)
        return pmm;
    else {                                Compute  $\tilde{P}_{m+1}^m$ .
        pmmp1=x*sqrt(2.0*m+3.0)*pmm;
        if (l == (m+1))
            return pmmp1;
        else {                            Compute  $\tilde{P}_l^m$ ,  $l > m + 1$ .
            oldfact=sqrt(2.0*m+3.0);
            for (ll=m+2;ll<=l;ll++) {
                fact=sqrt((4.0*ll*ll-1.0)/(ll*ll-m*m));
                pll=(x*pmmp1-pmm/oldfact)*fact;
                oldfact=fact;
                pmm=pmmp1;
                pmmp1=pll;
            }
            return pll;
        }
    }
}

```

Sometimes it is convenient to have the functions with the standard normalization, as defined by equation (6.7.4). Here is a routine that does this. Note that it will overflow for  $m \gtrsim 80$ , or even sooner if  $l \gg m$ .

```

Doub plgndr(const Int l, const Int m, const Doub x)
Computes the associated Legendre polynomial  $P_l^m(x)$ , equation (6.7.4). Here  $m$  and  $l$  are
integers satisfying  $0 \leq m \leq l$ , while  $x$  lies in the range  $-1 \leq x \leq 1$ . These functions will
overflow for  $m \gtrsim 80$ .
{
    const Doub PI=3.141592653589793238;
    if (m < 0 || m > l || abs(x) > 1.0)
        throw("Bad arguments in routine plgndr");
    Doub prod=1.0;
    for (Int j=l-m+1;j<=l+m;j++)
        prod *= j;
    return sqrt(4.0*PI*prod/(2*l+1))*plegendre(l,m,x);
}

```

plegendre.h

### 6.7.1 Fast Spherical Harmonic Transforms

Any smooth function on the surface of a sphere can be written as an expansion in spherical harmonics. Suppose the function can be well-approximated by truncating the expansion at  $l = l_{\max}$ :

$$\begin{aligned}
 f(\theta_i, \phi_j) &= \sum_{l=0}^{l_{\max}} \sum_{m=-l}^{m=l} a_{lm} Y_{lm}(\theta_i, \phi_j) \\
 &= \sum_{l=0}^{l_{\max}} \sum_{m=-l}^{m=l} a_{lm} \tilde{P}_l^m(\cos \theta_i) e^{im\phi_j}
 \end{aligned} \tag{6.7.12}$$

Here we have written the function evaluated at one of  $N_\theta$  sample points  $\theta_i$  and one of  $N_\phi$  sample points  $\phi_j$ . The total number of sample points is  $N = N_\theta N_\phi$ . In applications, typically  $N_\theta \sim N_\phi \sim \sqrt{N}$ . Since the total number of spherical harmonics in the sum (6.7.12) is  $l_{\max}^2$ , we also have  $l_{\max} \sim \sqrt{N}$ .

How many operations does it take to evaluate the sum (6.7.12)? Direct evaluation of  $l_{\max}^2$  terms at  $N$  sample points is an  $O(N^2)$  process. You might try to speed this up by choosing the sample points  $\phi_j$  to be equally spaced in angle and doing the sum over  $m$  by an FFT. Each FFT is  $O(N_\phi \ln N_\phi)$ , and you have to do  $O(N_\theta l_{\max})$  of them, for a total of  $O(N^{3/2} \ln N)$  operations, which is some improvement. A simple rearrangement [3-5] gives an even better way: Interchange the order of summation

$$\sum_{l=0}^{l_{\max}} \sum_{m=-l}^l \longleftrightarrow \sum_{m=-l_{\max}}^{l_{\max}} \sum_{l=|m|}^{l_{\max}} \quad (6.7.13)$$

so that

$$f(\theta_i, \phi_j) = \sum_{m=-l_{\max}}^{l_{\max}} q_m(\theta_i) e^{im\phi_j} \quad (6.7.14)$$

where

$$q_m(\theta_i) = \sum_{l=|m|}^{l_{\max}} a_{lm} \tilde{P}_l^m(\cos \theta_i) \quad (6.7.15)$$

Evaluating the sum in (6.7.15) is  $O(l_{\max})$ , and one must do this for  $O(l_{\max} N_\theta)$   $q_m$ 's, so the total work is  $O(N^{3/2})$ . To evaluate equation (6.7.14) by an FFT at fixed  $\theta_i$  is  $O(N_\phi \ln N_\phi)$ . There are  $N_\theta$  FFTs to be done, for a total operations count of  $O(N \ln N)$ , which is negligible in comparison. So the total algorithm is  $O(N^{3/2})$ . Note that you can evaluate equation (6.7.14) either by precomputing and storing the  $\tilde{P}_l^m$ 's using the recurrence relation (6.7.9), or by Clenshaw's method (§5.4).

What about inverting the transform (6.7.12)? The formal inverse for the expansion of a continuous function  $f(\theta, \phi)$  follows from the orthonormality of the  $Y_{lm}$ 's, equation (6.7.1),

$$a_{lm} = \int \sin \theta d\theta d\phi f(\theta, \phi) e^{-im\phi} \tilde{P}_l^m(\cos \theta) \quad (6.7.16)$$

For the discrete case, where we have a sampled function, the integral becomes a quadrature:

$$a_{lm} = \sum_{i,j} w(\theta_i) f(\theta_i, \phi_j) e^{-im\phi_j} \tilde{P}_l^m(\cos \theta_i) \quad (6.7.17)$$

Here  $w(\theta_i)$  are the quadrature weights. In principle we could consider weights that depend on  $\phi_j$  as well, but in practice we do the  $\phi$  quadrature by an FFT, so the weights are unity. A good choice for the weights for an equi-angular grid in  $\theta$  is given in Ref. [3], Theorem 3. Another possibility is to use Gaussian quadrature for the  $\theta$  integral. In this case, you choose the sample points so that the  $\cos \theta_i$ 's are the abscissas returned by `gauleg` and the  $w(\theta_i)$ 's are the corresponding weights. The best way to organize the calculation is to first do the FFTs, computing

$$g_m(\theta_i) = \sum_j f(\theta_i, \phi_j) e^{-im\phi_j} \quad (6.7.18)$$

Then

$$a_{lm} = \sum_i w(\theta_i) g_m(\theta_i) \tilde{P}_l^m(\cos \theta_i) \quad (6.7.19)$$

You can verify that the operations count is dominated by equation (6.7.19) and scales as  $O(N^{3/2})$  once again. In a real calculation, you should exploit all the symmetries that let you reduce the workload, such as  $g_{-m} = g_m^*$  and  $\tilde{P}_l^m[\cos(\pi - \theta)] = (-1)^{l+m} \tilde{P}_l^m(\cos \theta)$ .

Very recently, algorithms for fast Legendre transforms have been developed, similar in spirit to the FFT [3,6,7]. Theoretically, they reduce the forward and inverse spherical harmonic transforms to  $O(N \log^2 N)$  problems. However, current implementations [8] are not much faster than the  $O(N^{3/2})$  methods above for  $N \sim 500$ , and there are stability and accuracy problems that require careful attention [9]. Stay tuned!

#### CITED REFERENCES AND FURTHER READING:

- Magnus, W., and Oberhettinger, F. 1949, *Formulas and Theorems for the Functions of Mathematical Physics* (New York: Chelsea), pp. 54ff.[1]
- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, Chapter 8.[2]
- Driscoll, J.R., and Healy, D.M. 1994, “Computing Fourier Transforms and Convolutions on the 2-sphere,” *Advances in Applied Mathematics*, vol. 15, pp. 202–250.[3]
- Muciaccia, P.F., Natoli, P., and Vittorio, N. 1997, “Fast Spherical Harmonic Analysis: A Quick Algorithm for Generating and/or Inverting Full-Sky, High-Resolution Cosmic Microwave Background Anisotropy Maps,” *Astrophysical Journal*, vol. 488, pp. L63–66.[4]
- Oh, S.P., Spergel, D.N., and Hinshaw, G. 1999, “An Efficient Technique to Determine the Power Spectrum from Cosmic Microwave Background Sky Maps,” *Astrophysical Journal*, vol. 510, pp. 551–563, Appendix A.[5]
- Healy, D.M., Rockmore, D., Kostelec, P.J., and Moore, S. 2003, “FFTs for the 2-Sphere: Improvements and Variations,” *Journal of Fourier Analysis and Applications*, vol. 9, pp. 341–385.[6]
- Potts, D., Steidl, G., and Tasche, M. 1998, “Fast and Stable Algorithms for Discrete Spherical Fourier Transforms,” *Linear Algebra and Its Applications*, vol. 275-276, pp. 433–450.[7]
- Moore, S., Healy, D.M., Rockmore, D., and Kostelec, P.J. 2007+, *SpharmonicKit*. Software at <http://www.cs.dartmouth.edu/~geelong/sphere>.[8]
- Healy, D.M., Kostelec, P.J., and Rockmore, D. 2004, “Towards Safe and Effective High-Order Legendre Transforms with Applications to FFTs for the 2-Sphere,” *Advances in Computational Mathematics*, vol. 21, pp. 59–105.[9]

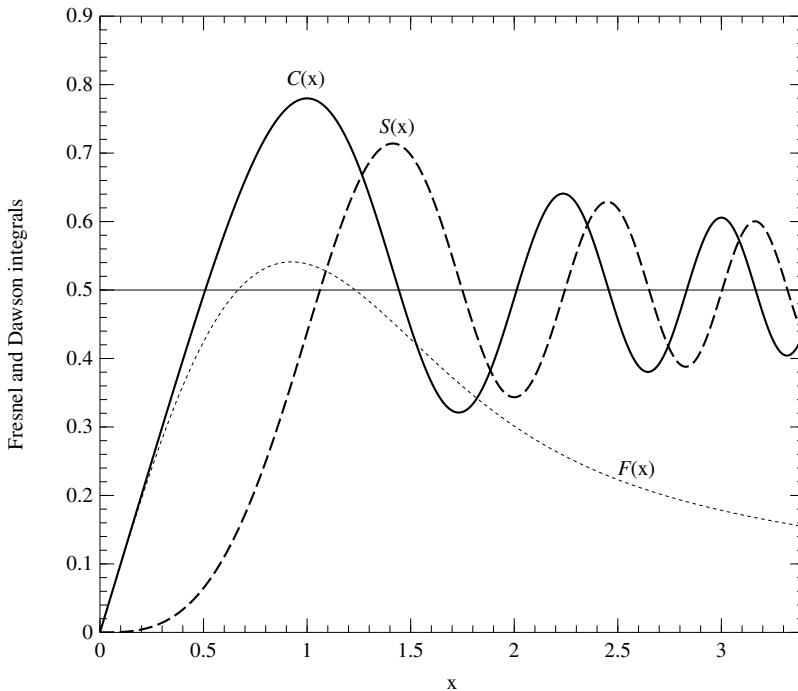
## 6.8 Fresnel Integrals, Cosine and Sine Integrals

### 6.8.1 Fresnel Integrals

The two Fresnel integrals are defined by

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right) dt, \quad S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right) dt \quad (6.8.1)$$

and are plotted in Figure 6.8.1.



**Figure 6.8.1.** Fresnel integrals  $C(x)$  and  $S(x)$  (§6.8), and Dawson's integral  $F(x)$  (§6.9).

The most convenient way of evaluating these functions to arbitrary precision is to use power series for small  $x$  and a continued fraction for large  $x$ . The series are

$$\begin{aligned} C(x) &= x - \left(\frac{\pi}{2}\right)^2 \frac{x^5}{5 \cdot 2!} + \left(\frac{\pi}{2}\right)^4 \frac{x^9}{9 \cdot 4!} - \dots \\ S(x) &= \left(\frac{\pi}{2}\right) \frac{x^3}{3 \cdot 1!} - \left(\frac{\pi}{2}\right)^3 \frac{x^7}{7 \cdot 3!} + \left(\frac{\pi}{2}\right)^5 \frac{x^{11}}{11 \cdot 5!} - \dots \end{aligned} \quad (6.8.2)$$

There is a complex continued fraction that yields both  $S(x)$  and  $C(x)$  simultaneously:

$$C(x) + iS(x) = \frac{1+i}{2} \operatorname{erf} z, \quad z = \frac{\sqrt{\pi}}{2}(1-i)x \quad (6.8.3)$$

where

$$\begin{aligned} e^{z^2} \operatorname{erfc} z &= \frac{1}{\sqrt{\pi}} \left( \frac{1}{z+} \frac{1/2}{z+} \frac{1}{z+} \frac{3/2}{z+} \frac{2}{z+} \dots \right) \\ &= \frac{2z}{\sqrt{\pi}} \left( \frac{1}{2z^2+1-} \frac{1 \cdot 2}{2z^2+5-} \frac{3 \cdot 4}{2z^2+9-} \dots \right) \end{aligned} \quad (6.8.4)$$

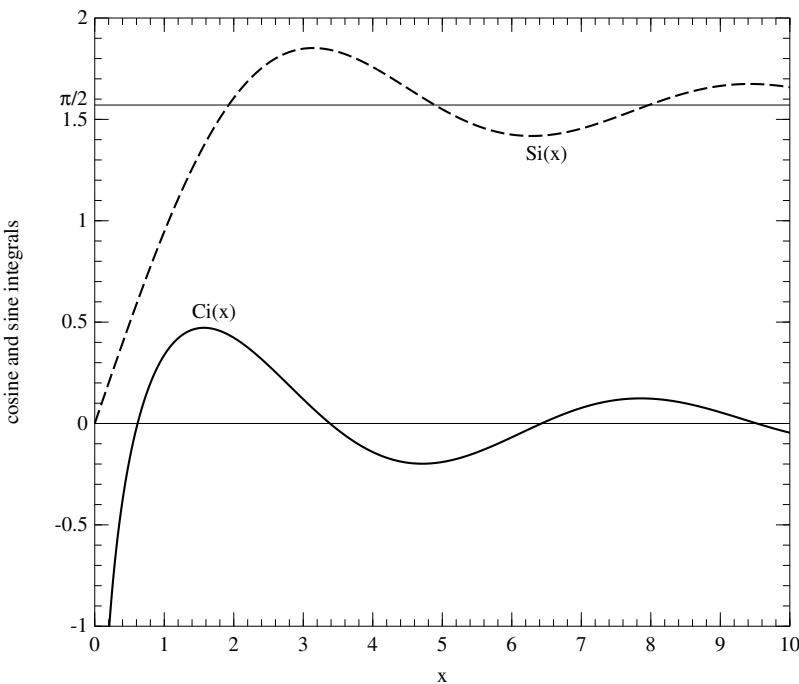
In the last line we have converted the “standard” form of the continued fraction to its “even” form (see §5.2), which converges twice as fast. We must be careful not to evaluate the alternating series (6.8.2) at too large a value of  $x$ ; inspection of the terms shows that  $x = 1.5$  is a good point to switch over to the continued fraction.

Note that for large  $x$

$$C(x) \sim \frac{1}{2} + \frac{1}{\pi x} \sin \left( \frac{\pi}{2} x^2 \right), \quad S(x) \sim \frac{1}{2} - \frac{1}{\pi x} \cos \left( \frac{\pi}{2} x^2 \right) \quad (6.8.5)$$

Thus the precision of the routine `frenel` may be limited by the precision of the library routines for sine and cosine for large  $x$ .

```
Complex frenel(const Doub x) { frenel.h
    Computes the Fresnel integrals  $S(x)$  and  $C(x)$  for all real  $x$ .  $C(x)$  is returned as the real part
    of cs and  $S(x)$  as the imaginary part.
    static const Int MAXIT=100;
    static const Doub PI=3.141592653589793238, PIBY2=(PI/2.0), XMIN=1.5,
        EPS=numeric_limits<Doub>::epsilon(),
        FPMIN=numeric_limits<Doub>::min(),
        BIG=numeric_limits<Doub>::max()*EPS;
    Here MAXIT is the maximum number of iterations allowed; EPS is the relative error;
    FPMIN is a number near the smallest representable floating-point number; BIG is a
    number near the machine overflow limit; and XMIN is the dividing line between using
    the series and continued fraction.
    Bool odd;
    Int k,n;
    Doub a,ax,fact,pix2,sign,sum,sumc,sums,term,test;
    Complex b,cc,d,h,del,cs;
    if ((ax=abs(x)) < sqrt(FPMIN)) {
        cs=ax; Special case: Avoid failure of convergence
    } else if (ax <= XMIN) { test because of underflow.
        sum=sums=0.0; Evaluate both series simultaneously.
        sumc=ax;
        sign=1.0;
        fact=PIBY2*ax*ax;
        odd=true;
        term=ax;
        n=3;
        for (k=1;k<=MAXIT;k++) {
            term *= fact/k;
            sum += sign*term/n;
            test=abs(sum)*EPS;
            if (odd) {
                sign = -sign;
                sums=sum;
                sum=sumc;
            } else {
                sumc=sum;
                sum=sums;
            }
            if (term < test) break;
            odd=!odd;
            n += 2;
        }
        if (k > MAXIT) throw("series failed in frenel");
        cs=Complex(sumc,sums); Evaluate continued fraction by modified
    } else { Lentz's method (§5.2).
        pix2=PI*ax*ax;
        b=Complex(1.0,-pix2);
        cc=BIG;
        d=h=1.0/b;
        n = -1;
        for (k=2;k<=MAXIT;k++) {
            n += 2;
            a = -n*(n+1);
            b += 4.0;
            d=1.0/(a*d+b); Denominators cannot be zero.
            cc=b+a/cc;
            del=cc*d;
            h *= del;
            if (abs(real(del)-1.0)+abs(imag(del)) <= EPS) break;
        }
    }
}
```



**Figure 6.8.2.** Sine and cosine integrals  $\text{Si}(x)$  and  $\text{Ci}(x)$ .

```

if (k > MAXIT) throw("cf failed in fresnel");
h *= Complex(ax,-ax);
cs=Complex(0.5,0.5)
    *(1.0-Complex(cos(0.5*pix2),sin(0.5*pix2))*h);
}
if (x < 0.0) cs = -cs;                                Use antisymmetry.
return cs;
}

```

## 6.8.2 Cosine and Sine Integrals

The cosine and sine integrals are defined by

$$\begin{aligned} \text{Ci}(x) &= \gamma + \ln x + \int_0^x \frac{\cos t - 1}{t} dt \\ \text{Si}(x) &= \int_0^x \frac{\sin t}{t} dt \end{aligned} \tag{6.8.6}$$

and are plotted in Figure 6.8.2. Here  $\gamma \approx 0.5772\dots$  is Euler's constant. We only need a way to calculate the functions for  $x > 0$ , because

$$\text{Si}(-x) = -\text{Si}(x), \quad \text{Ci}(-x) = \text{Ci}(x) - i\pi \tag{6.8.7}$$

Once again we can evaluate these functions by a judicious combination of power

series and complex continued fraction. The series are

$$\begin{aligned} \text{Si}(x) &= x - \frac{x^3}{3 \cdot 3!} + \frac{x^5}{5 \cdot 5!} - \dots \\ \text{Ci}(x) &= \gamma + \ln x + \left( -\frac{x^2}{2 \cdot 2!} + \frac{x^4}{4 \cdot 4!} - \dots \right) \end{aligned} \quad (6.8.8)$$

The continued fraction for the exponential integral  $E_1(ix)$  is

$$\begin{aligned} E_1(ix) &= -\text{Ci}(x) + i[\text{Si}(x) - \pi/2] \\ &= e^{-ix} \left( \frac{1}{ix+} \frac{1}{1+} \frac{1}{ix+} \frac{2}{1+} \frac{2}{ix+} \dots \right) \\ &= e^{-ix} \left( \frac{1}{1+ix-} \frac{1^2}{3+ix-} \frac{2^2}{5+ix-} \dots \right) \end{aligned} \quad (6.8.9)$$

The “even” form of the continued fraction is given in the last line and converges twice as fast for about the same amount of computation. A good crossover point from the alternating series to the continued fraction is  $x = 2$  in this case. As for the Fresnel integrals, for large  $x$  the precision may be limited by the precision of the sine and cosine routines.

**Complex cisi(const Doub x) {**

Computes the cosine and sine integrals  $\text{Ci}(x)$  and  $\text{Si}(x)$ . The function  $\text{Ci}(x)$  is returned as the real part of  $\text{cs}$ , and  $\text{Si}(x)$  as the imaginary part.  $\text{Ci}(0)$  is returned as a large negative number and no error message is generated. For  $x < 0$  the routine returns  $\text{Ci}(-x)$  and you must supply the  $-i\pi$  yourself.

**static const Int MAXIT=100;** Maximum number of iterations allowed.  
**static const Doub EULER=0.577215664901533, PIBY2=1.570796326794897,**

**TMIN=2.0, EPS=numeric\_limits<Doub>::epsilon(),**  
**FPMIN=numeric\_limits<Doub>::min()\*4.0,**  
**BIG=numeric\_limits<Doub>::max()\*EPS;**

Here  $EULER$  is Euler's constant  $\gamma$ ;  $PIBY2$  is  $\pi/2$ ;  $TMIN$  is the dividing line between using the series and continued fraction;  $EPS$  is the relative error, or absolute error near a zero of  $\text{Ci}(x)$ ;  $FPMIN$  is a number close to the smallest representable floating-point number; and  $BIG$  is a number near the machine overflow limit.

```

Int i,k;
Bool odd;
Doub a,err,fact,sign,sum,sumc,sums,t,term;
Complex h,b,c,d,del,cs;
if ((t=abs(x)) == 0.0) return -BIG; Special case.
if (t > TMIN) { Evaluate continued fraction by modified
    b=Complex(1.0,t); Lentz's method (§5.2).
    c=Complex(BIG,0.0);
    d=h=1.0/b;
    for (i=1;i<MAXIT;i++) {
        a= -i*i;
        b += 2.0;
        d=1.0/(a*d+b); Denominators cannot be zero.
        c=b+a/c;
        del=c*d;
        h *= del;
        if (abs(real(del)-1.0)+abs(imag(del)) <= EPS) break;
    }
    if (i >= MAXIT) throw("cf failed in cisi");
    h=Complex(cos(t),-sin(t))*h;
    cs= -conj(h)+Complex(0.0,PIBY2);
} else { Evaluate both series simultaneously.

```

cisi.h

```

if (t < sqrt(FPMIN)) {
    sumc=0.0;
    sums=t;
} else {
    sum=sums=sumc=0.0;
    sign=fact=1.0;
    odd=true;
    for (k=1;k<=MAXIT;k++) {
        fact *= t/k;
        term=fact/k;
        sum += sign*term;
        err=term/abs(sum);
        if (odd) {
            sign = -sign;
            sums=sum;
            sum=sumc;
        } else {
            sumc=sum;
            sum=sums;
        }
        if (err < EPS) break;
        odd=!odd;
    }
    if (k > MAXIT) throw("maxits exceeded in cisi");
}
cs=Complex(sumc+log(t)+EULER,sums);
}
if (x < 0.0) cs = conj(cs);
return cs;
}

```

Special case: Avoid failure of convergence test because of underflow.

**CITED REFERENCES AND FURTHER READING:**

- Stegun, I.A., and Zucker, R. 1976, "Automatic Computing Methods for Special Functions. III. The Sine, Cosine, Exponential integrals, and Related Functions," *Journal of Research of the National Bureau of Standards*, vol. 80B, pp. 291–311; 1981, "Automatic Computing Methods for Special Functions. IV. Complex Error Function, Fresnel Integrals, and Other Related Functions," *op. cit.*, vol. 86, pp. 661–686.
- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, Chapters 5 and 7.

## 6.9 Dawson's Integral

*Dawson's Integral*  $F(x)$  is defined by

$$F(x) = e^{-x^2} \int_0^x e^{t^2} dt \quad (6.9.1)$$

See Figure 6.8.1 for a graph of the function. The function can also be related to the complex error function by

$$F(z) = \frac{i\sqrt{\pi}}{2} e^{-z^2} [1 - \operatorname{erfc}(-iz)]. \quad (6.9.2)$$

A remarkable approximation for  $F(z)$ , due to Rybicki [1], is

$$F(z) = \lim_{h \rightarrow 0} \frac{1}{\sqrt{\pi}} \sum_{n \text{ odd}} \frac{e^{-(z-nh)^2}}{n} \quad (6.9.3)$$

What makes equation (6.9.3) unusual is that its accuracy increases *exponentially* as  $h$  gets small, so that quite moderate values of  $h$  (and correspondingly quite rapid convergence of the series) give very accurate approximations.

We will discuss the theory that leads to equation (6.9.3) later, in §13.11, as an interesting application of Fourier methods. Here we simply implement a routine for real values of  $x$  based on the formula.

It is first convenient to shift the summation index to center it approximately on the maximum of the exponential term. Define  $n_0$  to be the even integer nearest to  $x/h$ , and  $x_0 \equiv n_0 h$ ,  $x' \equiv x - x_0$ , and  $n' \equiv n - n_0$ , so that

$$F(x) \approx \frac{1}{\sqrt{\pi}} \sum_{n'=-N; \text{ odd}}^N \frac{e^{-(x'-n'h)^2}}{n' + n_0} \quad (6.9.4)$$

where the approximate equality is accurate when  $h$  is sufficiently small and  $N$  is sufficiently large. The computation of this formula can be greatly speeded up if we note that

$$e^{-(x'-n'h)^2} = e^{-x'^2} e^{-(n'h)^2} \left( e^{2x'h} \right)^{n'} \quad (6.9.5)$$

The first factor is computed once, the second is an array of constants to be stored, and the third can be computed recursively, so that only two exponentials need be evaluated. Advantage is also taken of the symmetry of the coefficients  $e^{-(n'h)^2}$  by breaking up the summation into positive and negative values of  $n'$  separately.

In the following routine, the choices  $h = 0.4$  and  $N = 11$  are made. Because of the symmetry of the summations and the restriction to odd values of  $n$ , the limits on the `for` loops are 0 to 5. The accuracy of the result in this version is about  $2 \times 10^{-7}$ . In order to maintain relative accuracy near  $x = 0$ , where  $F(x)$  vanishes, the program branches to the evaluation of the power series [2] for  $F(x)$ , for  $|x| < 0.2$ .

```
Doub dawson(const Doub x) {
    Returns Dawson's integral  $F(x) = \exp(-x^2) \int_0^x \exp(t^2) dt$  for any real  $x$ .
    static const Int NMAX=6;
    static VecDoub c(NMAX);
    static Bool init = true;
    static const Doub H=0.4, A1=2.0/3.0, A2=0.4, A3=2.0/7.0;
    Int i,n0;                               Flag is true if we need to initialize, else false.
    Doub d1,d2,e1,e2,sum,x2,xp,xx,ans;
    if (init) {
        init=false;
        for (i=0;i<NMAX;i++) c[i]=exp(-SQR((2.0*i+1.0)*H));
    }
    if (abs(x) < 0.2) {                   Use series expansion.
        x2=x*x;
        ans=x*(1.0-A1*x2*(1.0-A2*x2*(1.0-A3*x2)));
    } else {                                Use sampling theorem representation.
        xx=abs(x);
        n0=2*Int(0.5*xx/H+0.5);
        xp=xx-n0*H;
        e1=exp(2.0*xp*H);
    }
}
```

```

e2=e1*e1;
d1=n0+1;
d2=d1-2.0;
sum=0.0;
for (i=0;i<NMAX;i++,d1+=2.0,d2-=2.0,e1*=e2)
    sum += c[i]*(e1/d1+1.0/(d2*e1));
ans=0.5641895835*SIGN(exp(-xp*xp),x)*sum;           Constant is 1/sqrt(pi).
}
return ans;
}

```

Other methods for computing Dawson's integral are also known [2,3].

#### CITED REFERENCES AND FURTHER READING:

- Rybicki, G.B. 1989, "Dawson's Integral and The Sampling Theorem," *Computers in Physics*, vol. 3, no. 2, pp. 85–87.[1]  
 Cody, W.J., Pociorek, K.A., and Thatcher, H.C. 1970, "Chebyshev Approximations for Dawson's Integral," *Mathematics of Computation*, vol. 24, pp. 171–178.[2]  
 McCabe, J.H. 1974, "A Continued Fraction Expansion, with a Truncation Error Estimate, for Dawson's Integral," *Mathematics of Computation*, vol. 28, pp. 811–816.[3]

## 6.10 Generalized Fermi-Dirac Integrals

The generalized Fermi-Dirac integral is defined as

$$F_k(\eta, \theta) = \int_0^\infty \frac{x^k (1 + \frac{1}{2} \theta x)^{1/2}}{e^{x-\eta} + 1} dx \quad (6.10.1)$$

It occurs, for example, in astrophysical applications with  $\theta$  nonnegative and arbitrary  $\eta$ . In condensed matter physics one usually has the simpler case of  $\theta = 0$  and omits the “generalized” from the name of the function. The important values of  $k$  are  $-1/2, 1/2, 3/2$ , and  $5/2$ , but we'll consider arbitrary values greater than  $-1$ . Watch out for an alternative definition that multiplies the integral by  $1/\Gamma(k+1)$ .

For  $\eta \ll -1$  and  $\eta \gg 1$  there are useful series expansions for these functions (see, e.g., [1]). These give, for example,

$$\begin{aligned} F_{1/2}(\eta, \theta) &\rightarrow \frac{1}{\sqrt{2\theta}} e^\eta e^{1/\theta} K_1\left(\frac{1}{\theta}\right), \quad \eta \rightarrow -\infty \\ F_{1/2}(\eta, \theta) &\rightarrow \frac{1}{2\sqrt{2}} \eta^{3/2} y \frac{\sqrt{1+y^2} - \sinh^{-1} y}{(\sqrt{1+y^2} - 1)^{3/2}}, \quad \eta \rightarrow \infty \end{aligned} \quad (6.10.2)$$

Here  $y$  is defined by

$$1 + y^2 = (1 + \eta\theta)^2 \quad (6.10.3)$$

It is the middle range of  $\eta$  values that is difficult to handle.

For  $\theta = 0$ , Macleod [2] has given Chebyshev expansions accurate to  $10^{-16}$  for the four important  $k$  values, covering all  $\eta$  values. In this case, one need look no further for an algorithm. Goano [3] handles arbitrary  $k$  for  $\theta = 0$ . For nonzero  $\theta$ ,

it is reasonable to compute the functions by direct integration, using variable transformation to get rapidly converging quadratures [4]. (Of course, this works also for  $\theta = 0$ , but is not as efficient.) The usual transformation  $x = \exp(t - e^{-t})$  handles the singularity at  $x = 0$  and the exponential fall off at large  $x$  (cf. equation 4.5.14). For  $\eta \gtrsim 15$ , it is better to split the integral into two regions,  $[0, \eta]$  and  $[\eta, \eta + 60]$ . (The contribution beyond  $\eta + 60$  is negligible.) Each of these integrals can then be done with the DE rule. Between 60 and 500 function evaluations give full double precision, larger  $\eta$  requiring more function evaluations. A more efficient strategy would replace the quadrature by a series expansion for large  $\eta$ .

In the implementation below, note how `operator()` is overloaded to define both a function of one variable (for `Trapzd`) and a function of two variables (for `DERule`). Note also the syntax

```
Trapzd<Fermi> s(*this,a,b);
```

for declaring a `Trapzd` object inside the `Fermi` object itself.

```
struct Fermi {
    Doub kk,etaa,thetaa;
    Doub operator() (const Doub t);
    Doub operator() (const Doub x, const Doub del);
    Doub val(const Doub k, const Doub eta, const Doub theta);
};

Doub Fermi::operator() (const Doub t) {
    Integrand for trapezoidal quadrature of generalized Fermi-Dirac integral with transformation
     $x = \exp(t - e^{-t})$ .
    Doub x;
    x=exp(t-exp(-t));
    return x*(1.0+exp(-t))*pow(x,kk)*sqrt(1.0+thetaa*0.5*x)/
        (exp(x-etaa)+1.0);
}

Doub Fermi::operator() (const Doub x, const Doub del) {
    Integrand for DE rule quadrature of generalized Fermi-Dirac integral.
    if (x < 1.0)
        return pow(del,kk)*sqrt(1.0+thetaa*0.5*x)/(exp(x-etaa)+1.0);
    else
        return pow(x,kk)*sqrt(1.0+thetaa*0.5*x)/(exp(x-etaa)+1.0);
}

Doub Fermi::val(const Doub k, const Doub eta, const Doub theta)
    Computes the generalized Fermi-Dirac integral  $F_k(\eta, \theta)$ , where  $k > -1$  and  $\theta \geq 0$ . The
    accuracy is approximately the square of the parameter EPS. NMAX limits the total number of
    quadrature steps.
{
    const Doub EPS=3.0e-9;
    const Int NMAX=11;
    Doub a,aa,b,bb,hmax,olds,sum;
    kk=k;                                Load the arguments into the member variables
    etaa=eta;                            for use in the function evaluations.
    thetaa=theta;
    if (eta <= 15.0) {                    Set limits for  $x = \exp(t - e^{-t})$  mapping.
        a=-4.5;
        b=5.0;
        Trapzd<Fermi> s(*this,a,b);
        for (Int i=1;i<=NMAX;i++) {
            sum=s.next();
            if (i > 3)                  Test for convergence.
                if (abs(sum-olds) <= EPS*abs(olds))

```

```

        return sum;
    olds=sum;           Save value for next convergence test.
}
else {
    a=0.0;            Set limits for DE rule.
    b=eta;
    aa=eta;
    bb=eta+60.0;
    hmax=4.3;         Big enough to handle negative  $k$  or large  $\eta$ .
    DERule<Fermi> s(*this,a,b,hmax);
    DERule<Fermi> ss(*this,aa,bb,hmax);
    for (Int i=1;i<=NMAX;i++) {
        sum=s.next()+ss.next();
        if (i > 3)
            if (abs(sum-olds) <= EPS*abs(olds))
                return sum;
        olds=sum;
    }
}
throw("no convergence in fermi");
return 0.0;
}

```

You get values of the Fermi-Dirac functions by declaring a Fermi object:

```
Fermi ferm;
```

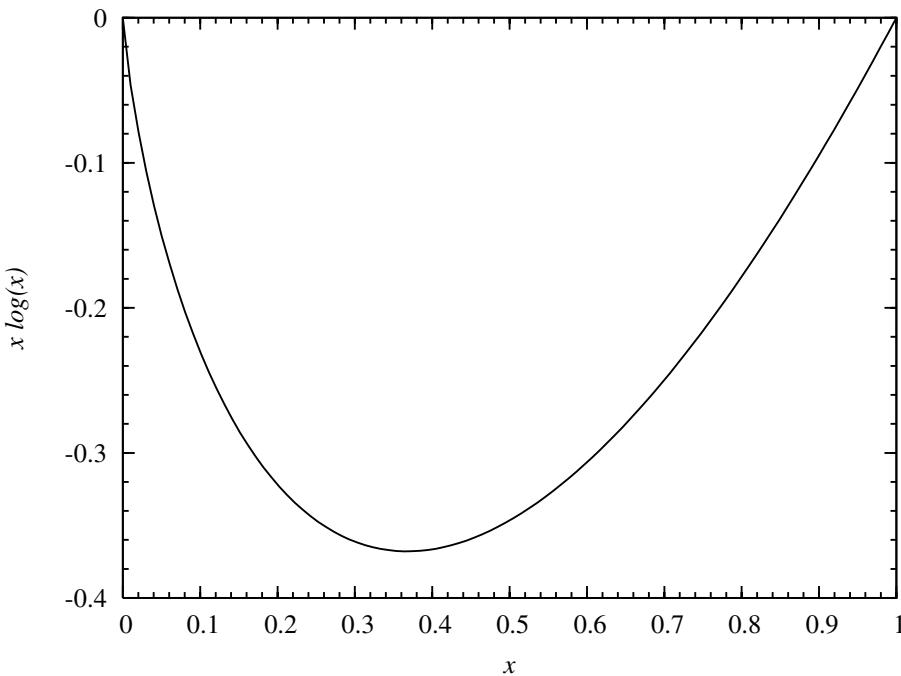
and then making repeated calls to the val function:

```
ans=ferm.val(k,eta,theta);
```

Other quadrature methods exist for these functions [5-7]. A reasonably efficient method [8] involves trapezoidal quadrature with “pole correction,” but it is restricted to  $\theta \lesssim 0.2$ . Generalized Bose-Einstein integrals can also be computed by the DE rule or the methods in these references.

#### CITED REFERENCES AND FURTHER READING:

- Cox, J.P., and Giuli, R.T. 1968, *Principles of Stellar Structure* (New York: Gordon and Breach), vol. II, §24.7.[1]
- Macleod, A.J. 1998, “Fermi-Dirac Functions of Order  $-1/2, 1/2, 3/2, 5/2$ ,” *ACM Transactions on Mathematical Software*, vol. 24, pp. 1–12. (Algorithm 779, available from netlib.)[2]
- Goano, M. 1995, “Computation of the Complete and Incomplete Fermi-Dirac Integral,” *ACM Transactions on Mathematical Software*, vol. 21, pp. 221–232. (Algorithm 745, available from netlib.)[3]
- Natarajan, A., and Kumar, N.M. 1993, “On the Numerical Evaluation of the Generalised Fermi-Dirac Integrals,” *Computer Physics Communications*, vol. 76, pp. 48–50.[4]
- Pichon, B. 1989, “Numerical Calculation of the Generalized Fermi-Dirac Integrals,” *Computer Physics Communications*, vol. 55, pp. 127–136.[5]
- Sagar, R.P. 1991, “A Gaussian Quadrature for the Calculation of Generalized Fermi-Dirac Integrals,” *Computer Physics Communications*, vol. 66, pp. 271–275.[6]
- Gautschi, W. 1992, “On the Computation of Generalized Fermi-Dirac and Bose-Einstein Integrals,” *Computer Physics Communications*, vol. 74, pp. 233–238.[7]
- Mohankumar, N., and Natarajan, A. 1996, “A Note on the Evaluation of the Generalized Fermi-Dirac Integral,” *Astrophysical Journal*, vol. 458, pp. 233–235.[8]



**Figure 6.11.1.** The function  $x \log(x)$  is shown for  $0 < x < 1$ . Although nearly invisible, an essential singularity at  $x = 0$  makes this function tricky to invert.

## 6.11 Inverse of the Function $x \log(x)$

The function

$$y(x) = x \log(x) \quad (6.11.1)$$

and its inverse function  $x(y)$  occur in a number of statistical and information theoretical contexts. Obviously  $y(x)$  is nonsingular for all positive  $x$ , and easy to evaluate. For  $x$  between 0 and 1, it is negative, with a single minimum at  $(x, y) = (e^{-1}, -e^{-1})$ . The function has the value 0 at  $x = 1$ , and it has the value 0 as its limit at  $x = 0$ , since the linear factor  $x$  easily dominates the singular logarithm.

Computing the inverse function  $x(y)$  is, however, not so easy. (We will need this inverse in §6.14.12.) From the appearance of Figure 6.11.1, it might seem easy to invert the function on its left branch, that is, return a value  $x$  between 0 and  $e^{-1}$  for every value  $y$  between 0 and  $-e^{-1}$ . However, the lurking logarithmic singularity at  $x = 0$  causes difficulties for many methods that you might try.

Polynomial fits work well over any range of  $y$  that is less than a decade or so (e.g., from 0.01 to 0.1), but fail badly if you demand high fractional precision extending all the way to  $y = 0$ .

What about Newton's method? We write

$$\begin{aligned} f(x) &\equiv x \log(x) - y \\ f'(x) &= 1 + \log(x) \end{aligned} \quad (6.11.2)$$

giving the iteration

$$x_{i+1} = x_i - \frac{x_i \log(x_i) - y}{1 + \log(x_i)} \quad (6.11.3)$$

This doesn't work. The problem is not with its rate of convergence, which is of course quadratic for any finite  $y$  if we start close enough to the solution (see §9.4). The problem is that the region in which it converges at all is very small, especially as  $y \rightarrow 0$ . So, if we don't already have a good approximation as we approach the singularity, we are sunk.

If we change variables, we can get different (not computationally equivalent) versions of Newton's method. For example, let

$$u \equiv \log(x), \quad x = e^u \quad (6.11.4)$$

Newton's method in  $u$  looks like this:

$$\begin{aligned} f(u) &= ue^u - y \\ f'(u) &= (1 + u)e^u \\ u_{i+1} &= u_i - \frac{u_i - e^{-u_i} y}{1 + u_i} \end{aligned} \quad (6.11.5)$$

But it turns out that iteration (6.11.5) is no better than (6.11.3).

The observation that leads to a good solution is that, since its log term varies only slowly,  $y = x \log(x)$  is only very modestly curved *when it is plotted in log-log coordinates*. (Actually it is the negative of  $y$  that is plotted, since log-log coordinates require positive quantities.) Algebraically, we rewrite equation (6.11.1) as

$$(-y) = (-u)e^u \quad (6.11.6)$$

(with  $u$  as defined above) and take logarithms, giving

$$\log(-y) = u + \log(-u) \quad (6.11.7)$$

This leads to the Newton formulas,

$$\begin{aligned} f(u) &= u + \log(-u) - \log(-y) \\ f'(u) &= \frac{u + 1}{u} \\ u_{i+1} &= u_i + \frac{u_i}{u_i + 1} \left[ \log\left(\frac{y}{u_i}\right) - u_i \right] \end{aligned} \quad (6.11.8)$$

It turns out that the iteration (6.11.8) converges quadratically over quite a broad region of initial guesses. For  $-0.2 < y < 0$ , you can just choose  $-10$  (for example) as a fixed initial guess. When  $-0.2 < y < -e^{-1}$ , one can use the Taylor series expansion around  $x = e^{-1}$ ,

$$y(x - e^{-1}) = -e^{-1} + \frac{1}{2}e(x - e^{-1})^2 + \dots \quad (6.11.9)$$

which yields

$$x \approx e^{-1} - \sqrt{2e^{-1}(y + e^{-1})} \quad (6.11.10)$$

With these initial guesses, (6.11.8) never takes more than six iterations to converge to double precision accuracy, and there is just one log and a few arithmetic operations per iteration. The implementation looks like this:

```
Doub invxlogx(Doub y) {
```

For negative  $y$ ,  $0 > y > -e^{-1}$ , return  $x$  such that  $y = x \log(x)$ . The value returned is always the smaller of the two roots and is in the range  $0 < x < e^{-1}$ .

```
    const Doub ooe = 0.367879441171442322;
    Doub t,u,to=0.;
    if (y >= 0. || y <= -ooe) throw("no such inverse value");
    if (y < -0.2) u = log(ooe+sqrt(2*ooe*(y+ooe))); First approximation by inverse
    else u = -10.;                                     of Taylor series.
    do {                                              See text for derivation.
        u += (t=(log(y/u)-u)*(u/(1.+u)));
        if (t < 1.e-8 && abs(t+to)<0.01*abs(t)) break;
        to = t;
    } while (abs(t/u) > 1.e-15);
    return exp(u);
}
```

ksdist.h

## 6.12 Elliptic Integrals and Jacobian Elliptic Functions

Elliptic integrals occur in many applications, because any integral of the form

$$\int R(t, s) dt \quad (6.12.1)$$

where  $R$  is a rational function of  $t$  and  $s$ , and  $s$  is the square root of a cubic or quartic polynomial in  $t$ , can be evaluated in terms of elliptic integrals. Standard references [1] describe how to carry out the reduction, which was originally done by Legendre. Legendre showed that only three basic elliptic integrals are required. The simplest of these is

$$I_1 = \int_y^x \frac{dt}{\sqrt{(a_1 + b_1 t)(a_2 + b_2 t)(a_3 + b_3 t)(a_4 + b_4 t)}} \quad (6.12.2)$$

where we have written the quartic  $s^2$  in factored form. In standard integral tables [2], one of the limits of integration is always a zero of the quartic, while the other limit lies closer than the next zero, so that there is no singularity within the interval. To evaluate  $I_1$ , we simply break the interval  $[y, x]$  into subintervals, each of which either begins or ends on a singularity. The tables, therefore, need only distinguish the eight cases in which each of the four zeros (ordered according to size) appears as the upper or lower limit of integration. In addition, when one of the  $b$ 's in (6.12.2) tends to zero, the quartic reduces to a cubic, with the largest or smallest singularity moving to  $\pm\infty$ ; this leads to eight more cases (actually just special cases of the first eight). The 16 cases in total are then usually tabulated in terms of Legendre's standard elliptic integral of the first kind, which we will define below. By a change of the variable of integration  $t$ , the zeros of the quartic are mapped to standard locations on the real axis. Then only two dimensionless parameters are needed to tabulate Legendre's integral. However, the symmetry of the original integral (6.12.2) under permutation of the roots is concealed in Legendre's notation. We will get back to Legendre's notation below. But first, here is a better approach:

Carlson [3] has given a new definition of a standard elliptic integral of the first kind,

$$R_F(x, y, z) = \frac{1}{2} \int_0^\infty \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}} \quad (6.12.3)$$

where  $x$ ,  $y$ , and  $z$  are nonnegative and at most one is zero. By standardizing the range of integration, he retains permutation symmetry for the zeros. (Weierstrass' canonical form also has this property.) Carlson first shows that when  $x$  or  $y$  is a zero of the quartic in (6.12.2), the integral  $I_1$  can be written in terms of  $R_F$  in a form that is symmetric under permutation of the remaining three zeros. In the general case, when neither  $x$  nor  $y$  is a zero, two such  $R_F$  functions can be combined into a single one by an *addition theorem*, leading to the fundamental formula

$$I_1 = 2R_F(U_{12}^2, U_{13}^2, U_{14}^2) \quad (6.12.4)$$

where

$$U_{ij} = (X_i X_j Y_k Y_m + Y_i Y_j X_k X_m)/(x - y) \quad (6.12.5)$$

$$X_i = (a_i + b_i x)^{1/2}, \quad Y_i = (a_i + b_i y)^{1/2} \quad (6.12.6)$$

and  $i, j, k, m$  is any permutation of 1, 2, 3, 4. A short-cut in evaluating these expressions is

$$\begin{aligned} U_{13}^2 &= U_{12}^2 - (a_1 b_4 - a_4 b_1)(a_2 b_3 - a_3 b_2) \\ U_{14}^2 &= U_{12}^2 - (a_1 b_3 - a_3 b_1)(a_2 b_4 - a_4 b_2) \end{aligned} \quad (6.12.7)$$

The  $U$ 's correspond to the three ways of pairing the four zeros, and  $I_1$  is thus manifestly symmetric under permutation of the zeros. Equation (6.12.4) therefore reproduces all 16 cases when one limit is a zero, and also includes the cases when neither limit is a zero.

Thus Carlson's function allows arbitrary ranges of integration and arbitrary positions of the branch points of the integrand relative to the interval of integration. To handle elliptic integrals of the second and third kinds, Carlson defines the standard integral of the third kind as

$$R_J(x, y, z, p) = \frac{3}{2} \int_0^\infty \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}} \quad (6.12.8)$$

which is symmetric in  $x$ ,  $y$ , and  $z$ . The degenerate case when two arguments are equal is denoted

$$R_D(x, y, z) = R_J(x, y, z, z) \quad (6.12.9)$$

and is symmetric in  $x$  and  $y$ . The function  $R_D$  replaces Legendre's integral of the second kind. The degenerate form of  $R_F$  is denoted

$$R_C(x, y) = R_F(x, y, y) \quad (6.12.10)$$

It embraces logarithmic, inverse circular, and inverse hyperbolic functions.

Carlson [4-7] gives integral tables in terms of the exponents of the linear factors of the quartic in (6.12.1). For example, the integral where the exponents are  $(\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{3}{2})$  can be expressed as a single integral in terms of  $R_D$ ; it accounts for 144 separate cases in Gradshteyn and Ryzhik [2].

Refer to Carlson's papers [3-8] for some of the practical details in reducing elliptic integrals to his standard forms, such as handling complex-conjugate zeros.

Turn now to the numerical evaluation of elliptic integrals. The traditional methods [9] are Gauss or Landen transformations. *Descending* transformations decrease the modulus  $k$  of the Legendre integrals toward zero, and *increasing* transformations increase it toward unity. In these limits the functions have simple analytic expressions. While these methods converge quadratically and are quite satisfactory for integrals of the first and second kinds, they generally lead to loss of significant figures in certain regimes for integrals of the third kind. Carlson's algorithms [10,11], by contrast, provide a unified method for all three kinds with no significant cancellations.

The key ingredient in these algorithms is the *duplication theorem*:

$$\begin{aligned} R_F(x, y, z) &= 2R_F(x + \lambda, y + \lambda, z + \lambda) \\ &= R_F\left(\frac{x + \lambda}{4}, \frac{y + \lambda}{4}, \frac{z + \lambda}{4}\right) \end{aligned} \quad (6.12.11)$$

where

$$\lambda = (xy)^{1/2} + (xz)^{1/2} + (yz)^{1/2} \quad (6.12.12)$$

This theorem can be proved by a simple change of variable of integration [12]. Equation (6.12.11) is iterated until the arguments of  $R_F$  are nearly equal. For equal arguments we have

$$R_F(x, x, x) = x^{-1/2} \quad (6.12.13)$$

When the arguments are close enough, the function is evaluated from a fixed Taylor expansion about (6.12.13) through fifth-order terms. While the iterative part of the algorithm is only linearly convergent, the error ultimately decreases by a factor of  $4^6 = 4096$  for each iteration. Typically only two or three iterations are required, perhaps six or seven if the initial values of the arguments have huge ratios. We list the algorithm for  $R_F$  here, and refer you to Carlson's paper [10] for the other cases.

Stage 1: For  $n = 0, 1, 2, \dots$  compute

$$\begin{aligned} \mu_n &= (x_n + y_n + z_n)/3 \\ X_n &= 1 - (x_n/\mu_n), \quad Y_n = 1 - (y_n/\mu_n), \quad Z_n = 1 - (z_n/\mu_n) \\ \epsilon_n &= \max(|X_n|, |Y_n|, |Z_n|) \end{aligned}$$

If  $\epsilon_n < \text{tol}$ , go to Stage 2; else compute

$$\begin{aligned} \lambda_n &= (x_n y_n)^{1/2} + (x_n z_n)^{1/2} + (y_n z_n)^{1/2} \\ x_{n+1} &= (x_n + \lambda_n)/4, \quad y_{n+1} = (y_n + \lambda_n)/4, \quad z_{n+1} = (z_n + \lambda_n)/4 \end{aligned}$$

and repeat this stage.

Stage 2: Compute

$$\begin{aligned} E_2 &= X_n Y_n - Z_n^2, \quad E_3 = X_n Y_n Z_n \\ R_F &= (1 - \frac{1}{10}E_2 + \frac{1}{14}E_3 + \frac{1}{24}E_2^2 - \frac{3}{44}E_2 E_3)/(\mu_n)^{1/2} \end{aligned}$$

In some applications the argument  $p$  in  $R_J$  or the argument  $y$  in  $R_C$  is negative, and the Cauchy principal value of the integral is required. This is easily handled by using the formulas

$$\begin{aligned} R_J(x, y, z, p) &= \\ &[ (\gamma - y)R_J(x, y, z, \gamma) - 3R_F(x, y, z) + 3R_C(xz/y, p\gamma/y)] / (y - p) \end{aligned} \quad (6.12.14)$$

where

$$\gamma \equiv y + \frac{(z - y)(y - x)}{y - p} \quad (6.12.15)$$

is positive if  $p$  is negative, and

$$R_C(x, y) = \left( \frac{x}{x - y} \right)^{1/2} R_C(x - y, -y) \quad (6.12.16)$$

The Cauchy principal value of  $R_J$  has a zero at some value of  $p < 0$ , so (6.12.14) will give some loss of significant figures near the zero.

**elliptint.h**

```
Doub rf(const Doub x, const Doub y, const Doub z) {
    Computes Carlson's elliptic integral of the first kind,  $R_F(x, y, z)$ .  $x$ ,  $y$ , and  $z$  must be non-negative, and at most one can be zero.
    static const Doub ERRTOL=0.0025, THIRD=1.0/3.0,C1=1.0/24.0, C2=0.1,
    C3=3.0/44.0, C4=1.0/14.0;
    static const Doub TINY=5.0*numeric_limits<Doub>::min(),
    BIG=0.2*numeric_limits<Doub>::max();
    Doub alamb,ave,delx,dely,delz,e2,e3,sqrtx,sqrty,sqrtz,xt,yt,zt;
    if (MIN(MIN(x,y),z) < 0.0 || MIN(MIN(x+y,x+z),y+z) < TINY || 
        MAX(MAX(x,y),z) > BIG) throw("invalid arguments in rf");
    xt=x;
    yt=y;
    zt=z;
    do {
        sqrtx=sqrt(xt);
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        ave=THIRD*(xt+yt+zt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
    } while (MAX(MAX(abs(delx),abs(dely)),abs(delz)) > ERRTOL);
    e2=delx*dely-delz*delz;
    e3=delx*dely*delz;
    return (1.0+(C1*e2-C2-C3*e3)*e2+C4*e3)/sqrt(ave);
}
```

A value of 0.0025 for the error tolerance parameter gives full double precision (16 significant digits). Since the error scales as  $\epsilon_n^6$ , we see that 0.08 would be adequate for single precision (7 significant digits), but would save at most two or three more iterations. Since the coefficients of the sixth-order truncation error are different for the other elliptic functions, these values for the error tolerance should be set to 0.04 (single precision) or 0.0012 (double precision) in the algorithm for  $R_C$ , and 0.05 or 0.0015 for  $R_J$  and  $R_D$ . As well as being an algorithm in its own right for certain combinations of elementary functions, the algorithm for  $R_C$  is used repeatedly in the computation of  $R_J$ .

The C++ implementations test the input arguments against two machine-dependent constants, TINY and BIG, to ensure that there will be no underflow or overflow during the computation. You can always extend the range of admissible argument values by using the homogeneity relations (6.12.22), below.

**elliptint.h**

```
Doub rd(const Doub x, const Doub y, const Doub z) {
    Computes Carlson's elliptic integral of the second kind,  $R_D(x, y, z)$ .  $x$  and  $y$  must be nonnegative, and at most one can be zero.  $z$  must be positive.
    static const Doub ERRTOL=0.0015, C1=3.0/14.0, C2=1.0/6.0, C3=9.0/22.0,
    C4=3.0/26.0, C5=0.25*C3, C6=1.5*C4;
    static const Doub TINY=2.0*pow(numeric_limits<Doub>::max(),-2./3.),
    BIG=0.1*ERRTOL*pow(numeric_limits<Doub>::min(),-2./3.);
    Doub alamb,ave,delx,dely,delz,ea,eb,ec,ed,ee,fac,sqrtx,sqrty,
    sqrtz,sum,xt,yt,zt;
    if (MIN(x,y) < 0.0 || MIN(x+y,z) < TINY || MAX(MAX(x,y),z) > BIG)
        throw("invalid arguments in rd");
    xt=x;
    yt=y;
    zt=z;
    sum=0.0;
    fac=1.0;
    do {
        sqrtx=sqrt(xt);
```

```

        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrty*(sqrtz+sqrtz)+sqrty*sqrtz;
        sum += fac/(sqrtz*(zt+alamb));
        fac=0.25*fac;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        ave=0.2*(xt+yt+3.0*zt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
    } while (MAX(MAX(abs(delx),abs(dely)),abs(delz)) > ERRTOL);
    ea=delx*dely;
    eb=delz*delz;
    ec=ea-eb;
    ed=ea-6.0*eb;
    ee=ed+ec+ec;
    return 3.0*sum+fac*(1.0+ed*(-C1+C5*ed-C6*delz*ee)
        +delz*(C2*ee+delz*(-C3*ec+delz*C4*ea)))/(ave*sqrt(ave));
}

```

**Doub rj(const Doub x, const Doub y, const Doub z, const Doub p) {**

Computes Carlson's elliptic integral of the third kind,  $R_J(x, y, z, p)$ .  $x$ ,  $y$ , and  $z$  must be nonnegative, and at most one can be zero.  $p$  must be nonzero. If  $p < 0$ , the Cauchy principal value is returned.

```

static const Doub ERRTOL=0.0015, C1=3.0/14.0, C2=1.0/3.0, C3=3.0/22.0,
    C4=3.0/26.0, C5=0.75*C3, C6=1.5*C4, C7=0.5*C2, C8=C3+C3;
static const Doub TINY=pow(5.0*numeric_limits<Doub>::min(),1./3.),
    BIG=0.3*pow(0.2*numeric_limits<Doub>::max(),1./3.);
Doub a,alamb,alpha,ans,ave,b,beta,delp,delx,dely,delz,ea,eb,ec,ed,ee,
    fac,pt,rcx,rho,sqrty,sqrty,sqrty,sum,tau,xt,yt,zt;
if (MIN(MIN(x,y),z) < 0.0 || MIN(MIN(x+y,x+z),MIN(y+z,abs(p))) < TINY
    || MAX(MAX(x,y),MAX(z,abs(p))) > BIG) throw("invalid arguments in rj");
sum=0.0;
fac=1.0;
if (p > 0.0) {
    xt=x;
    yt=y;
    zt=z;
    pt=p;
} else {
    xt=MIN(MIN(x,y),z);
    zt=MAX(MAX(x,y),z);
    yt=x+y+z-xt-zt;
    a=1.0/(yt-p);
    b=a*(zt-yt)*(yt-xt);
    pt=yt+b;
    rho=xt*zt/yt;
    tau=p*pt/yt;
    rcx=rc(rho,tau);
}
do {
    sqrty=sqrt(xt);
    sqrtz=sqrt(yt);
    sqrtz=sqrt(zt);
    alamb=sqrty*(sqrtz+sqrtz)+sqrty*sqrtz;
    alpha=SQR(pt*(sqrty+sqrtz)+sqrtx*sqrty*sqrtz);
    beta=pt*SQR(pt+alamb);
    sum += fac*rc(alpha,beta);
    fac=0.25*fac;
    xt=0.25*(xt+alamb);
    yt=0.25*(yt+alamb);
}

```

elliptint.h

```

zt=0.25*(zt+alamb);
pt=0.25*(pt+alamb);
ave=0.2*(xt+yt+zt+pt+pt);
delx=(ave-xt)/ave;
dely=(ave-yt)/ave;
delz=(ave-zt)/ave;
delp=(ave-pt)/ave;
} while (MAX(MAX(abs(delx),abs(dely)),
    MAX(abs(delz),abs(delp))) > ERRTOL);
ea=delx*(dely+delz)+dely*delz;
eb=delx*dely*delz;
ec=delp*delp;
ed=ea-3.0*ec;
ee=eb+2.0*delp*(ea-ec);
ans=3.0*sum+fac*(1.0+ed*(-C1+C5*ed-C6*ee)+eb*(C7+delp*(-C8+delp*C4))
    +delp*ea*(C2-delp*C3)-C2*delp*ec)/(ave*sqrt(ave));
if (p <= 0.0) ans=a*(b*ans+3.0*(rcx-rf(xt,yt,zt)));
return ans;
}

```

**elliptint.h**

```
Doub rc(const Doub x, const Doub y) {
```

Computes Carlson's degenerate elliptic integral,  $R_C(x, y)$ .  $x$  must be nonnegative and  $y$  must be nonzero. If  $y < 0$ , the Cauchy principal value is returned.

```

static const Doub ERRTOL=0.0012, THIRD=1.0/3.0, C1=0.3, C2=1.0/7.0,
    C3=0.375, C4=9.0/22.0;
static const Doub TINY=5.0*numeric_limits<Doub>::min(),
    BIG=0.2*numeric_limits<Doub>::max(), COMP1=2.236/sqrt(TINY),
    COMP2=SQR(TINY*BIG)/25.0;
Doub alamb,ave,s,w,xt,yt;
if (x < 0.0 || y == 0.0 || (x+abs(y)) < TINY || (x+abs(y)) > BIG ||
    (y<-COMP1 && x > 0.0 && x < COMP2)) throw("invalid arguments in rc");
if (y > 0.0) {
    xt=x;
    yt=y;
    w=1.0;
} else {
    xt=x-y;
    yt= -y;
    w=sqrt(x)/sqrt(xt);
}
do {
    alamb=2.0*sqrt(xt)*sqrt(yt)+yt;
    xt=0.25*(xt+alamb);
    yt=0.25*(yt+alamb);
    ave=THIRD*(xt+yt+yt);
    s=(yt-ave)/ave;
} while (abs(s) > ERRTOL);
return w*(1.0+s*s*(C1+s*(C2+s*(C3+s*C4))))/sqrt(ave);
}
```

At times you may want to express your answer in Legendre's notation. Alternatively, you may be given results in that notation and need to compute their values with the programs given above. It is a simple matter to transform back and forth. The *Legendre elliptic integral of the first kind* is defined as

$$F(\phi, k) \equiv \int_0^\phi \frac{d\theta}{\sqrt{1-k^2 \sin^2 \theta}} \quad (6.12.17)$$

The *complete elliptic integral of the first kind* is given by

$$K(k) \equiv F(\pi/2, k) \quad (6.12.18)$$

In terms of  $R_F$ ,

$$\begin{aligned} F(\phi, k) &= \sin \phi R_F(\cos^2 \phi, 1 - k^2 \sin^2 \phi, 1) \\ K(k) &= R_F(0, 1 - k^2, 1) \end{aligned} \quad (6.12.19)$$

The *Legendre elliptic integral of the second kind* and the *complete elliptic integral of the second kind* are given by

$$\begin{aligned} E(\phi, k) &\equiv \int_0^\phi \sqrt{1 - k^2 \sin^2 \theta} d\theta \\ &= \sin \phi R_F(\cos^2 \phi, 1 - k^2 \sin^2 \phi, 1) \\ &\quad - \frac{1}{3} k^2 \sin^3 \phi R_D(\cos^2 \phi, 1 - k^2 \sin^2 \phi, 1) \\ E(k) &\equiv E(\pi/2, k) = R_F(0, 1 - k^2, 1) - \frac{1}{3} k^2 R_D(0, 1 - k^2, 1) \end{aligned} \quad (6.12.20)$$

Finally, the *Legendre elliptic integral of the third kind* is

$$\begin{aligned} \Pi(\phi, n, k) &\equiv \int_0^\phi \frac{d\theta}{(1 + n \sin^2 \theta) \sqrt{1 - k^2 \sin^2 \theta}} \\ &= \sin \phi R_F(\cos^2 \phi, 1 - k^2 \sin^2 \phi, 1) \\ &\quad - \frac{1}{3} n \sin^3 \phi R_J(\cos^2 \phi, 1 - k^2 \sin^2 \phi, 1, 1 + n \sin^2 \phi) \end{aligned} \quad (6.12.21)$$

(Note that this sign convention for  $n$  is opposite that of Abramowitz and Stegun [13], and that their  $\sin \alpha$  is our  $k$ .)

```
Doub ellf(const Doub phi, const Doub ak) {
    Legendre elliptic integral of the first kind  $F(\phi, k)$ , evaluated using Carlson's function  $R_F$ . The argument ranges are  $0 \leq \phi \leq \pi/2$ ,  $0 \leq k \sin \phi \leq 1$ .
    Doub s=sin(phi);
    return s*rf(SQR(cos(phi)),(1.0-s*ak)*(1.0+s*ak),1.0);
}
```

elliptint.h

```
Doub elle(const Doub phi, const Doub ak) {
    Legendre elliptic integral of the second kind  $E(\phi, k)$ , evaluated using Carlson's functions  $R_D$  and  $R_F$ . The argument ranges are  $0 \leq \phi \leq \pi/2$ ,  $0 \leq k \sin \phi \leq 1$ .
    Doub cc,q,s;
    s=sin(phi);
    cc=SQR(cos(phi));
    q=(1.0-s*ak)*(1.0+s*ak);
    return s*(rf(cc,q,1.0)-(SQR(s*ak))*rd(cc,q,1.0)/3.0);
}
```

elliptint.h

```
Doub ellpi(const Doub phi, const Doub en, const Doub ak) {
    Legendre elliptic integral of the third kind  $\Pi(\phi, n, k)$ , evaluated using Carlson's functions  $R_J$  and  $R_F$ . (Note that the sign convention on  $n$  is opposite that of Abramowitz and Stegun.) The ranges of  $\phi$  and  $k$  are  $0 \leq \phi \leq \pi/2$ ,  $0 \leq k \sin \phi \leq 1$ .
    Doub cc,enss,q,s;
    s=sin(phi);
    enss=en*s*s;
    cc=SQR(cos(phi));
    q=(1.0-s*ak)*(1.0+s*ak);
    return s*(rf(cc,q,1.0)-enss*rj(cc,q,1.0,1.0+enss)/3.0);
}
```

elliptint.h

Carlson's functions are homogeneous of degree  $-\frac{1}{2}$  and  $-\frac{3}{2}$ , so

$$\begin{aligned} R_F(\lambda x, \lambda y, \lambda z) &= \lambda^{-1/2} R_F(x, y, z) \\ R_J(\lambda x, \lambda y, \lambda z, \lambda p) &= \lambda^{-3/2} R_J(x, y, z, p) \end{aligned} \quad (6.12.22)$$

Thus, to express a Carlson function in Legendre's notation, permute the first three arguments into ascending order, use homogeneity to scale the third argument to be 1, and then use equations (6.12.19) – (6.12.21).

### 6.12.1 Jacobian Elliptic Functions

The Jacobian elliptic function  $\text{sn}$  is defined as follows: Instead of considering the elliptic integral

$$u(y, k) \equiv u = F(\phi, k) \quad (6.12.23)$$

consider the *inverse* function

$$y = \sin \phi = \text{sn}(u, k) \quad (6.12.24)$$

Equivalently,

$$u = \int_0^{\text{sn}} \frac{dy}{\sqrt{(1 - y^2)(1 - k^2 y^2)}} \quad (6.12.25)$$

When  $k = 0$ ,  $\text{sn}$  is just  $\sin$ . The functions  $\text{cn}$  and  $\text{dn}$  are defined by the relations

$$\text{sn}^2 + \text{cn}^2 = 1, \quad k^2 \text{sn}^2 + \text{dn}^2 = 1 \quad (6.12.26)$$

The routine given below actually takes  $m_c \equiv k_c^2 = 1 - k^2$  as an input parameter. It also computes all three functions  $\text{sn}$ ,  $\text{cn}$ , and  $\text{dn}$  since computing all three is no harder than computing any one of them. For a description of the method, see [9].

```
elliptint.h
void sncndn(const Doub uu, const Doub emmc, Doub &sn, Doub &cn, Doub &dn) {
    Returns the Jacobian elliptic functions sn(u, k_c), cn(u, k_c), and dn(u, k_c). Here uu = u, while
    emmc = k_c^2.
    static const Doub CA=1.0e-8;           The accuracy is the square of CA.
    Bool bo;
    Int i,ii,l;
    Doub a,b,c,d,emc,u;
    VecDoub em(13),en(13);
    emc=emmc;
    u=uu;
    if (emc != 0.0) {
        bo=(emc < 0.0);
        if (bo) {
            d=1.0-emc;
            emc /= -1.0/d;
            u *= (d=sqrt(d));
        }
        a=1.0;
        dn=1.0;
        for (i=0;i<13;i++) {
            l=i;
            em[i]=a;
            en[i]=(emc=sqrt(emc));
            c=0.5*(a+emc);
            if (abs(a-emc) <= CA*a) break;
            emc *= a;
        }
    }
}
```

```

        a=c;
    }
    u *= c;
    sn=sin(u);
    cn=cos(u);
    if (sn != 0.0) {
        a=cn/sn;
        c *= a;
        for (ii=1;ii>=0;ii--) {
            b=em[ii];
            a *= c;
            c *= dn;
            dn=(en[ii]+a)/(b+a);
            a=c/b;
        }
        a=1.0/sqrt(c*c+1.0);
        sn=(sn >= 0.0 ? a : -a);
        cn=c*sn;
    }
    if (bo) {
        a=dn;
        dn=cn;
        cn=a;
        sn /= d;
    }
} else {
    cn=1.0/cosh(u);
    dn=cn;
    sn=tanh(u);
}
}

```

#### CITED REFERENCES AND FURTHER READING:

- Erdélyi, A., Magnus, W., Oberhettinger, F., and Tricomi, F.G. 1953, *Higher Transcendental Functions*, Vol. II, (New York: McGraw-Hill).[1]
- Gradshteyn, I.S., and Ryzhik, I.W. 1980, *Table of Integrals, Series, and Products* (New York: Academic Press).[2]
- Carlson, B.C. 1977, "Elliptic Integrals of the First Kind," *SIAM Journal on Mathematical Analysis*, vol. 8, pp. 231–242.[3]
- Carlson, B.C. 1987, "A Table of Elliptic Integrals of the Second Kind," *Mathematics of Computation*, vol. 49, pp. 595–606[4]; 1988, "A Table of Elliptic Integrals of the Third Kind," *op. cit.*, vol. 51, pp. 267–280[5]; 1989, "A Table of Elliptic Integrals: Cubic Cases," *op. cit.*, vol. 53, pp. 327–333[6]; 1991, "A Table of Elliptic Integrals: One Quadratic Factor," *op. cit.*, vol. 56, pp. 267–280.[7]
- Carlson, B.C., and FitzSimons, J. 2000, "Reduction Theorems for Elliptic Integrands with the Square Root of Two Quadratic Factors," *Journal of Computational and Applied Mathematics*, vol. 118, pp. 71–85.[8]
- Bulirsch, R. 1965, "Numerical Calculation of Elliptic Integrals and Elliptic Functions," *Numerische Mathematik*, vol. 7, pp. 78–90; 1965, *op. cit.*, vol. 7, pp. 353–354; 1969, *op. cit.*, vol. 13, pp. 305–315.[9]
- Carlson, B.C. 1979, "Computing Elliptic Integrals by Duplication," *Numerische Mathematik*, vol. 33, pp. 1–16.[10]
- Carlson, B.C., and Notis, E.M. 1981, "Algorithms for Incomplete Elliptic Integrals," *ACM Transactions on Mathematical Software*, vol. 7, pp. 398–403.[11]
- Carlson, B.C. 1978, "Short Proofs of Three Theorems on Elliptic Integrals," *SIAM Journal on Mathematical Analysis*, vol. 9, p. 524–528.[12]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>, Chapter 17.[13]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 78–79.

## 6.13 Hypergeometric Functions

As was discussed in §5.14, a fast, general routine for the the complex hypergeometric function  ${}_2F_1(a, b, c; z)$  is difficult or impossible. The function is defined as the analytic continuation of the hypergeometric series

$$\begin{aligned} {}_2F_1(a, b, c; z) = & 1 + \frac{ab}{c} \frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)} \frac{z^2}{2!} + \dots \\ & + \frac{a(a+1)\dots(a+j-1)b(b+1)\dots(b+j-1)}{c(c+1)\dots(c+j-1)} \frac{z^j}{j!} + \dots \end{aligned} \quad (6.13.1)$$

This series converges only within the unit circle  $|z| < 1$  (see [1]), but one's interest in the function is not confined to this region.

Section 5.14 discussed the method of evaluating this function by direct path integration in the complex plane. We here merely list the routines that result.

Implementation of the function hypgeo is straightforward and is described by comments in the program. The machinery associated with Chapter 17's routine for integrating differential equations, Odeint, is only minimally intrusive and need not even be completely understood: Use of Odeint requires one function call to the constructor, with a prescribed format for the derivative routine Hypderiv, followed by a call to the integrate method.

The function hypgeo will fail, of course, for values of  $z$  too close to the singularity at 1. (If you need to approach this singularity, or the one at  $\infty$ , use the “linear transformation formulas” in §15.3 of [1].) Away from  $z = 1$ , and for moderate values of  $a, b, c$ , it is often remarkable how few steps are required to integrate the equations. A half-dozen is typical.

```
hypgeo.h Complex hypgeo(const Complex &a, const Complex &b,const Complex &c,
                      const Complex &z)
```

Complex hypergeometric function  ${}_2F_1$  for complex  $a, b, c$ , and  $z$ , by direct integration of the hypergeometric equation in the complex plane. The branch cut is taken to lie along the real axis,  $\text{Re } z > 1$ .

```
{
    const Doub atol=1.0e-14,rtol=1.0e-14;      Accuracy parameters.
    Complex ans,dz,z0,y[2];
    VecDoub yy(4);
    if (norm(z) <= 0.25) {                      Use series...
        hypers(a,b,c,z,ans,y[1]);
        return ans;
    }
    ...or pick a starting point for the path integration.
    else if (real(z) < 0.0) z0=Complex(-0.5,0.0);
    else if (real(z) <= 1.0) z0=Complex(0.5,0.0);
```

```

else z0=Complex(0.0,imag(z) >= 0.0 ? 0.5 : -0.5);
dz=z-z0;
hypser(a,b,c,z0,y[0],y[1]);           Get starting function and derivative.
yy[0]=real(y[0]);
yy[1]=imag(y[0]);
yy[2]=real(y[1]);
yy[3]=imag(y[1]);
Hypderiv d(a,b,c,z0,dz);           Set up the functor for the derivatives.
Output out;                         Suppress output in Odeint.
Odeint<StepperBS<Hypderiv> > ode(yy,0.0,1.0,atol,rtol,0.1,0.0,out,d);
The arguments to Odeint are the vector of independent variables, the starting and ending
values of the dependent variable, the accuracy parameters, an initial guess for the stepsize,
a minimum stepsize, and the names of the output object and the derivative object. The
integration is performed by the Bulirsch-Stoer stepping routine.
ode.integrate();
y[0]=Complex(yy[0],yy[1]);
return y[0];
}

```

```

void hypser(const Complex &a, const Complex &b, const Complex &c,           hypgeo.h
           const Complex &z, Complex &series, Complex &deriv)
Returns the hypergeometric series  ${}_2F_1$  and its derivative, iterating to machine accuracy. For
 $|z| \leq 1/2$  convergence is quite rapid.
{
    deriv=0.0;
    Complex fac=1.0;
    Complex temp=fac;
    Complex aa=a;
    Complex bb=b;
    Complex cc=c;
    for (Int n=1;n<=1000;n++) {
        fac *= ((aa*bb)/cc);
        deriv += fac;
        fac *= ((1.0/n)*z);
        series=temp+fac;
        if (series == temp) return;
        temp=series;
        aa += 1.0;
        bb += 1.0;
        cc += 1.0;
    }
    throw("convergence failure in hypser");
}

```

```

struct Hypderiv {                           hypgeo.h
Functor to compute derivatives for the hypergeometric equation; see text equation (5.14.4).
    Complex a,b,c,z0,dz;
    Hypderiv(const Complex &aa, const Complex &bb,
              const Complex &cc, const Complex &z00,
              const Complex &dzz) : a(aa),b(bb),c(cc),z0(z00),dz(dzz) {}
    void operator() (const Doub s, VecDoub_I &yy, VecDoub_0 &ddyds) {
        Complex z,y[2],dyds[2];
        y[0]=Complex(yy[0],yy[1]);
        y[1]=Complex(yy[2],yy[3]);
        z=z0+s*dz;
        dyds[0]=y[1]*dz;
        dyds[1]=(a*b*y[0]-(c-(a+b+1.0)*z)*y[1])*dz/(z*(1.0-z));
        ddyds[0]=real(dyds[0]);
        ddyds[1]=imag(dyds[0]);
        ddyds[2]=real(dyds[1]);
        ddyds[3]=imag(dyds[1]);
    }
};

```

**CITED REFERENCES AND FURTHER READING:**

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at <http://www.nr.com/aands>.[1]

## 6.14 Statistical Functions

Certain special functions get frequent use because of their relation to common univariate statistical distributions, that is, probability densities in a single variable. In this section we survey a number of such common distributions in a unified way, giving, in each case, routines for computing the probability density function  $p(x)$ ; the cumulative density function or *cdf*, written  $P(< x)$ ; and the inverse of the cumulative density function  $x(P)$ . The latter function is needed for finding the values of  $x$  associated with specified *percentile points* or *quantiles* in significance tests, for example, the 0.5%, 5%, 95% or 99.5% points.

The emphasis of this section is on defining and computing these statistical functions. Section §7.3 is a related section that discusses how to generate random deviates from the distributions discussed here. We defer discussion of the actual use of these distributions in statistical tests to Chapter 14.

### 6.14.1 Normal (Gaussian) Distribution

If  $x$  is drawn from a *normal distribution* with mean  $\mu$  and standard deviation  $\sigma$ , then we write

$$\begin{aligned} x &\sim N(\mu, \sigma), \quad \sigma > 0 \\ p(x) &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\left[\frac{x-\mu}{\sigma}\right]^2\right) \end{aligned} \tag{6.14.1}$$

with  $p(x)$  the probability density function. Note the special use of the notation “ $\sim$ ” in this section, which can be read as “is drawn from a distribution.” The variance of the distribution is, of course,  $\sigma^2$ .

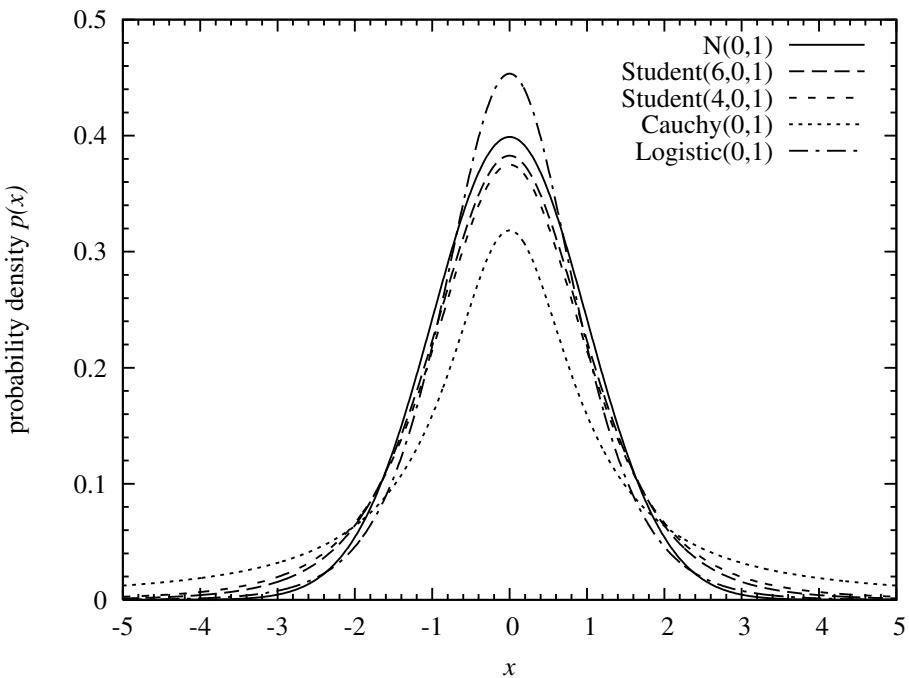
The cumulative distribution function is the probability of a value  $\leq x$ . For the normal distribution, this is given in terms of the complementary error function by

$$\text{cdf} \equiv P(< x) \equiv \int_{-\infty}^x p(x')dx' = \frac{1}{2}\text{erfc}\left(-\frac{1}{\sqrt{2}}\left[\frac{x-\mu}{\sigma}\right]\right) \tag{6.14.2}$$

The inverse cdf can thus be calculated in terms of the inverse erfc,

$$x(P) = \mu - \sqrt{2}\sigma\text{erfc}^{-1}(2P) \tag{6.14.3}$$

The following structure implements the above relations.



**Figure 6.14.1.** Examples of centrally peaked distributions that are symmetric on the real line. Any of these can substitute for the normal distribution either as an approximation or in applications such as robust estimation. They differ largely in the decay rate of their tails.

```

struct Normaldist : Erf {
    Normal distribution, derived from the error function Erf.
    Doub mu, sig;
    Normaldist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {
        Constructor. Initialize with  $\mu$  and  $\sigma$ . The default with no arguments is  $N(0, 1)$ .
        if (sig <= 0.) throw("bad sig in Normaldist");
    }
    Doub p(Doub x) {
        Return probability density function.
        return (0.398942280401432678/sig)*exp(-0.5*SQR((x-mu)/sig));
    }
    Doub cdf(Doub x) {
        Return cumulative distribution function.
        return 0.5*erfc(-0.707106781186547524*(x-mu)/sig);
    }
    Doub invcdf(Doub p) {
        Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Normaldist");
        return -1.41421356237309505*sig*inverfc(2.*p)+mu;
    }
};
```

erf.h

We will use the conventions of the above code for all the distributions in this section. A distribution's parameters (here,  $\mu$  and  $\sigma$ ) are set by the constructor and then referenced as needed by the member functions. The density function is always  $p()$ , the cdf is  $cdf()$ , and the inverse cdf is  $invcdf()$ . We will generally check the arguments of probability functions for validity, since many program bugs can show up as, e.g., a probability out of the range  $[0, 1]$ .

## 6.14.2 Cauchy Distribution

Like the normal distribution, the *Cauchy distribution* is a centrally peaked, symmetric distribution with a parameter  $\mu$  that specifies its center and a parameter  $\sigma$  that specifies its width. *Unlike* the normal distribution, the Cauchy distribution has tails that decay very slowly at infinity, as  $|x|^{-2}$ , so slowly that moments higher than the zeroth moment (the area under the curve) don't even exist. The parameter  $\mu$  is therefore, strictly speaking, not the mean, and the parameter  $\sigma$  is not, technically, the standard deviation. But these two parameters substitute for those moments as measures of central position and width.

The defining probability density is

$$x \sim \text{Cauchy}(\mu, \sigma), \quad \sigma > 0$$

$$p(x) = \frac{1}{\pi\sigma} \left( 1 + \left[ \frac{x - \mu}{\sigma} \right]^2 \right)^{-1} \quad (6.14.4)$$

If  $x \sim \text{Cauchy}(0, 1)$ , then also  $1/x \sim \text{Cauchy}(0, 1)$  and also  $(ax + b)^{-1} \sim \text{Cauchy}(-b/a, 1/a)$ .

The cdf is given by

$$\text{cdf} \equiv P(< x) \equiv \int_{-\infty}^x p(x') dx' = \frac{1}{2} + \frac{1}{\pi} \arctan \left( \frac{x - \mu}{\sigma} \right) \quad (6.14.5)$$

The inverse cdf is given by

$$x(P) = \mu + \sigma \tan \left( \pi [P - \frac{1}{2}] \right) \quad (6.14.6)$$

Figure 6.14.1 shows Cauchy(0, 1) as compared to the normal distribution N(0, 1), as well as several other similarly shaped distributions discussed below.

The Cauchy distribution is sometimes called the *Lorentzian distribution*.

distributions.h

```
struct Cauchydist {
    Cauchy distribution.
    Doub mu, sig;
    Cauchydist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {
        Constructor. Initialize with  $\mu$  and  $\sigma$ . The default with no arguments is Cauchy(0, 1).
        if (sig <= 0.) throw("bad sig in Cauchydist");
    }
    Doub p(Doub x) {
        Return probability density function.
        return 0.318309886183790671/(sig*(1.+SQR((x-mu)/sig)));
    }
    Doub cdf(Doub x) {
        Return cumulative distribution function.
        return 0.5+0.318309886183790671*atan2(x-mu,sig);
    }
    Doub invcdf(Doub p) {
        Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Cauchydist");
        return mu + sig*tan(3.14159265358979324*(p-0.5));
    }
};
```

### 6.14.3 Student-t Distribution

A generalization of the Cauchy distribution is the Student-t distribution, named for the early 20th century statistician William Gosset, who published under the name “Student” because his employer, Guinness Breweries, required him to use a pseudonym. Like the Cauchy distribution, the Student-t distribution has power-law tails at infinity, but it has an additional parameter  $\nu$  that specifies how rapidly they decay, namely as  $|t|^{-(\nu+1)}$ . When  $\nu$  is an integer, the number of convergent moments, including the zeroth, is thus  $\nu$ .

The defining probability density (conventionally written in a variable  $t$  instead of  $x$ ) is

$$t \sim \text{Student}(\nu, \mu, \sigma), \quad \nu > 0, \sigma > 0$$

$$p(t) = \frac{\Gamma(\frac{1}{2}[\nu + 1])}{\Gamma(\frac{1}{2}\nu)\sqrt{\nu\pi}\sigma} \left(1 + \frac{1}{\nu} \left[\frac{t - \mu}{\sigma}\right]^2\right)^{-\frac{1}{2}(\nu+1)} \quad (6.14.7)$$

The Cauchy distribution is obtained in the case  $\nu = 1$ . In the opposite limit,  $\nu \rightarrow \infty$ , the normal distribution is obtained. In pre-computer days, this was the basis of various approximation schemes for the normal distribution, now all generally irrelevant. Figure 6.14.1 shows examples of the Student-t distribution for  $\nu = 1$  (Cauchy),  $\nu = 4$ , and  $\nu = 6$ . The approach to the normal distribution is evident.

The mean of  $\text{Student}(\nu, \mu, \sigma)$  is (by symmetry)  $\mu$ . The variance is not  $\sigma^2$ , but rather

$$\text{Var}\{\text{Student}(\nu, \mu, \sigma)\} = \frac{\nu}{\nu - 2} \sigma^2 \quad (6.14.8)$$

For additional moments, and other properties, see [1].

The cdf is given by an incomplete beta function. If we let

$$x \equiv \frac{\nu}{\nu + \left(\frac{t-\mu}{\sigma}\right)^2} \quad (6.14.9)$$

then

$$\text{cdf} \equiv P(< t) \equiv \int_{-\infty}^t p(t') dt' = \begin{cases} \frac{1}{2} I_x(\frac{1}{2}\nu, \frac{1}{2}), & t \leq \mu \\ 1 - \frac{1}{2} I_x(\frac{1}{2}\nu, \frac{1}{2}), & t > \mu \end{cases} \quad (6.14.10)$$

The inverse cdf is given by an inverse incomplete beta function (see code below for the exact formulation).

In practice, the Student-t cdf is the above form is rarely used, since most statistical tests using Student-t are double-sided. Conventionally, the two-tailed function  $A(t|\nu)$  is defined (only) for the case  $\mu = 0$  and  $\sigma = 1$  by

$$A(t|\nu) \equiv \int_{-t}^{+t} p(t') dt' = 1 - I_x(\frac{1}{2}\nu, \frac{1}{2}) \quad (6.14.11)$$

with  $x$  as given above. The statistic  $A(t|\nu)$  is notably used in the test of whether two observed distributions have the same mean. The code below implements both equations (6.14.10) and (6.14.11), as well as their inverses.

```

struct Studenttdist : Beta {
    Student-t distribution, derived from the beta function Beta.
    Doub nu, mu, sig, np, fac;
    Studenttdist(Doub nnu, Doub mmu = 0., Doub ssig = 1.)
        : nu(nnu), mu(mmu), sig(ssig) {
    Constructor. Initialize with  $\nu$ ,  $\mu$  and  $\sigma$ . The default with one argument is Student( $\nu, 0, 1$ ).

        if (sig <= 0. || nu <= 0.) throw("bad sig,nu in Studentdist");
        np = 0.5*(nu + 1.);
        fac = gammln(np)-gammln(0.5*nu);
    }
    Doub p(Doub t) {
        Return probability density function.
        return exp(-np*log(1.+SQR((t-mu)/sig)/nu)+fac)
            /(sqrt(3.14159265358979324*nu)*sig);
    }
    Doub cdf(Doub t) {
        Return cumulative distribution function.
        Doub p = 0.5*betai(0.5*nu, 0.5, nu/(nu+SQR((t-mu)/sig)));
        if (t >= mu) return 1. - p;
        else return p;
    }
    Doub invcdf(Doub p) {
        Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Studentdist");
        Doub x = invbetai(2.*MIN(p,1.-p), 0.5*nu, 0.5);
        x = sig*sqrt(nu*(1.-x)/x);
        return (p >= 0.5? mu+x : mu-x);
    }
    Doub aa(Doub t) {
        Return the two-tailed cdf  $A(t|\nu)$ .
        if (t < 0.) throw("bad t in Studentdist");
        return 1.-betai(0.5*nu, 0.5, nu/(nu+SQR(t)));
    }
    Doub invaa(Doub p) {
        Return the inverse, namely  $t$  such that  $p = A(t|\nu)$ .
        if (p < 0. || p >= 1.) throw("bad p in Studentdist");
        Doub x = invbetai(1.-p, 0.5*nu, 0.5);
        return sqrt(nu*(1.-x)/x);
    }
};
```

#### 6.14.4 Logistic Distribution

The *logistic distribution* is another symmetric, centrally peaked distribution that can be used instead of the normal distribution. Its tails decay exponentially, but still much more slowly than the normal distribution's "exponent of the square."

The defining probability density is

$$p(y) = \frac{e^{-y}}{(1 + e^{-y})^2} = \frac{e^y}{(1 + e^y)^2} = \frac{1}{4}\operatorname{sech}^2\left(\frac{1}{2}y\right) \quad (6.14.12)$$

The three forms are algebraically equivalent, but, to avoid overflows, it is wise to use the negative and positive exponential forms for positive and negative values of  $y$ , respectively.

The variance of the distribution (6.14.12) turns out to be  $\pi^2/3$ . Since it is convenient to have parameters  $\mu$  and  $\sigma$  with the conventional meanings of mean and standard deviation, equation (6.14.12) is often replaced by the *standardized logistic*

distribution,

$$x \sim \text{Logistic}(\mu, \sigma), \quad \sigma > 0$$

$$p(x) = \frac{\pi}{4\sqrt{3}\sigma} \text{sech}^2 \left( \frac{\pi}{2\sqrt{3}} \left[ \frac{x-\mu}{\sigma} \right] \right) \quad (6.14.13)$$

which implies equivalent forms using the positive and negative exponentials (see code below).

The cdf is given by

$$\text{cdf} \equiv P(< x) \equiv \int_{-\infty}^x p(x') dx' = \left[ 1 + \exp \left( -\frac{\pi}{\sqrt{3}} \left[ \frac{x-\mu}{\sigma} \right] \right) \right]^{-1} \quad (6.14.14)$$

The inverse cdf is given by

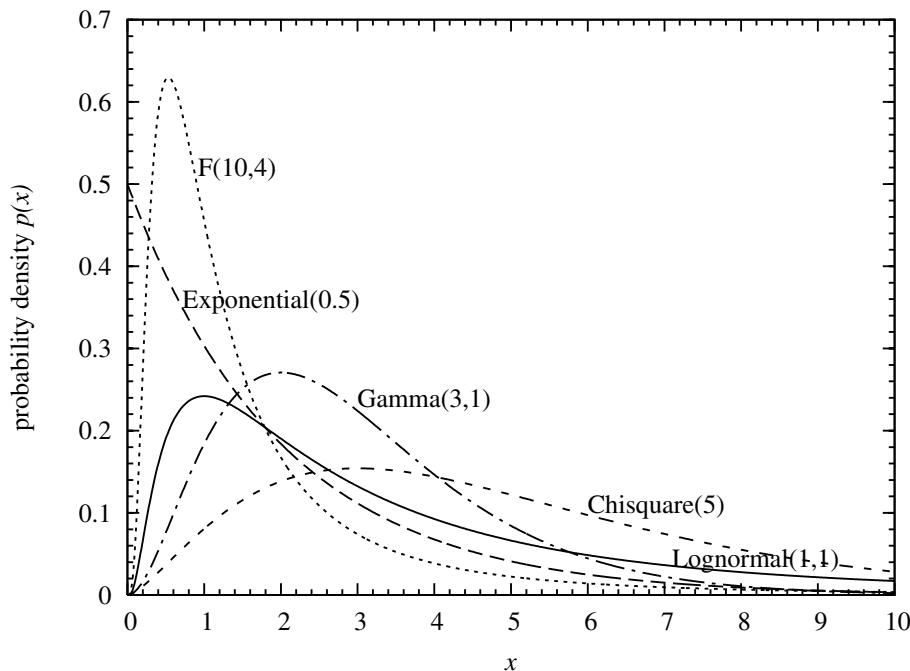
$$x(P) = \mu + \frac{\sqrt{3}}{\pi} \sigma \log \left( \frac{P}{1-P} \right) \quad (6.14.15)$$

```
struct Logisticdist {
    Logistic distribution.                                            distributions.h
    Doub mu, sig;
    Logisticdist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {
        Constructor. Initialize with  $\mu$  and  $\sigma$ . The default with no arguments is Logistic(0, 1).
        if (sig <= 0.) throw("bad sig in Logisticdist");
    }
    Doub p(Doub x) {
        Return probability density function.
        Doub e = exp(-abs(1.81379936423421785*(x-mu)/sig));
        return 1.81379936423421785*e/(sig*SQR(1.+e));
    }
    Doub cdf(Doub x) {
        Return cumulative distribution function.
        Doub e = exp(-abs(1.81379936423421785*(x-mu)/sig));
        if (x >= mu) return 1.0/(1.+e);                                Because we used abs to control over-
        else return e/(1.+e);                                         flow, we now have two cases.
    }
    Doub invcdf(Doub p) {
        Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Logisticdist");
        return mu + 0.551328895421792049*sig*log(p/(1.-p));
    }
};
```

The logistic distribution is cousin to the *logit transformation* that maps the open unit interval  $0 < p < 1$  onto the real line  $-\infty < u < \infty$  by the relation

$$u = \log \left( \frac{p}{1-p} \right) \quad (6.14.16)$$

Back when a book of tables and a slide rule were a statistician's working tools, the logit transformation was used to approximate processes on the interval by analytically simpler processes on the real line. A uniform distribution on the interval maps by the logit transformation to a logistic distribution on the real line. With the computer's ability to calculate distributions on the interval directly (beta distributions, for example), that motivation has vanished.



**Figure 6.14.2.** Examples of common distributions on the half-line  $x > 0$ .

Another cousin is the *logistic equation*,

$$\frac{dy}{dt} \propto y(y_{\max} - y) \quad (6.14.17)$$

a differential equation describing the growth of some quantity  $y$ , starting off as an exponential but reaching, asymptotically, a value  $y_{\max}$ . The solution of this equation is identical, up to a scaling, to the cdf of the logistic distribution.

### 6.14.5 Exponential Distribution

With the *exponential distribution* we now turn to common distribution functions defined on the positive real axis  $x \geq 0$ . Figure 6.14.2 shows examples of several of the distributions that we will discuss. The exponential is the simplest of them all. It has a parameter  $\beta$  that can control its width (in inverse relationship), but its mode is always at zero:

$$\begin{aligned} x &\sim \text{Exponential}(\beta), & \beta > 0 \\ p(x) &= \beta \exp(-\beta x), & x > 0 \end{aligned} \quad (6.14.18)$$

$$\text{cdf} \equiv P(< x) \equiv \int_0^x p(x') dx' = 1 - \exp(-\beta x) \quad (6.14.19)$$

$$x(P) = -\frac{1}{\beta} \log(1 - P) \quad (6.14.20)$$

The mean and standard deviation of the exponential distribution are both  $1/\beta$ . The median is  $\log(2)/\beta$ . Reference [1] has more to say about the exponential distribution than you would ever think possible.

```

struct Expondist {
    Exponential distribution.
    Doub bet;
    Expondist(Doub b) : bet(b) {}
    Constructor. Initialize with  $\beta$ .
    if (b <= 0.) throw("bad b in Expondist");
}
Doub p(Doub x) {
    Return probability density function.
    if (x < 0.) throw("bad x in Expondist");
    return bet*exp(-bet*x);
}
Doub cdf(Doub x) {
    Return cumulative distribution function.
    if (x < 0.) throw("bad x in Expondist");
    return 1.-exp(-bet*x);
}
Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
    if (p < 0. || p >= 1.) throw("bad p in Expondist");
    return -log(1.-p)/bet;
}
};

distributions.h

```

## 6.14.6 Weibull Distribution

The Weibull distribution generalizes the exponential distribution in a way that is often useful in hazard, survival, or reliability studies. When the lifetime (time to failure) of an item is exponentially distributed, there is a constant probability per unit time that an item will fail, if it has not already done so. That is,

$$\text{hazard} \equiv \frac{p(x)}{P(>x)} \propto \text{constant} \quad (6.14.21)$$

Exponentially lived items don't age; they just keep rolling the same dice until, one day, their number comes up. In many other situations, however, it is observed that an item's hazard (as defined above) does change with time, say as a power law,

$$\frac{p(x)}{P(>x)} \propto x^{\alpha-1}, \quad \alpha > 0 \quad (6.14.22)$$

The distribution that results is the Weibull distribution, named for Swedish physicist Waloddi Weibull, who used it as early as 1939. When  $\alpha > 1$ , the hazard increases with time, as for components that wear out. When  $0 < \alpha < 1$ , the hazard decreases with time, as for components that experience "infant mortality."

We say that

$$x \sim \text{Weibull}(\alpha, \beta) \quad \text{iff} \quad y \equiv \left(\frac{x}{\beta}\right)^{\alpha} \sim \text{Exponential}(1) \quad (6.14.23)$$

The probability density is

$$p(x) = \left(\frac{\alpha}{\beta}\right) \left(\frac{x}{\beta}\right)^{\alpha-1} e^{-(x/\beta)^{\alpha}}, \quad x > 0 \quad (6.14.24)$$

The cdf is

$$\text{cdf} \equiv P(< x) \equiv \int_0^x p(x') dx' = 1 - e^{-(x/\beta)^\alpha} \quad (6.14.25)$$

The inverse cdf is

$$x(P) = \beta [-\log(1 - P)]^{1/\alpha} \quad (6.14.26)$$

For  $0 < \alpha < 1$ , the distribution has an infinite (but integrable) cusp at  $x = 0$  and is monotonically decreasing. The exponential distribution is the case of  $\alpha = 1$ . When  $\alpha > 1$ , the distribution is zero at  $x = 0$  and has a single maximum at the value  $x = \beta [(\alpha - 1)/\alpha]^{1/\alpha}$ .

The mean and variance are given by

$$\begin{aligned} \mu &= \beta \Gamma(1 + \alpha^{-1}) \\ \sigma^2 &= \beta^2 \left\{ \Gamma(1 + 2\alpha^{-1}) - [\Gamma(1 + \alpha^{-1})]^2 \right\} \end{aligned} \quad (6.14.27)$$

With correct normalization, equation (6.14.22) becomes

$$\text{hazard} \equiv \frac{p(x)}{P(> x)} = \left( \frac{\alpha}{\beta} \right) \left( \frac{x}{\beta} \right)^{\alpha-1} \quad (6.14.28)$$

### 6.14.7 Lognormal Distribution

Many processes that live on the positive  $x$ -axis are naturally approximated by normal distributions on the “ $\log(x)$ -axis,” that is, for  $-\infty < \log(x) < \infty$ . A simple, but important, example is the multiplicative random walk, which starts at some positive value  $x_0$ , and then generates new values by a recurrence like

$$x_{i+1} = \begin{cases} x_i(1 + \epsilon) & \text{with probability 0.5} \\ x_i/(1 + \epsilon) & \text{with probability 0.5} \end{cases} \quad (6.14.29)$$

Here  $\epsilon$  is some small, fixed, constant.

These considerations motivate the definition

$$x \sim \text{Lognormal}(\mu, \sigma) \quad \text{iff} \quad u \equiv \frac{\log(x) - \mu}{\sigma} \sim N(0, 1) \quad (6.14.30)$$

or the equivalent definition

$$\begin{aligned} x &\sim \text{Lognormal}(\mu, \sigma), \quad \sigma > 0 \\ p(x) &= \frac{1}{\sqrt{2\pi}\sigma x} \exp\left(-\frac{1}{2} \left[ \frac{\log(x) - \mu}{\sigma} \right]^2\right), \quad x > 0 \end{aligned} \quad (6.14.31)$$

Note the required extra factor of  $x^{-1}$  in front of the exponential: The density that is “normal” is  $p(\log x)d \log x$ .

While  $\mu$  and  $\sigma$  are the mean and standard deviation in  $\log x$  space, they are *not* so in  $x$  space. Rather,

$$\begin{aligned} \text{Mean}\{\text{Lognormal}(\mu, \sigma)\} &= e^{\mu + \frac{1}{2}\sigma^2} \\ \text{Var}\{\text{Lognormal}(\mu, \sigma)\} &= e^{2\mu} e^{\sigma^2} (e^{\sigma^2} - 1) \end{aligned} \quad (6.14.32)$$

The cdf is given by

$$\text{cdf} \equiv P(< x) \equiv \int_0^x p(x') dx' = \frac{1}{2} \operatorname{erfc} \left( -\frac{1}{\sqrt{2}} \left[ \frac{\log(x) - \mu}{\sigma} \right] \right) \quad (6.14.33)$$

The inverse to the cdf involves the inverse complementary error function,

$$x(P) = \exp[\mu - \sqrt{2}\sigma \operatorname{erfc}^{-1}(2P)] \quad (6.14.34)$$

```
struct Lognormaldist : Erf {
    Lognormal distribution, derived from the error function Erf.
    Doub mu, sig;
    Lognormaldist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {
        if (sig <= 0.) throw("bad sig in Lognormaldist");
    }
    Doub p(Doub x) {
        Return probability density function.
        if (x < 0.) throw("bad x in Lognormaldist");
        if (x == 0.) return 0.;
        return (0.398942280401432678/(sig*x))*exp(-0.5*SQR((log(x)-mu)/sig));
    }
    Doub cdf(Doub x) {
        Return cumulative distribution function.
        if (x < 0.) throw("bad x in Lognormaldist");
        if (x == 0.) return 0.;
        return 0.5*erfc(-0.707106781186547524*(log(x)-mu)/sig);
    }
    Doub invcdf(Doub p) {
        Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Lognormaldist");
        return exp(-1.41421356237309505*sig*inverfc(2.*p)+mu);
    }
};
```

Multiplicative random walks like (6.14.29) and lognormal distributions are key ingredients in the economic theory of efficient markets, leading to (among many other results) the celebrated *Black-Scholes formula* for the probability distribution of the price of an investment after some elapsed time  $\tau$ . A key piece of the Black-Scholes derivation is implicit in equation (6.14.32): If an investment's average return is zero (which may be true in the limit of zero risk), then its price cannot simply be a widening lognormal distribution with fixed  $\mu$  and increasing  $\sigma$ , for its expected value would then diverge to infinity! The actual Black-Scholes formula thus defines both how  $\sigma$  increases with time (basically as  $\tau^{1/2}$ ) and how  $\mu$  correspondingly decreases with time, so as to keep the overall mean under control. A simplified version of the Black-Scholes formula can be written as

$$S(\tau) \sim S(0) \times \text{Lognormal} \left( r\tau - \frac{1}{2}\sigma^2\tau, \sigma\sqrt{\tau} \right) \quad (6.14.35)$$

where  $S(\tau)$  is the price of a stock at time  $\tau$ ,  $r$  is its expected (annualized) rate of return, and  $\sigma$  is now redefined to be the stock's (annualized) volatility. The definition of volatility is that, for small values of  $\tau$ , the fractional variance of the stock's price is  $\sigma^2\tau$ . You can check that (6.14.35) has the desired expectation value  $E[S(\tau)] = S(0)$ , for all  $\tau$ , if  $r = 0$ . A good reference is [3].

### 6.14.8 Chi-Square Distribution

The *chi-square* (or  $\chi^2$ ) distribution has a single parameter  $\nu > 0$  that controls both the location and width of its peak. In most applications  $\nu$  is an integer and is referred to as the *number of degrees of freedom* (see §14.3).

The defining probability density is

$$\begin{aligned} \chi^2 &\sim \text{Chisquare}(\nu), \quad \nu > 0 \\ p(\chi^2) d\chi^2 &= \frac{1}{2^{\frac{1}{2}\nu} \Gamma(\frac{1}{2}\nu)} (\chi^2)^{\frac{1}{2}\nu-1} \exp(-\frac{1}{2}\chi^2) d\chi^2, \quad \chi^2 > 0 \end{aligned} \quad (6.14.36)$$

where we have written the differentials  $d\chi^2$  merely to emphasize that  $\chi^2$ , not  $\chi$ , is to be viewed as the independent variable.

The mean and variance are given by

$$\begin{aligned} \text{Mean}\{\text{Chisquare}(\nu)\} &= \nu \\ \text{Var}\{\text{Chisquare}(\nu)\} &= 2\nu \end{aligned} \quad (6.14.37)$$

When  $\nu \geq 2$  there is a single mode at  $\chi^2 = \nu - 2$ .

The chi-square distribution is actually just a special case of the gamma distribution, below, so its cdf is given by an incomplete gamma function  $P(a, x)$ ,

$$\text{cdf} \equiv P(< \chi^2) \equiv P(\chi^2 | \nu) \equiv \int_0^{\chi^2} p(\chi^{2'}) d\chi^{2'} = P\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) \quad (6.14.38)$$

One frequently also sees the complement of the cdf, which can be calculated either from the incomplete gamma function  $P(a, x)$ , or from its complement  $Q(a, x)$  (often more accurate if  $P$  is very close to 1):

$$Q(\chi^2 | \nu) \equiv 1 - P(\chi^2 | \nu) = 1 - P\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) \equiv Q\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) \quad (6.14.39)$$

The inverse cdf is given in terms of the function that is the inverse of  $P(a, x)$  on its second argument, which we here denote  $P^{-1}(a, p)$ :

$$x(P) = 2P^{-1}\left(\frac{\nu}{2}, P\right) \quad (6.14.40)$$

```
ncgammabeta.h
struct Chisqdist : Gamma {
     $\chi^2$  distribution, derived from the gamma function Gamma.
    Doub nu,fac;
    Chisqdist(Doub nnu) : nu(nnu) {
        Constructor. Initialize with  $\nu$ .
        if (nu <= 0.) throw("bad nu in Chisqdist");
        fac = 0.693147180559945309*(0.5*nu)+gammln(0.5*nu);
    }
    Doub p(Doub x2) {
        Return probability density function.
        if (x2 <= 0.) throw("bad x2 in Chisqdist");
        return exp(-0.5*(x2-(nu-2.)*log(x2))-fac);
    }
    Doub cdf(Doub x2) {
```

```

Return cumulative distribution function.
    if (x2 < 0.) throw("bad x2 in Chisqdist");
    return gammp(0.5*nu,0.5*x2);
}
Doub invcdf(Doub p) {
Return inverse cumulative distribution function.
    if (p < 0. || p >= 1.) throw("bad p in Chisqdist");
    return 2.*invgammp(p,0.5*nu);
}
};

```

### 6.14.9 Gamma Distribution

The *gamma distribution* is defined by

$$\begin{aligned} x &\sim \text{Gamma}(\alpha, \beta), \quad \alpha > 0, \beta > 0 \\ p(x) &= \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}, \quad x > 0 \end{aligned} \quad (6.14.41)$$

The exponential distribution is the special case with  $\alpha = 1$ . The chi-square distribution is the special case with  $\alpha = \nu/2$  and  $\beta = 1/2$ .

The mean and variance are given by,

$$\begin{aligned} \text{Mean}\{\text{Gamma}(\alpha, \beta)\} &= \alpha/\beta \\ \text{Var}\{\text{Gamma}(\alpha, \beta)\} &= \alpha/\beta^2 \end{aligned} \quad (6.14.42)$$

When  $\alpha \geq 1$  there is a single mode at  $x = (\alpha - 1)/\beta$ .

Evidently, the cdf is the incomplete gamma function

$$\text{cdf} \equiv P(< x) \equiv \int_0^x p(x') dx' = P(\alpha, \beta x) \quad (6.14.43)$$

while the inverse cdf is given in terms of the inverse of  $P(a, x)$  on its second argument by

$$x(P) = \frac{1}{\beta} P^{-1}(\alpha, P) \quad (6.14.44)$$

```

struct Gammadist : Gamma {
    Doub alph, bet, fac;
    Gammadist(Doub aalph, Doub bbet = 1.) : alph(aalph), bet(bbet) {
        Constructor. Initialize with  $\alpha$  and  $\beta$ .
        if (alph <= 0. || bet <= 0.) throw("bad alph,bet in Gammadist");
        fac = alph*log(bet)-gammln(alph);
    }
    Doub p(Doub x) {
        Return probability density function.
        if (x <= 0.) throw("bad x in Gammadist");
        return exp(-bet*x+(alph-1.)*log(x)+fac);
    }
    Doub cdf(Doub x) {
        Return cumulative distribution function.
        if (x < 0.) throw("bad x in Gammadist");
        return gammp(alph,bet*x);
    }
};
```

incgammabeta.h

```

Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
    if (p < 0. || p >= 1.) throw("bad p in Gammadist");
    return invgammp(p,alph)/bet;
}

```

### 6.14.10 F-Distribution

The *F-distribution* is parameterized by two positive values  $v_1$  and  $v_2$ , usually (but not always) integers.

The defining probability density is

$$F \sim F(v_1, v_2), \quad v_1 > 0, v_2 > 0$$

$$p(F) = \frac{\frac{1}{2}^{v_1} \frac{1}{2}^{v_2}}{B(\frac{1}{2}v_1, \frac{1}{2}v_2)} \frac{F^{\frac{1}{2}v_1-1}}{(v_2 + v_1 F)^{(v_1+v_2)/2}}, \quad F > 0 \quad (6.14.45)$$

where  $B(a, b)$  denotes the beta function. The mean and variance are given by

$$\text{Mean}\{F(v_1, v_2)\} = \frac{v_2}{v_2 - 2}, \quad v_2 > 2 \quad (6.14.46)$$

$$\text{Var}\{F(v_1, v_2)\} = \frac{2v_2^2(v_1 + v_2 - 2)}{v_1(v_2 - 2)^2(v_2 - 4)}, \quad v_2 > 4$$

When  $v_1 \geq 2$  there is a single mode at

$$F = \frac{v_2(v_1 - 2)}{v_1(v_2 + 2)} \quad (6.14.47)$$

For fixed  $v_1$ , if  $v_2 \rightarrow \infty$ , the *F*-distribution becomes a chi-square distribution, namely

$$\lim_{v_2 \rightarrow \infty} F(v_1, v_2) \cong \frac{1}{v_1} \text{Chisquare}(v_1) \quad (6.14.48)$$

where “ $\cong$ ” means “are identical distributions.”

The *F*-distribution’s cdf is given in terms of the incomplete beta function  $I_x(a, b)$  by

$$\text{cdf} \equiv P(< x) \equiv \int_0^x p(x') dx' = I_{v_1 F / (v_2 + v_1 F)} \left( \frac{1}{2}v_1, \frac{1}{2}v_2 \right) \quad (6.14.49)$$

while the inverse cdf is given in terms of the inverse of  $I_x(a, b)$  on its subscript argument by

$$u \equiv I_p^{-1} \left( \frac{1}{2}v_1, \frac{1}{2}v_2 \right)$$

$$x(P) = \frac{v_2 u}{v_1(1-u)} \quad (6.14.50)$$

A frequent use of the *F*-distribution is to test whether two observed samples have the same variance.

```

struct Fdist : Beta {
    F distribution, derived from the beta function Beta.
    Doub nu1,nu2;
    Doub fac;
    Fdist(Doub nnu1, Doub nnu2) : nu1(nnu1), nu2(nnu2) {
        Constructor. Initialize with ν1 and ν2.
        if (nu1 <= 0. || nu2 <= 0.) throw("bad nu1,nu2 in Fdist");
        fac = 0.5*(nu1*log(nu1)+nu2*log(nu2))+gammln(0.5*(nu1+nu2))
            -gammln(0.5*nu1)-gammln(0.5*nu2);
    }
    Doub p(Doub f) {
        Return probability density function.
        if (f <= 0.) throw("bad f in Fdist");
        return exp((0.5*nu1-1.)*log(f)-0.5*(nu1+nu2)*log(nu2+nu1*f)+fac);
    }
    Doub cdf(Doub f) {
        Return cumulative distribution function.
        if (f < 0.) throw("bad f in Fdist");
        return betai(0.5*nu1,0.5*nu2,nu1*f/(nu2+nu1*f));
    }
    Doub invcdf(Doub p) {
        Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Fdist");
        Doub x = invbetai(p,0.5*nu1,0.5*nu2);
        return nu2*x/(nu1*(1.-x));
    }
};
```

### 6.14.11 Beta Distribution

The *beta distribution* is defined on the unit interval  $0 < x < 1$  by

$$x \sim \text{Beta}(\alpha, \beta), \quad \alpha > 0, \beta > 0$$

$$p(x) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1} (1-x)^{\beta-1}, \quad 0 < x < 1 \quad (6.14.51)$$

The mean and variance are given by

$$\text{Mean}\{\text{Beta}(\alpha, \beta)\} = \frac{\alpha}{\alpha + \beta} \quad (6.14.52)$$

$$\text{Var}\{\text{Beta}(\alpha, \beta)\} = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$$

When  $\alpha > 1$  and  $\beta > 1$ , there is a single mode at  $(\alpha - 1)/(\alpha + \beta - 2)$ . When  $\alpha < 1$  and  $\beta < 1$ , the distribution function is “U-shaped” with a minimum at this same value. In other cases there is neither a maximum nor a minimum.

In the limit that  $\beta$  becomes large as  $\alpha$  is held fixed, all the action in the beta distribution shifts toward  $x = 0$ , and the density function takes the shape of a gamma distribution. More precisely,

$$\lim_{\beta \rightarrow \infty} \beta \text{Beta}(\alpha, \beta) \cong \text{Gamma}(\alpha, 1) \quad (6.14.53)$$

The cdf is the incomplete beta function

$$\text{cdf} \equiv P(< x) \equiv \int_0^x p(x') dx' = I_x(\alpha, \beta) \quad (6.14.54)$$

while the inverse cdf is given in terms of the inverse of  $I_x(\alpha, \beta)$  on its subscript argument by

$$x(P) = I_p^{-1}(\alpha, \beta) \quad (6.14.55)$$

ncgammabeta.h

```
struct Betadist : Beta {
    Beta distribution, derived from the beta function Beta.
    Doub alph, bet, fac;
    Betadist(Doub aalph, Doub bbet) : alph(aalph), bet(bbet) {
        Constructor. Initialize with  $\alpha$  and  $\beta$ .
        if (alph <= 0. || bet <= 0.) throw("bad alph,bet in Betadist");
        fac = gammln(alph+bet)-gammln(alph)-gammln(bet);
    }
    Doub p(Doub x) {
        Return probability density function.
        if (x <= 0. || x >= 1.) throw("bad x in Betadist");
        return exp((alph-1.)*log(x)+(bet-1.)*log(1.-x)+fac);
    }
    Doub cdf(Doub x) {
        Return cumulative distribution function.
        if (x < 0. || x > 1.) throw("bad x in Betadist");
        return betai(alph,bet,x);
    }
    Doub invcdf(Doub p) {
        Return inverse cumulative distribution function.
        if (p < 0. || p > 1.) throw("bad p in Betadist");
        return invbetai(p,alph,bet);
    }
};
```

### 6.14.12 Kolmogorov-Smirnov Distribution

The *Kolmogorov-Smirnov* or *KS* distribution, defined for positive  $z$ , is key to an important statistical test that is discussed in §14.3. Its probability density function does not directly enter into the test and is virtually never even written down. What one typically needs to compute is the cdf, denoted  $P_{KS}(z)$ , or its complement,  $Q_{KS}(z) \equiv 1 - P_{KS}(z)$ .

The cdf  $P_{KS}(z)$  is defined by the series

$$P_{KS}(z) = 1 - 2 \sum_{j=1}^{\infty} (-1)^{j-1} \exp(-2j^2 z^2) \quad (6.14.56)$$

or by the equivalent series (nonobviously so!)

$$P_{KS}(z) = \frac{\sqrt{2\pi}}{z} \sum_{j=1}^{\infty} \exp\left(-\frac{(2j-1)^2 \pi^2}{8z^2}\right) \quad (6.14.57)$$

Limiting values are what you'd expect for cdf's named "P" and "Q":

$$\begin{aligned} P_{KS}(0) &= 0 & P_{KS}(\infty) &= 1 \\ Q_{KS}(0) &= 1 & Q_{KS}(\infty) &= 0 \end{aligned} \quad (6.14.58)$$

Both of the series (6.14.56) and (6.14.57) are convergent for all  $z > 0$ . Moreover, for any  $z$ , one or the other series converges extremely rapidly, requiring no more

than three terms to get to IEEE double precision fractional accuracy. A good place to switch from one series to the other is at  $z \approx 1.18$ . This renders the KS functions computable by a single exponential and a small number of arithmetic operations (see code below).

Getting the inverse functions  $P_{KS}^{-1}(P)$  and  $Q_{KS}^{-1}(Q)$ , which return a value of  $z$  from a  $P$  or  $Q$  value, is a little trickier. For  $Q \lesssim 0.3$  (that is,  $P \gtrsim 0.7$ ), an iteration based on (6.14.56) works nicely:

$$\begin{aligned}x_0 &\equiv 0 \\x_{i+1} &= \frac{1}{2}Q + x_i^4 - x_i^9 + x_i^{16} - x_i^{25} + \dots \\z(Q) &= \sqrt{-\frac{1}{2} \log(x_\infty)}\end{aligned}\tag{6.14.59}$$

For  $x \lesssim 0.06$  you only need the first two powers of  $x_i$ .

For larger values of  $Q$ , that is,  $P \lesssim 0.7$ , the number of powers of  $x$  required quickly becomes excessive. A useful approach is to write (6.14.57) as

$$\begin{aligned}y \log(y) &= -\frac{\pi P^2}{8} \left(1 + y^4 + y^{12} + \dots + y^{2j(j-1)} + \dots\right)^{-1} \\z(P) &= \frac{\pi/2}{\sqrt{-\log(y)}}\end{aligned}\tag{6.14.60}$$

If we can get a good enough initial guess for  $y$ , we can solve the first equation in (6.14.60) by a variant of Halley's method: Use values of  $y$  from the *previous* iteration on the right-hand side of (6.14.60), and use Halley only for the  $y \log(y)$  piece, so that the first and second derivatives are analytically simple functions.

A good initial guess is obtained by using the inverse function to  $y \log(y)$  (the function `invxlogx` in §6.11) with the argument  $-\pi P^2/8$ . The number of iterations within the `invxlogx` function and the Halley loop is never more than half a dozen in each, often less. Code for the KS functions and their inverses follows.

```
struct KSdist {
    Kolmogorov-Smirnov cumulative distribution functions and their inverses.
    Doub pks(Doub z) {
        Return cumulative distribution function.
        if (z < 0.) throw("bad z in KSdist");
        if (z == 0.) return 0.;
        if (z < 1.18) {
            Doub y = exp(-1.23370055013616983/SQR(z));
            return 2.25675833419102515*sqrt(-log(y))
                   *(y + pow(y,9) + pow(y,25) + pow(y,49));
        } else {
            Doub x = exp(-2.*SQR(z));
            return 1. - 2.*(x - pow(x,4) + pow(x,9));
        }
    }
    Doub qks(Doub z) {
        Return complementary cumulative distribution function.
        if (z < 0.) throw("bad z in KSdist");
        if (z == 0.) return 1.;
        if (z < 1.18) return 1.-pks(z);
        Doub x = exp(-2.*SQR(z));
        return 2.*(x - pow(x,4) + pow(x,9));
    }
    Doub invqks(Doub q) {
```

ksdist.h

Return inverse of the complementary cumulative distribution function.

```

Doub y,logy,yp,x,xp,f,ff,u,t;
if (q <= 0. || q > 1.) throw("bad q in KSdist");
if (q == 1.) return 0.;
if (q > 0.3) {
    f = -0.392699081698724155*SQR(1.-q);
    y = invxlogx(f);                                Initial guess.
    do {
        yp = y;
        logy = log(y);
        ff = f/SQR(1.+ pow(y,4)+ pow(y,12));
        u = (y*logy-ff)/(1.+logy);      Newton's method correction.
        y = y - (t=u/MAX(0.5,1.-0.5*u/(y*(1.+logy)))); Halley.
    } while (abs(t/y)>1.e-15);
    return 1.57079632679489662/sqrt(-log(y));
} else {
    x = 0.03;
    do {                                         Iteration (6.14.59).
        xp = x;
        x = 0.5*q+pow(x,4)-pow(x,9);
        if (x > 0.06) x += pow(x,16)-pow(x,25);
    } while (abs((xp-x)/x)>1.e-15);
    return sqrt(-0.5*log(x));
}
}
Doub invpkz(Doub p) {return invqks(1.-p);}
Return inverse of the cumulative distribution function.
};
```

### 6.14.13 Poisson Distribution

The eponymous *Poisson distribution* was derived by Poisson in 1837. It applies to a process where discrete, uncorrelated events occur at some mean rate per unit time. If, for a given period,  $\lambda$  is the mean expected number of events, then the probability distribution of seeing exactly  $k$  events,  $k \geq 0$ , can be written as

$$\begin{aligned} k &\sim \text{Poisson}(\lambda), \quad \lambda > 0 \\ p(k) &= \frac{1}{k!} \lambda^k e^{-\lambda}, \quad k = 0, 1, \dots \end{aligned} \tag{6.14.61}$$

Evidently  $\sum_k p(k) = 1$ , since the  $k$ -dependent factors in (6.14.61) are just the series expansion of  $e^\lambda$ .

The mean and variance of  $\text{Poisson}(\lambda)$  are both  $\lambda$ . There is a single mode at  $k = \lfloor \lambda \rfloor$ , that is, at  $\lambda$  rounded down to an integer.

The Poisson distribution's cdf is an incomplete gamma function  $Q(a, x)$ ,

$$P_\lambda(< k) = Q(k, \lambda) \tag{6.14.62}$$

Since  $k$  is discrete,  $P_\lambda(< k)$  is of course different from  $P_\lambda(\leq k)$ , the latter being given by

$$P_\lambda(\leq k) = Q(k + 1, \lambda) \tag{6.14.63}$$

Some particular values are

$$P_\lambda(< 0) = 0 \quad P_\lambda(< 1) = e^{-\lambda} \quad P_\lambda(< \infty) = 1 \tag{6.14.64}$$

Some other relations involving the incomplete gamma functions  $Q(a, x)$  and  $P(a, x)$  are

$$\begin{aligned} P_\lambda(\geq k) &= P(k, \lambda) = 1 - Q(k, \lambda) \\ P_\lambda(> k) &= P(k + 1, \lambda) = 1 - Q(k + 1, \lambda) \end{aligned} \quad (6.14.65)$$

Because of the discreteness in  $k$ , the inverse of the cdf must be defined with some care: Given a value  $P$ , we define  $k_\lambda(P)$  as the integer such that

$$P_\lambda(< k) \leq P < P_\lambda(\leq k) \quad (6.14.66)$$

In the interest of conciseness, the code below cheats a little bit and allows the right-hand  $<$  to be  $\leq$ . If you may be supplying  $P$ 's that are *exact*  $P_\lambda(< k)$ 's, then you will need to check both  $k_\lambda(P)$  as returned, and  $k_\lambda(P) + 1$ . (This will essentially never happen for “round”  $P$ 's like 0.95, 0.99, etc.)

```
struct Poisondist : Gamma {  
    Poisson distribution, derived from the gamma function Gamma.  
    Doub lam;  
    Poisondist(Doub llam) : lam(llam) {  
        Constructor. Initialize with  $\lambda$ .  
        if (lam <= 0.) throw("bad lam in Poisondist");  
    }  
    Doub p(Int n) {  
        Return probability density function.  
        if (n < 0) throw("bad n in Poisondist");  
        return exp(-lam + n*log(lam) - gammaln(n+1.));  
    }  
    Doub cdf(Int n) {  
        Return cumulative distribution function.  
        if (n < 0) throw("bad n in Poisondist");  
        if (n == 0) return 0.;  
        return gammq((Doub)n, lam);  
    }  
    Int invcdf(Doub p) {  
        Given argument  $P$ , return integer  $n$  such that  $P(< n) \leq P \leq P(< n + 1)$ .  
        Int n,nl,nu,inc=1;  
        if (p <= 0. || p >= 1.) throw("bad p in Poisondist");  
        if (p < exp(-lam)) return 0;  
        n = (Int)MAX(sqrt(lam),5.);  
        if (p < cdf(n)) {  
            Starting guess near peak of density.  
            do {  
                inc *= 2;  
                n = MAX(n-inc,0);  
            } while (p < cdf(n));  
            nl = n; nu = n + inc/2;  
        } else {  
            do {  
                inc *= 2;  
                n += inc;  
            } while (p > cdf(n));  
            nu = n; nl = n - inc/2;  
        }  
        while (nu-nl>1) {  
            Now contract the interval by bisection.  
            n = (nl+nu)/2;  
            if (p < cdf(n)) nu = n;  
            else nl = n;  
        }  
        return nl;  
    };  
};
```

### 6.14.14 Binomial Distribution

Like the Poisson distribution, the *binomial distribution* is a discrete distribution over  $k \geq 0$ . It has two parameters,  $n \geq 1$ , the “sample size” or maximum value for which  $k$  can be nonzero; and  $p$ , the “event probability” (not to be confused with  $p(k)$ , the probability of a particular  $k$ ). We write

$$\begin{aligned} k &\sim \text{Binomial}(n, p), \quad n \geq 1, 0 < p < 1 \\ p(k) &= \binom{n}{k} p^k (1-p)^{n-k}, \quad k = 0, 1, \dots, n \end{aligned} \tag{6.14.67}$$

where  $\binom{n}{k}$  is, of course, the binomial coefficient.

The mean and variance are given by

$$\begin{aligned} \text{Mean}\{\text{Binomial}(n, p)\} &= np \\ \text{Var}\{\text{Binomial}(n, p)\} &= np(1-p) \end{aligned} \tag{6.14.68}$$

There is a single mode at the value  $k$  that satisfies

$$(n+1)p - 1 < k \leq (n+1)p \tag{6.14.69}$$

The distribution is symmetrical iff  $p = \frac{1}{2}$ . Otherwise it has positive skewness for  $p < \frac{1}{2}$  and negative for  $p > \frac{1}{2}$ . Many additional properties are described in [2].

The Poisson distribution is obtained from the binomial distribution in the limit  $n \rightarrow \infty$ ,  $p \rightarrow 0$  with the  $np$  remaining finite. More precisely,

$$\lim_{n \rightarrow \infty} \text{Binomial}(n, \lambda/n) \cong \text{Poisson}(\lambda) \tag{6.14.70}$$

The binomial distribution’s cdf can be computed from the incomplete beta function  $I_x(a, b)$ ,

$$P(< k) = 1 - I_p(k, n - k + 1) \tag{6.14.71}$$

so we also have (analogously to the Poisson distribution)

$$\begin{aligned} P(\leq k) &= 1 - I_p(k + 1, n - k) \\ P(> k) &= I_p(k + 1, n - k) \\ P(\geq k) &= I_p(k, n - k + 1) \end{aligned} \tag{6.14.72}$$

Some particular values are

$$P(< 0) = 0 \quad P(< [n+1]) = 1 \tag{6.14.73}$$

The inverse cdf is defined exactly as for the Poisson distribution, above, and with the same small warning about the code.

`ncgammabeta.h`

```
struct Binomialdist : Beta {
    Binomial distribution, derived from the beta function Beta.
    Int n;
    Doub pe, fac;
    Binomialdist(Int nn, Doub ppe) : n(nn), pe(ppe) {
        Constructor. Initialize with n (sample size) and p (event probability).
        if (n <= 0 || pe <= 0. || pe >= 1.) throw("bad args in Binomialdist");
```

```

        fac = gammeln(n+1.);
    }
Doub p(Int k) {
Return probability density function.
    if (k < 0) throw("bad k in Binomialdist");
    if (k > n) return 0.;
    return exp(k*log(pe)+(n-k)*log(1.-pe)
        +fac-gammeln(k+1.)-gammeln(n-k+1.));
}
Doub cdf(Int k) {
Return cumulative distribution function.
    if (k < 0) throw("bad k in Binomialdist");
    if (k == 0) return 0. ;
    if (k > n) return 1. ;
    return 1. - betai((Doub)k,n-k+1.,pe);
}
Int invcdf(Doub p) {
Given argument  $P$ , return integer  $n$  such that  $P(< n) \leq P \leq P(< n + 1)$ .
    Int k,kl,ku,inc=1;
    if (p <= 0. || p >= 1.) throw("bad p in Binomialdist");
    k = MAX(0,MIN(n,(Int)(n*pe)));           Starting guess near peak of density.
    if (p < cdf(k)) {                         Expand interval until we bracket.
        do {
            k = MAX(k-inc,0);
            inc *= 2;
        } while (p < cdf(k));
        kl = k; ku = k + inc/2;
    } else {
        do {
            k = MIN(k+inc,n+1);
            inc *= 2;
        } while (p > cdf(k));
        ku = k; kl = k - inc/2;
    }
    while (ku-kl>1) {                         Now contract the interval by bisection.
        k = (kl+ku)/2;
        if (p < cdf(k)) ku = k;
        else kl = k;
    }
    return kl;
}
};

```

#### CITED REFERENCES AND FURTHER READING:

- Johnson, N.L. and Kotz, S. 1970, *Continuous Univariate Distributions*, 2 vols. (Boston: Houghton Mifflin).[1]
- Johnson, N.L. and Kotz, S. 1969, *Discrete Distributions* (Boston: Houghton Mifflin).[2]
- Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. 2003, *Bayesian Data Analysis*, 2nd ed. (Boca Raton, FL: Chapman & Hall/CRC), Appendix A.
- Lyuu, Y-D. 2002, *Financial Engineering and Computation* (Cambridge, UK: Cambridge University Press).[3]

# Random Numbers

## 7.0 Introduction

It may seem perverse to use a computer, that most precise and deterministic of all machines conceived by the human mind, to produce “random” numbers. More than perverse, it may seem to be a conceptual impossibility. After all, any program produces output that is entirely predictable, hence not truly “random.”

Nevertheless, practical computer “random number generators” are in common use. We will leave it to philosophers of the computer age to resolve the paradox in a deep way (see, e.g., Knuth [1] §3.5 for discussion and references). One sometimes hears computer-generated sequences termed *pseudo-random*, while the word *random* is reserved for the output of an intrinsically random physical process, like the elapsed time between clicks of a Geiger counter placed next to a sample of some radioactive element. We will not try to make such fine distinctions.

A working definition of randomness in the context of computer-generated sequences is to say that the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that *uses* its output. In other words, any two different random number generators ought to produce statistically the same results when coupled to your particular applications program. If they don’t, then at least one of them is not (from your point of view) a good generator.

The above definition may seem circular, comparing, as it does, one generator to another. However, there exists a large body of random number generators that mutually do satisfy the definition over a very, very broad class of applications programs. And it is also found empirically that statistically identical results are obtained from random numbers produced by physical processes. So, because such generators are known to exist, we can leave to the philosophers the problem of defining them.

The pragmatic point of view is thus that randomness is in the eye of the beholder (or programmer). What is random enough for one application may not be random enough for another. Still, one is not entirely adrift in a sea of incommensurable applications programs: There is an accepted list of statistical tests, some sensible and some merely enshrined by history, that on the whole do a very good job of ferreting out any nonrandomness that is likely to be detected by an applications program (in this case, yours). Good random number generators ought to pass all of these tests,

or at least the user had better be aware of any that they fail, so that he or she will be able to judge whether they are relevant to the case at hand.

For references on this subject, the one to turn to first is Knuth [1]. Be cautious about any source earlier than about 1995, since the field progressed enormously in the following decade.

#### CITED REFERENCES AND FURTHER READING:

- Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 3, especially §3.5.[1]  
Gentle, J.E. 2003, *Random Number Generation and Monte Carlo Methods*, 2nd ed. (New York: Springer).

## 7.1 Uniform Deviates

Uniform deviates are just random numbers that lie within a specified range, typically 0.0 to 1.0 for floating-point numbers, or 0 to  $2^{32} - 1$  or  $2^{64} - 1$  for integers. Within the range, any one number is just as likely as any other. They are, in other words, what you probably think “random numbers” are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example, numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work.

The state of the art for generating uniform deviates has advanced considerably in the last decade and now begins to resemble a mature field. It is now reasonable to expect to get “perfect” deviates in no more than a dozen or so arithmetic or logical operations per deviate, and fast, “good enough” deviates in many fewer operations than that. Three factors have all contributed to the field’s advance: first, new mathematical algorithms; second, better understanding of the practical pitfalls; and, third, standardization of programming languages in general, and of integer arithmetic in particular — and especially the universal availability of unsigned 64-bit arithmetic in C and C++. It may seem ironic that something as down-in-the-weeds as this last factor can be so important. But, as we will see, it really is.

The greatest lurking danger for a user today is that many out-of-date and inferior methods remain in general use. Here are some traps to watch for:

- Never use a generator principally based on a *linear congruential generator* (LCG) or a *multiplicative linear congruential generator* (MLCG). We say more about this below.
- Never use a generator with a period less than  $\sim 2^{64} \approx 2 \times 10^{19}$ , or any generator whose period is undisclosed.
- Never use a generator that warns against using its low-order bits as being completely random. That was good advice once, but it now indicates an obsolete algorithm (usually a LCG).

- Never use the built-in generators in the C and C++ languages, especially `rand` and `srand`. These have no standard implementation and are often badly flawed.

If all scientific papers whose results are in doubt because of one or more of the above traps were to disappear from library shelves, there would be a gap on each shelf about as big as your fist.

You may also want to watch for indications that a generator is overengineered, and therefore wasteful of resources:

- Avoid generators that take more than (say) two dozen arithmetic or logical operations to generate a 64-bit integer or double precision floating result.
- Avoid using generators (over-)designed for serious cryptographic use.
- Avoid using generators with period  $> 10^{100}$ . You *really* will never need it, and, above some minimum bound, the period of a generator has little to do with its quality.

Since we have told you what to avoid from the past, we should immediately follow with the received wisdom of the present:

An acceptable random generator must combine at least two (ideally, unrelated) methods. The methods combined should evolve independently and share no state. The combination should be by simple operations that do not produce results less random than their operands.

If you don't want to read the rest of this section, then use the following code to generate all the uniform deviates you'll ever need. This is our suspenders-and-belt, full-body-armor, never-any-doubt generator;\* and, it also meets the above guidelines for avoiding wasteful, overengineered methods. (The fastest generators that we recommend, below, are only  $\sim 2.5 \times$  faster, even when their code is copied inline into an application.)

`ran.h`

```
struct Ran {
```

Implementation of the highest quality recommended generator. The constructor is called with an integer seed and creates an instance of the generator. The member functions `int64`, `doub`, and `int32` return the next values in the random sequence, as a variable type indicated by their names. The period of the generator is  $\approx 3.138 \times 10^{57}$ .

```
    Ullong u,v,w;
    Ran(Ullong j) : v(4101842887655102017LL), w(1) {
        Constructor. Call with any integer seed (except value of v above).
        u = j ^ v; int64();
        v = u; int64();
        w = v; int64();
    }
    inline Ullong int64() {
```

\*“What about the \$1000 reward?” some long-time readers may wonder. That is a tale in itself: Two decades ago, the first edition of *Numerical Recipes* included a flawed random number generator. (Forgive us, we were young!) In the second edition, in a misguided attempt to buy back some credibility, we offered a prize of \$1000 to the “first reader who convinces us” that that edition’s best generator was in any way flawed. No one ever won that prize (`ran2` is a sound generator within its stated limits). We did learn, however, that many people don’t understand what constitutes a statistical proof. Multiple claimants over the years have submitted claims based on one of two fallacies: (1) finding, after much searching, some particular seed that makes the first few random values seem unusual, or (2) finding, after some millions of trials, a statistic that, just that once, is as unlikely as a part in a million. In the interests of our own sanity, we are not offering any rewards in this edition. And the previous offer is hereby revoked.

---

```

Return 64-bit random integer. See text for explanation of method.
    u = u * 2862933555777941757LL + 7046029254386353087LL;
    v ^= v >> 17; v ^= v << 31; v ^= v >> 8;
    w = 4294957665U*(w & 0xffffffff) + (w >> 32);
    Ullong x = u ^ (u << 21); x ^= x >> 35; x ^= x << 4;
    return (x + v) ^ w;
}
inline Doub doub() { return 5.42101086242752217E-20 * int64(); }
Return random double-precision floating value in the range 0. to 1.
inline UInt int32() { return (UInt)int64(); }
Return 32-bit random integer.
};

```

The basic premise here is that a random generator, because it maintains internal state between calls, should be an object, a `struct`. You can declare more than one instance of it (although it is hard to think of a reason for doing so), and different instances will in no way interact.

The constructor `Ran()` takes a single integer argument, which becomes the seed for the sequence generated. Different seeds generate (for all practical purposes) completely different sequences. Once constructed, an instance of `Ran` offers several different formats for random output. To be specific, suppose you have created an instance by the declaration

```
Ran myran(17);
```

where `myran` is now the name of this instance, and 17 is its seed. Then, the function `myran.int64()` returns a random 64-bit unsigned integer; the function `myran.int32()` returns an unsigned 32-bit integer; and the function `myran.doub()` returns a double-precision floating value in the range 0.0 to 1.0. You can intermix calls to these functions as you wish. You can use *any* returned random bits for any purpose. If you need a random integer between 1 and  $n$  (inclusive), say, then the expression  $1 + \text{myran.int64()} \% (n-1)$  is perfectly OK (though there are faster idioms than the use of `%`).

In the rest of this section, we briefly review some history (the rise and fall of the LCG), then give details on some of the algorithmic methods that go into a good generator, and on how to combine those methods. Finally, we will give some further recommended generators, additional to `Ran` above.

### 7.1.1 Some History

With hindsight, it seems clear that the whole field of random number generation was mesmerized, for far too long, by the simple recurrence equation

$$I_{j+1} = aI_j + c \pmod{m} \quad (7.1.1)$$

Here  $m$  is called the *modulus*,  $a$  is a positive integer called the *multiplier*, and  $c$  (which may be zero) is nonnegative integer called the *increment*. For  $c \neq 0$ , equation (7.1.1) is called a linear congruential generator (LCG). When  $c = 0$ , it is sometimes called a multiplicative LCG or MLCG.

The recurrence (7.1.1) must eventually repeat itself, with a period that is obviously no greater than  $m$ . If  $m$ ,  $a$ , and  $c$  are properly chosen, then the period will be of maximal length, i.e., of length  $m$ . In that case, all possible integers between 0 and  $m - 1$  occur at some point, so any initial “seed” choice of  $I_0$  is as good as any other:

The sequence just takes off from that point, and successive values  $I_j$  are the returned “random” values.

The idea of LCGs goes back to the dawn of computing, and they were widely used in the 1950s and thereafter. The trouble in paradise first began to be noticed in the mid-1960s (e.g., [1]): If  $k$  random numbers at a time are used to plot points in  $k$ -dimensional space (with each coordinate between 0 and 1), then the points will not tend to “fill up” the  $k$ -dimensional space, but rather will lie on  $(k - 1)$ -dimensional “planes.” There will be *at most* about  $m^{1/k}$  such planes. If the constants  $m$  and  $a$  are not very carefully chosen, there will be *many fewer than that*. The number  $m$  was usually close to the machine’s largest representable integer, often  $\sim 2^{32}$ . So, for example, the number of planes on which triples of points lie in three-dimensional space can be no greater than about the cube root of  $2^{32}$ , about 1600. You might well be focusing attention on a physical process that occurs in a small fraction of the total volume, so that the discreteness of the planes can be very pronounced.

Even worse, many early generators happened to make particularly bad choices for  $m$  and  $a$ . One infamous such routine, RANDU, with  $a = 65539$  and  $m = 2^{31}$ , was widespread on IBM mainframe computers for many years, and widely copied onto other systems. One of us recalls as a graduate student producing a “random” plot with only 11 planes and being told by his computer center’s programming consultant that he had misused the random number generator: “We guarantee that each number is random individually, but we don’t guarantee that more than one of them is random.” That set back our graduate education by at least a year!

LCGs and MLCGs have additional weaknesses: When  $m$  is chosen as a power of 2 (e.g., RANDU), then the low-order bits generated are hardly random at all. In particular, the least significant bit has a period of at most 2, the second at most 4, the third at most 8, and so on. But, if you don’t choose  $m$  as a power of 2 (in fact, choosing  $m$  prime is generally a good thing), then you generally need access to double-length registers to do the multiplication and modulo functions in equation (7.1.1). These were often unavailable in computers of the time (and usually still are).

A lot of effort subsequently went into “fixing” these weaknesses. An elegant number-theoretical test of  $m$  and  $a$ , the *spectral test*, was developed to characterize the density of planes in arbitrary dimensional space. (See [2] for a recent review that includes graphical renderings of some of the appallingly poor generators that were used historically, and also [3].) *Schrage’s method* [4] was invented to do the multiplication  $a I_j$  with only 32-bit arithmetic for  $m$  as large as  $2^{32} - 1$ , but, unfortunately, only for certain  $a$ ’s, not always the best ones. The review by Park and Miller [5] gives a good contemporary picture of LCGs in their heyday.

Looking back, it seems clear that the field’s long preoccupation with LCGs was somewhat misguided. There is no technological reason that the better, non-LCG, generators of the last decade could not have been discovered decades earlier, nor any reason that the impossible dream of an elegant “single algorithm” generator could not also have been abandoned much earlier (in favor of the more pragmatic patchwork in combined generators). As we will explain below, LCGs and MLCGs can still be useful, but only in carefully controlled situations, and with due attention to their manifest weaknesses.

### 7.1.2 Recommended Methods for Use in Combined Generators

Today, there are at least a dozen plausible algorithms that deserve serious consideration for use in random generators. Our selection of a few is motivated by aesthetics as much as mathematics. We like algorithms with few and fast operations, with foolproof initialization, and with state small enough to keep in registers or first-level cache (if the compiler and hardware are able to do so). This means that we tend to avoid otherwise fine algorithms whose state is an array of some length, despite the relative simplicity with which such algorithms can achieve truly humongous periods. For overviews of broader sets of methods, see [6] and [7].

To be recommendable for use in a combined generator, we require a method to be understood theoretically to some degree, and to pass a reasonably broad suite of empirical tests (or, if it fails, have weaknesses that are well characterized). Our minimal theoretical standard is that the period, the set of returned values, and the set of valid initializations should be completely understood. As a minimal empirical standard, we have used the second release (2003) of Marsaglia's whimsically named Diehard battery of statistical tests [8].\* An alternative test suite, NIST-STS [9], might be used instead, or in addition.

Simply requiring a combined generator to pass Diehard or NIST-STS is not an acceptably stringent test. These suites make only  $\sim 10^7$  calls to the generator, whereas a user program might make  $10^{12}$  or more. Much more meaningful is to require that each method in a combined generator separately pass the chosen suite. Then the combination generator (if correctly constructed) should be vastly better than any one component. In the tables below, we use the symbol “ $*$ ” to indicate that a method passes the Diehard tests by itself. (For 64-bit quantities, the statement is that the 32 high and low bits each pass.) Correspondingly, the words “can be used as random,” below, do not imply perfect randomness, but only a minimum level for quick-and-dirty applications where a better, combined, generator is just not needed.

We turn now to specific methods, starting with methods that use 64-bit unsigned arithmetic (what we call `ULLONG`, that is, `unsigned long long` in the Linux/Unix world, or `unsigned __int64` on planet Microsoft).

**(A) 64-bit Xorshift Method.** This generator was discovered and characterized by Marsaglia [10]. In just three XORs and three shifts (generally fast operations) it produces a full period of  $2^{64} - 1$  on 64 bits. (The missing value is zero, which perpetuates itself and must be avoided.) High and low bits pass Diehard. A generator can use either the three-line update rule, below, that starts with `<<`, or the rule that starts with `>>`. (The two update rules produce different sequences, related by bit reversal.)

|             |   |
|-------------|---|
| state:      | $x$ (unsigned 64-bit)   |
| initialize: | $x \neq 0$  |
| update:     | $x \leftarrow x \wedge (x >> a_1),$<br>$x \leftarrow x \wedge (x << a_2),$<br>$x \leftarrow x \wedge (x >> a_3);$ |
| or          | $x \leftarrow x \wedge (x << a_1),$<br>$x \leftarrow x \wedge (x >> a_2),$  |

---

\*Be sure that you use a version of Diehard that includes the so-called “Gorilla Test.”

|                     |                                      |
|---------------------|--------------------------------------|
|                     | $x \leftarrow x \wedge (x \ll a_3);$ |
| can use as random:  | $x$ (all bits) *                     |
| can use in bit mix: | $x$ (all bits)                       |
| can improve by:     | output 64-bit MLCG successor         |
| period:             | $2^{64} - 1$                         |

Here is a very brief outline of the theory behind these generators: Consider the 64 bits of the integer as components in a vector of length 64, in a linear space where addition and multiplication are done modulo 2. Noting that XOR ( $\wedge$ ) is the same as addition, each of the three lines in the updating can be written as the action of a  $64 \times 64$  matrix on a vector, where the matrix is all zeros except for ones on the diagonal, and on exactly one super- or subdiagonal (corresponding to  $\ll$  or  $\gg$ ). Denote this matrix as  $S_k$ , where  $k$  is the shift argument (positive for left-shift, say, and negative for right-shift). Then, one full step of updating (three lines of the updating rule, above) corresponds to multiplication by the matrix  $T \equiv S_{k_3}S_{k_2}S_{k_1}$ .

One next needs to find triples of integers  $(k_1, k_2, k_3)$ , for example  $(21, -35, 4)$ , that give the full  $M \equiv 2^{64} - 1$  period. Necessary and sufficient conditions are that  $T^M = 1$  (the identity matrix) and that  $T^N \neq 1$  for these seven values of  $N$ :  $M/6700417$ ,  $M/65537$ ,  $M/641$ ,  $M/257$ ,  $M/17$ ,  $M/5$ , and  $M/3$ , that is,  $M$  divided by each of its seven distinct prime factors. The required large powers of  $T$  are readily computed by successive squarings, requiring only on the order of  $64^4$  operations. With this machinery, one can find full-period triples  $(k_1, k_2, k_3)$  by exhaustive search, at a reasonable cost.

Brent [11] has pointed out that the 64-bit xorshift method produces, at each bit position, a sequence of bits that is identical to one produced by a certain linear feedback shift register (LFSR) on 64 bits. (We will learn more about LFSRs in §7.5.) The xorshift method thus potentially has some of the same strengths and weaknesses as an LFSR. Mitigating this, however, is the fact that the primitive polynomial equivalent of a typical xorshift generator has many nonzero terms, giving it better statistical properties than LFSR generators based, for example, on primitive trinomials. In effect, the xorshift generator is a way to step simultaneously 64 nontrivial one-bit LFSR registers, using only six fast, 64-bit operations. There are other ways of making fast steps on LFSRs, and combining the output of more than one such generator [12,13], but none as simple as the xorshift method.

While each bit position in an xorshift generator has the same recurrence, and therefore the same sequence with period  $2^{64} - 1$ , the method guarantees offsets to each sequence such that all nonzero 64-bit words are produced *across* the bit positions during one complete cycle (as we just saw).

A selection of full-period triples is tabulated in [10]. Only a small fraction of full-period triples actually produce generators that pass Diehard. Also, a triple may pass in its  $\ll$ -first version, and fail in its  $\gg$ -first version, or vice versa. Since the two versions produce simply bit-reversed sequences, a failure of either sense must obviously be considered a failure of both (and a weakness in Diehard). The following recommended parameter sets pass Diehard for both the  $\ll$  and  $\gg$  rules. The sets near the top of the list may be slightly superior to the sets near the bottom. The column labeled ID assigns an identification string to each recommended generator that we will refer to later.

| ID | $a_1$ | $a_2$ | $a_3$ |
|----|-------|-------|-------|
| A1 | 21    | 35    | 4     |
| A2 | 20    | 41    | 5     |
| A3 | 17    | 31    | 8     |
| A4 | 11    | 29    | 14    |
| A5 | 14    | 29    | 11    |
| A6 | 30    | 35    | 13    |
| A7 | 21    | 37    | 4     |
| A8 | 21    | 43    | 4     |
| A9 | 23    | 41    | 18    |

It is easy to design a test that the xorshift generator fails if used by itself. Each bit at step  $i + 1$  depends on at most 8 bits of step  $i$ , so some simple logical combinations of the two timesteps (and appropriate masks) will show immediate non-randomness. Also, when the state passes through a value with only small numbers of 1 bits, as it must eventually do (so-called states of *low Hamming weight*), it will take longer than expected to recover. Nevertheless, used in combination, the xorshift generator is an exceptionally powerful and useful method. Much grief could have been avoided had it, instead of LCGs, been discovered in 1949!

**(B) Multiply with Carry (MWC) with Base  $b = 2^{32}$ .** Also discovered by Marsaglia, the *base b* of an MWC generator is most conveniently chosen to be a power of 2 that is half the available word length (i.e.,  $b = 32$  for 64-bit words). The MWC is then defined by its *multiplier a*.

|                     |  |
|---------------------|--|
| state:              | $x$ (unsigned 64-bit)                            |
| initialize:         | $1 \leq x \leq 2^{32} - 1$                       |
| update:             | $x \leftarrow a(x \& [2^{32} - 1]) + (x \gg 32)$ |
| can use as random:  | $x$ (low 32 bits) *                              |
| can use in bit mix: | $x$ (all 64 bits)                                |
| can improve by:     | output 64-bit xorshift successor to 64 bit $x$   |
| period:             | $(2^{32}a - 2)/2$ (a prime)                      |

An MWC generator with parameters  $b$  and  $a$  is related theoretically [14] to, though not identical to, an LCG with modulus  $m = ab - 1$  and multiplier  $a$ . It is easy to find values of  $a$  that make  $m$  a prime, so we get, in effect, the benefit of a prime modulus using only power-of-two modular arithmetic. It is not possible to choose  $a$  to give the maximal period  $m$ , but if  $a$  is chosen to make both  $m$  and  $(m - 1)/2$  prime, then the period of the MCG is  $(m - 1)/2$ , almost as good. A fraction of candidate  $a$ 's thus chosen passes the standard statistical test suites; a spectral test [14] is a promising development, but we have not made use of it here.

Although only the low  $b$  bits of the state  $x$  can be taken as algorithmically random, there is considerable randomness in all the bits of  $x$  that represent the product  $ab$ . This is very convenient in a combined generator, allowing the entire state  $x$  to be used as a component. In fact, the first two recommended  $a$ 's below give  $ab$  so close to  $2^{64}$  (within about 2 ppm) that the high bits of  $x$  actually pass Diehard. (This is a good example of how any test suite can fail to find small amounts of highly nonrandom behavior, in this case as many as 8000 missing values in the top 32 bits.)

Apart from this kind of consideration, the values below are recommended with no particular ordering.

| ID | $a$        |
|----|------------|
| B1 | 4294957665 |
| B2 | 4294963023 |
| B3 | 4162943475 |
| B4 | 3947008974 |
| B5 | 3874257210 |
| B6 | 2936881968 |
| B7 | 2811536238 |
| B8 | 2654432763 |
| B9 | 1640531364 |

**(C) LCG Modulo  $2^{64}$ .** Why in the world do we include this generator after vilifying it so thoroughly above? For the parameters given (which strongly pass the spectral test), its high 32 bits almost, but don't quite, pass Diehard, and its low 32 bits are a complete disaster. Yet, as we will see when we discuss the construction of combined generators, there is still a niche for it to fill. The recommended multipliers  $a$  below have good spectral characteristics [15].

|                     |                                     |
|---------------------|-------------------------------------|
| state:              | $x$ (unsigned 64-bit)               |
| initialize:         | any value                           |
| update:             | $x \leftarrow ax + c \pmod{2^{64}}$ |
| can use as random:  | $x$ (high 32 bits, with caution)    |
| can use in bit mix: | $x$ (high 32 bits)                  |
| can improve by:     | output 64-bit xorshift successor    |
| period:             | $2^{64}$                            |

| ID | $a$                 | $c$ (any odd value ok) |
|----|---------------------|------------------------|
| C1 | 3935559000370003845 | 2691343689449507681    |
| C2 | 3202034522624059733 | 4354685564936845319    |
| C3 | 2862933555777941757 | 7046029254386353087    |

**(D) MLCG Modulo  $2^{64}$ .** As for the preceding one, the useful role for this generator is strictly limited. The low bits are highly nonrandom. The recommended multipliers have good spectral characteristics (some from [15]).

|                     |                                  |
|---------------------|----------------------------------|
| state:              | $x$ (unsigned 64-bit)            |
| initialize:         | $x \neq 0$                       |
| update:             | $x \leftarrow ax \pmod{2^{64}}$  |
| can use as random:  | $x$ (high 32 bits, with caution) |
| can use in bit mix: | $x$ (high 32 bits)               |
| can improve by:     | output 64-bit xorshift successor |
| period:             | $2^{62}$                         |

| ID | $a$                 |
|----|---------------------|
| D1 | 2685821657736338717 |
| D2 | 7664345821815920749 |
| D3 | 4768777513237032717 |
| D4 | 1181783497276652981 |
| D5 | 702098784532940405  |

**(E) MLCG with  $m \gg 2^{32}$ ,  $m$  Prime.** When 64-bit unsigned arithmetic is available, the MLCGs with prime moduli and large multipliers of good spectral character are decent 32-bit generators. Their main liability is that the 64-bit multiply and 64-bit remainder operations are quite expensive for the mere 32 (or so) bits of the result.

|                     |   |
|---------------------|---|
| state:              | $x$ (unsigned 64-bit)                               |
| initialize:         | $1 \leq x \leq m - 1$                               |
| update:             | $x \leftarrow ax \pmod{m}$                          |
| can use as random:  | $x \quad (1 \leq x \leq m - 1)$ or low 32 bits    * |
| can use in bit mix: | (same)  |
| period:             | $m - 1$   |

The parameter values below were kindly computed for us by P. L'Ecuyer. The multipliers are about the best that can be obtained for the prime moduli, close to powers of 2, shown. Although the recommended use is for only the low 32 bits (which all pass Diehard), you can see that (depending on the modulus) as many as 43 reasonably good bits can be obtained for the cost of the 64-bit multiply and remainder operations.

| ID  | $m$                           | $a$                              |
|-----|-------------------------------|----------------------------------|
| E1  | $2^{39} - 7 = 549755813881$   | 10014146<br>30508823<br>25708129 |
| E4  | $2^{41} - 21 = 2199023255531$ | 5183781<br>1070739<br>6639568    |
| E7  | $2^{42} - 11 = 4398046511093$ | 1781978<br>2114307<br>1542852    |
| E10 | $2^{43} - 57 = 8796093022151$ | 2096259<br>2052163<br>2006881    |

**(F) MLCG with  $m \gg 2^{32}$ ,  $m$  Prime, and  $a(m - 1) \approx 2^{64}$ .** A variant, for use in combined generators, is to choose  $m$  and  $a$  to make  $a(m - 1)$  as close as possible to  $2^{64}$ , while still requiring that  $m$  be prime and that  $a$  pass the spectral test. The purpose of this maneuver is to make  $ax$  a 64-bit value with good randomness in its high bits, for use in combined generators. The expense of the multiply and remainder operations is still the big liability, however. The low 32 bits of  $x$  are not significantly less random than those of the previous MLCG generators E1–E12.

|                     |  |
|---------------------|--|
| state:              | $x$ (unsigned 64-bit)                          |
| initialize:         | $1 \leq x \leq m - 1$                          |
| update:             | $x \leftarrow ax \pmod{m}$                     |
| can use as random:  | $x$ ( $1 \leq x \leq m - 1$ ) or low 32 bits * |
| can use in bit mix: | $ax$ (but don't use both $ax$ and $x$ ) *      |
| can improve by:     | output 64-bit xorshift successor of $ax$       |
| period:             | $m - 1$  |

| ID | $m$                                       | $a$     |
|----|---|---------|
| F1 | $1148 \times 2^{32} + 11 = 4930622455819$ | 3741260 |
| F2 | $1264 \times 2^{32} + 9 = 5428838662153$  | 3397916 |
| F3 | $2039 \times 2^{32} + 3 = 8757438316547$  | 2106408 |

### 7.1.3 How to Construct Combined Generators

While the construction of combined generators is an art, it should be informed by underlying mathematics. Rigorous theorems about combined generators are usually possible only when the generators being combined are algorithmically related; but that in itself is usually a bad thing to do, on the general principle of “don’t put all your eggs in one basket.” So, one is left with guidelines and rules of thumb.

The methods being combined should be independent of one another. They must share no state (although their initializations are allowed to derive from some convenient common seed). They should have different, incommensurate, periods. And, ideally, they should “look like” each other algorithmically as little as possible. This latter criterion is where some art necessarily enters.

The output of the combination generator should in no way perturb the independent evolution of the individual methods, nor should the operations effecting combination have any side effects.

The methods should be combined by binary operations whose output is no less random than one input if the other input is held fixed. For 32- or 64-bit unsigned arithmetic, this in practice means that only the  $+$  and  $\wedge$  operators can be used. As an example of a forbidden operator, consider multiplication: If one operand is a power of 2, then the product will end in trailing zeros, no matter how random is the other operand.

All bit positions in the combined output should depend on high-quality bits from at least two methods, and may also depend on lower-quality bits from additional methods. In the tables above, the bits labeled “can use as random” are considered high quality; those labeled “can use in bit mix” are considered low quality, unless they also pass a statistical suite such as Diehard.

There is one further trick at our disposal, the idea of using a method as a *successor relation* instead of as a generator in its own right. Each of the methods described above is a mapping from some 64-bit state  $x_i$  to a unique successor state  $x_{i+1}$ . For a method to pass a good statistical test suite, it must have no detectable correlations between a state and its successor. If, in addition, the method has period  $2^{64}$  or  $2^{64} - 1$ , then all values (except possibly zero) occur exactly once as successor states.

Suppose we take the output of a generator, say C1 above, with period  $2^{64}$ , and run it through generator A6, whose period is  $2^{64} - 1$ , as a successor relation. This is conveniently denoted by “A6(C1),” which we will call a *composed* generator. Note that the composed output is emphatically *not* fed back into the state of C1, which

continues unperturbed. The composed generator A6(C1) has the period of C1, not, unfortunately, the product of the two periods. But its random mapping of C1’s output values effectively fixes C1’s problems with short-period low bits. (The better so if the form of A6 with left-shift first is used.) And, A6(C1) will also fix A6’s weakness that a bit depends only on a few bits of the previous state. We will thus consider a carefully constructed composed generator as being a combined generator, on a par with direct combining via  $+$  or  $\wedge$ .

Composition is inferior to direct combining in that it costs almost as much but does not increase the size of the state or the length of the period. It is superior to direct combining in its ability to mix widely differing bit positions. In the previous example we would not have accepted A6+C1 as a combined generator, because the low bits of C1 are so poor as to add little value to the combination; but A6(C1) has no such liability, and much to recommend it. In the preceding summary tables of each method, we have indicated recommended combinations for composed generators in the table entries, “can improve by.”

We can now completely describe the generator in Ran, above, by the pseudo-equation,

$$\text{Ran} = [\text{A1}_l(\text{C3}) + \text{A3}_r] \wedge \text{B1} \quad (7.1.2)$$

that is, the combination and/or composition of four different generators. For the methods A1 and A3, the subscripts  $l$  and  $r$  denote whether a left- or right-shift operation is done first. The period of Ran is the least common multiple of the periods of C3, A3, and B1.

The simplest and fastest generator that we can readily recommend is

$$\text{Ranq1} \equiv \text{D1}(\text{A1}_r) \quad (7.1.3)$$

implemented as

```
struct Ranq1 {
    Recommended generator for everyday use. The period is ≈ 1.8 × 1019. Calling conventions
    same as Ran, above.
    Ullong v;
    Ranq1(Ullong j) : v(4101842887655102017LL) {
        v ^= j;
        v = int64();
    }
    inline Ullong int64() {
        v ^= v >> 21; v ^= v << 35; v ^= v >> 4;
        return v * 2685821657736338717LL;
    }
    inline Doub doub() { return 5.42101086242752217E-20 * int64(); }
    inline UInt int32() { return (UInt)int64(); }
};
```

ran.h

Ranq1 generates a 64-bit random integer in 3 shifts, 3 xors, and one multiply, or a double floating value in one additional multiply. Its method is concise enough to go easily inline in an application. It has a period of “only”  $1.8 \times 10^{19}$ , so it should not be used by an application that makes more than  $\sim 10^{12}$  calls. With that restriction, we think that Ranq1 will do just fine for 99.99% of all user applications, and that Ran can be reserved for the remaining 0.01%.

If the “short” period of Ranq1 bothers you (which it shouldn’t), you can instead use

$$\text{Ranq2} \equiv \text{A3}_r \wedge \text{B1} \quad (7.1.4)$$

whose period is  $8.5 \times 10^{37}$ .

```
ran.h struct Ranq2 {
    Backup generator if Ranq1 has too short a period and Ran is too slow. The period is ≈ 8.5 ×
    1037. Calling conventions same as Ran, above.
    Ullong v,w;
    Ranq2(Ullong j) : v(4101842887655102017LL), w(1) {
        v ^= j;
        w = int64();
        v = int64();
    }
    inline Ullong int64() {
        v ^= v >> 17; v ^= v << 31; v ^= v >> 8;
        w = 4294957665U*(w & 0xffffffff) + (w >> 32);
        return v ^ w;
    }
    inline Doub doub() { return 5.42101086242752217E-20 * int64(); }
    inline Uint int32() { return (Uint)int64(); }
};
```

### 7.1.4 Random Hashes and Random Bytes

Every once in a while, you want a random sequence  $H_i$  whose values you can visit or revisit in any order of  $i$ 's. That is to say, you want a *random hash* of the integers  $i$ , one that passes serious tests for randomness, even for very ordered sequences of  $i$ 's. In the language already developed, you want a generator that has no state at all and is built entirely of successor relationships, starting with the value  $i$ .

An example that easily passes the Diehard test is

$$\text{Ranhash} \equiv A2_l(D3(A7_r(C1(i)))) \quad (7.1.5)$$

Note the alternation between successor relations that utilize 64-bit multiplication and ones using shifts and XORs.

```
ran.h struct Ranhash {
    High-quality random hash of an integer into several numeric types.
    inline Ullong int64(Ullong u) {
        Returns hash of u as a 64-bit integer.
        Ullong v = u * 3935559000370003845LL + 2691343689449507681LL;
        v ^= v >> 21; v ^= v << 37; v ^= v >> 4;
        v *= 4768777513237032717LL;
        v ^= v << 20; v ^= v >> 41; v ^= v << 5;
        return v;
    }
    inline Uint int32(Ullong u)
        Returns hash of u as a 32-bit integer.
        { return (Uint)(int64(u) & 0xffffffff); }
    inline Doub doub(Ullong u)
        Returns hash of u as a double-precision floating value between 0. and 1.
        { return 5.42101086242752217E-20 * int64(u); }
};
```

Since Ranhash has no state, it has no constructor. You just call its  $\text{int64}(i)$  function, or any of its other functions, with your value of  $i$  whenever you want.

**Random Bytes.** In a different set of circumstances, you may want to generate random integers a byte at a time. You can of course pull bytes out of any of the above

recommended combination generators, since they are constructed to be equally good on all bits. The following code, added to any of the generators above, augments them with an `int8()` method. (Be sure to initialize `bc` to zero in the constructor.)

```
Ullong breg;
Int bc;
inline unsigned char int8() {
    if (bc--) return (unsigned char)(breg >>= 8);
    breg = int64();
    bc = 7;
    return (unsigned char)breg;
}
```

If you want a more byte-oriented, though not necessarily faster, algorithm, an interesting one — in part because of its interesting history — is Rivest's RC4, used in many Internet applications. RC4 was originally a proprietary algorithm of RSA, Inc., but it was protected simply as a trade secret and not by either patent or copyright. The result was that when the secret was breached, by an anonymous posting to the Internet in 1994, RC4 became, in almost all respects, public property. The name RC4 is still protectable, and is a trademark of RSA. So, to be scrupulous, we give the following implementation another name, Ranbyte.

```
struct Ranbyte {
    Generator for random bytes using the algorithm generally known as RC4. ran.h
    Int s[256], i, j, ss;
    Uint v;
    Ranbyte(Int u) {
        Constructor. Call with any integer seed.
        v = 2244614371U ^ u;
        for (i=0; i<256; i++) {s[i] = i;}
        for (j=0, i=0; i<256; i++) {
            ss = s[i];
            j = (j + ss + (v >> 24)) & 0xff;
            s[i] = s[j]; s[j] = ss;
            v = (v << 24) | (v >> 8);
        }
        i = j = 0;
        for (Int k=0; k<256; k++) int8();
    }
    inline unsigned char int8() {
        Returns next random byte in the sequence.
        i = (i+1) & 0xff;
        ss = s[i];
        j = (j+ss) & 0xff;
        s[i] = s[j]; s[j] = ss;
        return (unsigned char)(s[(s[i]+s[j]) & 0xff]);
    }
    Uint int32() {
        Returns a random 32-bit integer constructed from 4 random bytes. Slow!
        v = 0;
        for (int k=0; k<4; k++) {
            i = (i+1) & 0xff;
            ss = s[i];
            j = (j+ss) & 0xff;
            s[i] = s[j]; s[j] = ss;
            v = (v << 8) | s[(s[i]+s[j]) & 0xff];
        }
        return v;
    }
}
```

```

Doub doub() {
    Returns a random double-precision floating value between 0. and 1. Slow!!
    return 2.32830643653869629E-10 * (int32() +
        2.32830643653869629E-10 * int32() );
}
};

```

Notice that there is a lot of overhead in starting up an instance of `Ranbyte`, so you should not create instances inside loops that are executed many times. The methods that return 32-bit integers, or double floating-point values, are *slow* in comparison to the other generators above, but are provided in case you want to use `Ranbyte` as a test substitute for another, perhaps questionable, generator.

If you find any nonrandomness at all in `Ranbyte`, don't tell us. But there are several national cryptological agencies that might, or might not, want to talk to you!

### 7.1.5 Faster Floating-Point Values

The steps above that convert a 64-bit integer to a double-precision floating-point value involves both a nontrivial type conversion and a 64-bit floating multiply. They are performance bottlenecks. One can instead directly move the random bits into the right place in the double word with union structure, a mask, and some 64-bit logical operations; but in our experience this is not significantly faster.

To generate faster floating-point values, if that is an absolute requirement, we need to bend some of our design rules. Here is a variant of “Knuth’s subtractive generator,” which is a so-called *lagged Fibonacci generator* on a circular list of 55 values, with lags 24 and 55. Its interesting feature is that new values are generated directly as floating point, by the floating-point subtraction of two previous values.

```

ran.h struct Ranfib {
    Doub dtab[55], dd;
    Int inext, inextp;
    Ranfib(ULLong j) : inext(0), inextp(31) {
        Constructor. Call with any integer seed. Uses Ranq1 to initialize.
        Ranq1 init(j);
        for (int k=0; k<55; k++) dtab[k] = init.doub();
    }
    Doub doub() {
        Returns random double-precision floating value between 0. and 1.
        if (++inext == 55) inext = 0;
        if (++inextp == 55) inextp = 0;
        dd = dtab[inext] - dtab[inextp];
        if (dd < 0) dd += 1.0;
        return (dtab[inext] = dd);
    }
    inline unsigned long int32()
        Returns random 32-bit integer. Recommended only for testing purposes.
        { return (unsigned long)(doub() * 4294967295.0);}
};

```

The `int32` method is included merely for testing, or incidental use. Note also that we use `Ranq1` to initialize `Ranfib`’s table of 55 random values. See earlier editions of Knuth or *Numerical Recipes* for a (somewhat awkward) way to do the initialization purely internally.

`Ranfib` fails the Diehard “birthday test,” which is able to discern the simple relation among the three values at lags 0, 24, and 55. Aside from that, it is a good,

but not great, generator, with speed as its principal recommendation.

### 7.1.6 Timing Results

Timings depend so intimately on highly specific hardware and compiler details, that it is hard to know whether a single set of tests is of any use at all. This is especially true of combined generators, because a good compiler, or a CPU with sophisticated instruction look-ahead, can interleave and pipeline the operations of the individual methods, up to the final combination operations. Also, as we write, desktop computers are in transition from 32 bits to 64, which will affect the timing of 64-bit operations. So, you ought to familiarize yourself with C’s “`clock_t clock(void)`” facility and run your own experiments.

That said, the following tables give typical results for routines in this section, normalized to a 3.4 GHz Pentium CPU, vintage 2004. The units are  $10^6$  returned values per second. Large numbers are better.

| Generator | <code>int64()</code> | <code>doub()</code> | <code>int8()</code> |
|-----------|----------------------|---------------------|---------------------|
| Ran       | 19                   | 10                  | 51                  |
| Ranq1     | 39                   | 13                  | 59                  |
| Ranq2     | 32                   | 12                  | 58                  |
| Ranfib    |                      | 24                  |                     |
| Ranbyte   |                      |                     | 43                  |

The `int8()` timings for Ran, Ranq1, and Ranq2 refer to versions augmented as indicated above.

### 7.1.7 When You Have Only 32-Bit Arithmetic

Our best advice is: Get a better compiler! But if you seriously must live in a world with only unsigned 32-bit arithmetic, then here are some options. None of these individually pass Diehard.

#### (G) 32-Bit Xorshift RNG

|                     |   |
|---------------------|---|
| state:              | $x$ (unsigned 32-bit)   |
| initialize:         | $x \neq 0$  |
| update:             | $x \leftarrow x \wedge (x >> b_1),$<br>$x \leftarrow x \wedge (x << b_2),$<br>$x \leftarrow x \wedge (x >> b_3);$ |
| or                  | $x \leftarrow x \wedge (x << b_1),$<br>$x \leftarrow x \wedge (x >> b_2),$<br>$x \leftarrow x \wedge (x << b_3);$ |
| can use as random:  | $x$ (32 bits, with caution)   |
| can use in bit mix: | $x$ (32 bits)   |
| can improve by:     | output 32-bit MLCG successor  |
| period:             | $2^{32} - 1$  |

| ID | $b_1$ | $b_2$ | $b_3$ |
|----|-------|-------|-------|
| G1 | 13    | 17    | 5     |
| G2 | 7     | 13    | 3     |
| G3 | 9     | 17    | 6     |
| G4 | 6     | 13    | 5     |
| G5 | 9     | 21    | 2     |
| G6 | 17    | 15    | 5     |
| G7 | 3     | 13    | 7     |
| G8 | 5     | 13    | 6     |
| G9 | 12    | 21    | 5     |

**(H) MWC with Base  $b = 2^{16}$** 

|                     |  |
|---------------------|--|
| state:              | $x, y$ (unsigned 32-bit)   |
| initialize:         | $1 \leq x, y \leq 2^{16} - 1$  |
| update:             | $x \leftarrow a(x \& [2^{16} - 1]) + (x \gg 16)$<br>$y \leftarrow b(y \& [2^{16} - 1]) + (y \gg 16)$ |
| can use as random:  | $(x \ll 16) + y$   |
| can use in bit mix: | same, or (with caution) $x$ or $y$   |
| can improve by:     | output 32-bit xorshift successor   |
| period:             | $(2^{16}a - 2)(2^{16}b - 2)/4$ (product of two primes)   |

| ID | $a$   | $b$   |
|----|-------|-------|
| H1 | 62904 | 41874 |
| H2 | 64545 | 34653 |
| H3 | 34653 | 64545 |
| H4 | 57780 | 55809 |
| H5 | 48393 | 57225 |
| H6 | 63273 | 33378 |

**(I) LCG Modulo  $2^{32}$** 

|                     |                                     |
|---------------------|-------------------------------------|
| state:              | $x$ (unsigned 32-bit)               |
| initialize:         | any value                           |
| update:             | $x \leftarrow ax + c \pmod{2^{32}}$ |
| can use as random:  | not recommended                     |
| can use in bit mix: | not recommended                     |
| can improve by:     | output 32-bit xorshift successor    |
| period:             | $2^{32}$                            |

| ID | $a$        | $c$ (any odd ok) |
|----|------------|------------------|
| I1 | 1372383749 | 1289706101       |
| I2 | 2891336453 | 1640531513       |
| I3 | 2024337845 | 797082193        |
| I4 | 32310901   | 626627237        |
| I5 | 29943829   | 1013904223       |

(J) MLCG Modulo  $2^{32}$ 

|                     |                                  |
|---------------------|----------------------------------|
| state:              | $x$ (unsigned 32-bit)            |
| initialize:         | $x \neq 0$                       |
| update:             | $x \leftarrow ax \pmod{2^{32}}$  |
| can use as random:  | not recommended                  |
| can use in bit mix: | not recommended                  |
| can improve by:     | output 32-bit xorshift successor |
| period:             | $2^{30}$                         |

| ID | $a$        |
|----|------------|
| J1 | 1597334677 |
| J2 | 741103597  |
| J3 | 1914874293 |
| J4 | 990303917  |
| J5 | 747796405  |

A high-quality, if somewhat slow, combined generator is

$$\text{Ranlim32} \equiv [\text{G3}_l(\text{I2}) + \text{G1}_r] \wedge [\text{G6}_l(\text{H6}_b) + \text{H5}_b] \quad (7.1.6)$$

implemented as

```
struct Ranlim32 {
    High-quality random generator using only 32-bit arithmetic. Same conventions as Ran. Period
    ≈  $3.11 \times 10^{37}$ . Recommended only when 64-bit arithmetic is not available.
    Uint u,v,w1,w2;
    Ranlim32(Uint j) : v(2244614371U), w1(521288629U), w2(362436069U) {
        u = j ^ v; int32();
        v = u; int32();
    }
    inline Uint int32() {
        u = u * 2891336453U + 1640531513U;
        v ^= v >> 13; v ^= v << 17; v ^= v >> 5;
        w1 = 33378 * (w1 & 0xffff) + (w1 >> 16);
        w2 = 57225 * (w2 & 0xffff) + (w2 >> 16);
        Uint x = u ^ (u << 9); x ^= x >> 17; x ^= x << 6;
        Uint y = w1 ^ (w1 << 17); y ^= y >> 15; y ^= y << 5;
        return (x + v) ^ (y + w2);
    }
    inline Doub doub() { return 2.32830643653869629E-10 * int32(); }
    inline Doub truedoub() {
        return 2.32830643653869629E-10 * (int32() +
        2.32830643653869629E-10 * int32());
    }
};
```

ran.h

Note that the `doub()` method returns floating-point numbers with only 32 bits of precision. For full precision, use the slower `truedoub()` method.

**CITED REFERENCES AND FURTHER READING:**

- Gentle, J.E. 2003, *Random Number Generation and Monte Carlo Methods*, 2nd ed. (New York: Springer), Chapter 1.
- Marsaglia, G 1968, "Random Numbers Fall Mainly in the Planes", *Proceedings of the National Academy of Sciences*, vol. 61, pp. 25–28.[1]

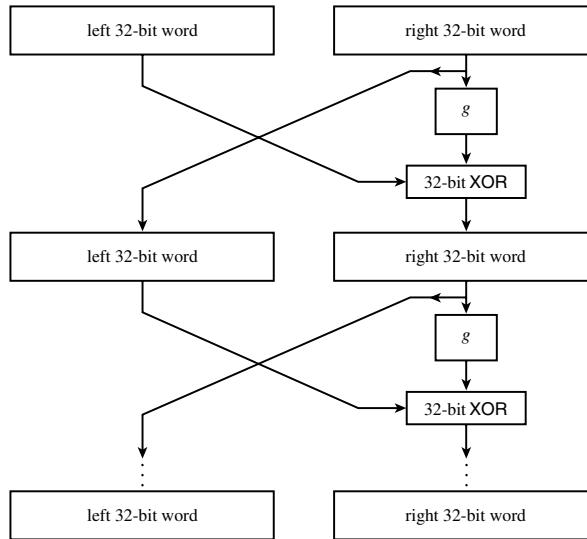
- Entacher, K. 1997, "A Collection of Selected Pseudorandom Number Generators with Linear Structures", Technical Report No. 97, Austrian Center for Parallel Computation, University of Vienna. [Available on the Web at multiple sites.].[2]
- Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley).[3]
- Schrage, L. 1979, "A More Portable Fortran Random Number Generator," *ACM Transactions on Mathematical Software*, vol. 5, pp. 132–138.[4]
- Park, S.K., and Miller, K.W. 1988, "Random Number Generators: Good Ones Are Hard to Find," *Communications of the ACM*, vol. 31, pp. 1192–1201.[5]
- L'Ecuyer, P. 1997 "Uniform Random Number Generators: A Review," *Proceedings of the 1997 Winter Simulation Conference*, Andradóttir, S. et al., eds. (Piscataway, NJ: IEEE).[6]
- Marsaglia, G. 1999, "Random Numbers for C: End, at Last?", posted 1999 January 20 to sci.stat.math.[7]
- Marsaglia, G. 2003, "Diehard Battery of Tests of Randomness v0.2 beta," 2007+ at [http://www.cs.hku.hk/~diehard/.\[8\]](http://www.cs.hku.hk/~diehard/.[8])
- Rukhin, A. et al. 2001, "A Statistical Test Suite for Random and Pseudorandom Number Generators", NIST Special Publication 800-22 (revised to May 15, 2001).[9]
- Marsaglia, G. 2003, "Xorshift RNGs", *Journal of Statistical Software*, vol. 8, no. 14, pp. 1-6.[10]
- Brent, R.P. 2004, "Note on Marsaglia's Xorshift Random Number Generators", *Journal of Statistical Software*, vol. 11, no. 5, pp. 1-5.[11]
- L'Ecuyer, P. 1996, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics of Computation*, vol. 65, pp. 203-213.[12]
- L'Ecuyer, P. 1999, "Tables of Maximally Equidistributed Combined LSFR Generators," *Mathematics of Computation*, vol. 68, pp. 261-269.[13]
- Couture, R. and L'Ecuyer, P. 1997, "Distribution Properties of Multiply-with-Carry Random Number Generators," *Mathematics of Computation*, vol. 66, pp. 591-607.[14]
- L'Ecuyer, P. 1999, "Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure", *Mathematics of Computation*, vol. 68, pp. 249-260.[15]

## 7.2 Completely Hashing a Large Array

We introduced the idea of a random hash or *hash function* in §7.1.4. Once in a while we might want a hash function that operates not on a single word, but on an entire array of length  $M$ . Being perfectionists, we want every single bit in the hashed output array to depend on every single bit in the given input array. One way to achieve this is to borrow structural concepts from algorithms as unrelated as the Data Encryption Standard (DES) and the Fast Fourier Transform (FFT).

DES, like its progenitor cryptographic system LUCIFER, is a so-called “block product cipher” [1]. It acts on 64 bits of input by iteratively applying (16 times, in fact) a kind of highly nonlinear bit-mixing function. Figure 7.2.1 shows the flow of information in DES during this mixing. The function  $g$ , which takes 32 bits into 32 bits, is called the “cipher function.” Meyer and Matyas [1] discuss the importance of the cipher function being nonlinear, as well as other design criteria.

DES constructs its cipher function  $g$  from an intricate set of bit permutations and table lookups acting on short sequences of consecutive bits. For our purposes, a different function  $g$  that can be rapidly computed in a high-level computer language is preferable. Such a function probably weakens the algorithm cryptographically. Our purposes are not, however, cryptographic: We want to find the fastest  $g$ , and the smallest number of iterations of the mixing procedure in Figure 7.2.1, such that our output random sequence passes the tests that are customarily applied to random number generators. The resulting algorithm is not DES, but rather a kind of “pseudo-DES,” better suited to the purpose at hand.



**Figure 7.2.1.** The Data Encryption Standard (DES) iterates a nonlinear function  $g$  on two 32-bit words, in the manner shown here (after Meyer and Matyas [1]).

Following the criterion mentioned above, that  $g$  should be nonlinear, we must give the integer multiply operation a prominent place in  $g$ . Confining ourselves to multiplying 16-bit operands into a 32-bit result, the general idea of  $g$  is to calculate the three distinct 32-bit products of the high and low 16-bit input half-words, and then to combine these, and perhaps additional fixed constants, by fast operations (e.g., add or exclusive-or) into a single 32-bit result.

There are only a limited number of ways of effecting this general scheme, allowing systematic exploration of the alternatives. Experimentation and tests of the randomness of the output lead to the sequence of operations shown in Figure 7.2.2. The few new elements in the figure need explanation: The values  $C_1$  and  $C_2$  are fixed constants, chosen randomly with the constraint that they have exactly 16 1-bits and 16 0-bits; combining these constants via exclusive-or ensures that the overall  $g$  has no bias toward 0- or 1-bits. The “reverse half-words” operation in Figure 7.2.2 turns out to be essential; otherwise, the very lowest and very highest bits are not properly mixed by the three multiplications.

It remains to specify the smallest number of iterations  $N_{it}$  that we can get away with. For purposes of this section, we recommend  $N_{it} = 2$ . We have not found any statistical deviation from randomness in sequences of up to  $10^9$  random deviates derived from this scheme. However, we include  $C_1$  and  $C_2$  constants for  $N_{it} \leq 4$ .

```

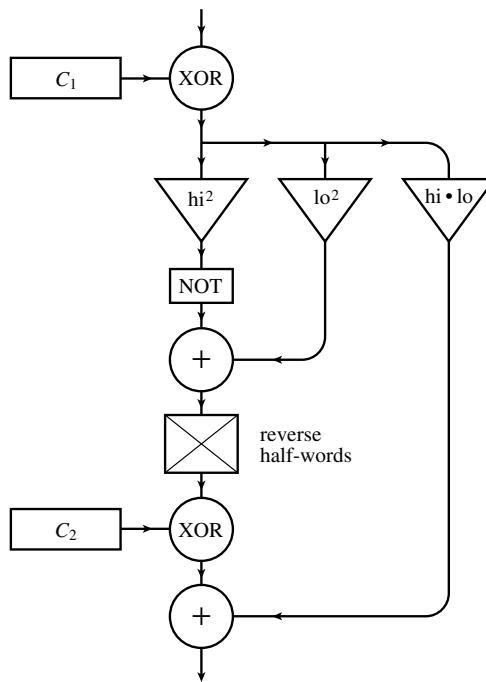
void psdes(UInt &lword, UInt &rword) {
    Pseudo-DES hashing of the 64-bit word (lword,rword). Both 32-bit arguments are returned
    hashed on all bits.

    const int NITER=2;
    static const UInt c1[4]={
        0xbaa96887L, 0x1e17d32cL, 0x03bcdcc3cL, 0x0f33d1b2L};
    static const UInt c2[4]={
        0x4bf3b58L, 0xe874f0c3L, 0x6955c5a6L, 0x55a7ca46L};
    UInt ia,ia(ib,iswap,itmph=0,itmpl=0;
    for (i=0;i<NITER;i++) {
        Perform niter iterations of DES logic, using a simpler (noncryptographic) nonlinear func-
        tion instead of DES's.
        ia = (iswap=rword) ^ c1[i];
        itmpl = ia & 0xffff;
        itmph = ia >> 16;

```

hashall.h

The bit-rich constants  $c1$  and (below)
 $c2$  guarantee lots of nonlinear mix-
ing.



**Figure 7.2.2.** The nonlinear function  $g$  used by the routine psdes.

```

ib=itmpl*itmpl+ ~(itmph*itmph);
rword = lword ^ (((ia = (ib >> 16) |
((ib & 0xffff) << 16)) ^ c2[i])+itmpl*itmph);
lword = iswap;
}
}

```

Thus far, this doesn't seem to have much to do with completely hashing a large array. However, psdes gives us a building block, a routine for mutually hashing two arbitrary 32-bit integers. We now turn to the FFT concept of the *butterfly* to extend the hash to a whole array.

The butterfly is a particular algorithmic construct that applies to an array of length  $N$ , a power of 2. It brings every element into mutual communication with every other element in about  $N \log_2 N$  operations. A useful metaphor is to imagine that one array element has a disease that infects any other element with which it has contact. Then the butterfly has two properties of interest here: (i) After its  $\log_2 N$  stages, everyone has the disease. Furthermore, (ii) after  $j$  stages,  $2^j$  elements are infected; there is never an “eye of the needle” or “necking down” of the communication path.

The butterfly is very simple to describe: In the first stage, every element in the first half of the array mutually communicates with its corresponding element in the second half of the array. Now recursively do this same thing to each of the halves, and so on. We can see by induction that every element now has a communication path to every other one: Obviously it works when  $N = 2$ . And if it works for  $N$ , it must also work for  $2N$ , because the first step gives every element a communication path into both its own and the other half of the array, after which it has, by assumption, a path everywhere.

We need to modify the butterfly slightly, so that our array size  $M$  does not have to be a power of 2. Let  $N$  be the next larger power of 2. We do the butterfly on the (virtual) size  $N$ , ignoring any communication with nonexistent elements larger than  $M$ . This, by itself, doesn't do the job, because the later elements in the first  $N/2$  were not able to “infect” the second  $N/2$  (and similarly at later recursive levels). However, if we do one extra communication between

elements of the first  $N/2$  and second  $N/2$  at the very end, then all missing communication paths are restored by traveling through the first  $N/2$  elements.

The third line in the following code is an idiom that sets  $n$  to the next larger power of 2 greater or equal to  $m$ , a miniature masterpiece due to S.E. Anderson [2]. If you look closely, you'll see that it is itself a sort of butterfly, but now on bits!

```
void hashall(VecUInt &arr) {
    Int m=arr.size(), n=m-1;
    n|=n>>1; n|=n>>2; n|=n>>4; n|=n>>8; n|=n>>16; n++;
    Incredibly, n is now the next power of 2 ≥ m.
    Int nb=n, nb2=n>>1, j, jb;
    if (n<2) throw("size must be > 1");
    while (nb > 1) {
        for (jb=0; jb<n-nb+1; jb+=nb)
            for (j=0; j<nb2; j++)
                if (jb+j+nb2 < m) psdes(arr[jb+j], arr[jb+j+nb2]);
        nb = nb2;
        nb2 >>= 1;
    }
    nb2 = n>>1;
    if (m != n) for (j=nb2; j<m; j++) psdes(arr[j], arr[j-nb2]);
    Final mix needed only if m is not a power of 2.
}
```

hashall.h

#### CITED REFERENCES AND FURTHER READING:

- Meyer, C.H. and Matyas, S.M. 1982, *Cryptography: A New Dimension in Computer Data Security* (New York: Wiley). [1]
- Zonst, A.E. 2000, *Understanding the FFT*, 2nd revised ed. (Titusville, FL: Citrus Press).
- Anderson, S.E. 2005, "Bit Twiddling Hacks," 2007+ at <http://graphics.stanford.edu/~seander/bithacks.html>. [2]
- Data Encryption Standard*, 1977 January 15, Federal Information Processing Standards Publication, number 46 (Washington: U.S. Department of Commerce, National Bureau of Standards).
- Guidelines for Implementing and Using the NBS Data Encryption Standard*, 1981 April 1, Federal Information Processing Standards Publication, number 74 (Washington: U.S. Department of Commerce, National Bureau of Standards).

## 7.3 Deviates from Other Distributions

In §7.1 we learned how to generate random deviates with a uniform probability between 0 and 1, denoted  $U(0, 1)$ . The probability of generating a number between  $x$  and  $x + dx$  is

$$p(x)dx = \begin{cases} dx & 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases} \quad (7.3.1)$$

and we write

$$x \sim U(0, 1) \quad (7.3.2)$$

As in §6.14, the symbol  $\sim$  can be read as "is drawn from the distribution."

In this section, we learn how to generate random deviates drawn from other probability distributions, including all of those discussed in §6.14. Discussion of specific distributions is interleaved with the discussion of the general methods used.

### 7.3.1 Exponential Deviates

Suppose that we generate a uniform deviate  $x$  and then take some prescribed function of it,  $y(x)$ . The probability distribution of  $y$ , denoted  $p(y)dy$ , is determined by the fundamental transformation law of probabilities, which is simply

$$|p(y)dy| = |p(x)dx| \quad (7.3.3)$$

or

$$p(y) = p(x) \left| \frac{dx}{dy} \right| \quad (7.3.4)$$

As an example, take

$$y(x) = -\ln(x) \quad (7.3.5)$$

with  $x \sim U(0, 1)$ . Then

$$p(y)dy = \left| \frac{dx}{dy} \right| dy = e^{-y} dy \quad (7.3.6)$$

which is the exponential distribution with unit mean, Exponential(1), discussed in §6.14.5. This distribution occurs frequently in real life, usually as the distribution of waiting times between independent Poisson-random events, for example the radioactive decay of nuclei. You can also easily see (from 7.3.6) that the quantity  $y/\beta$  has the probability distribution  $\beta e^{-\beta y}$ , so

$$y/\beta \sim \text{Exponential}(\beta) \quad (7.3.7)$$

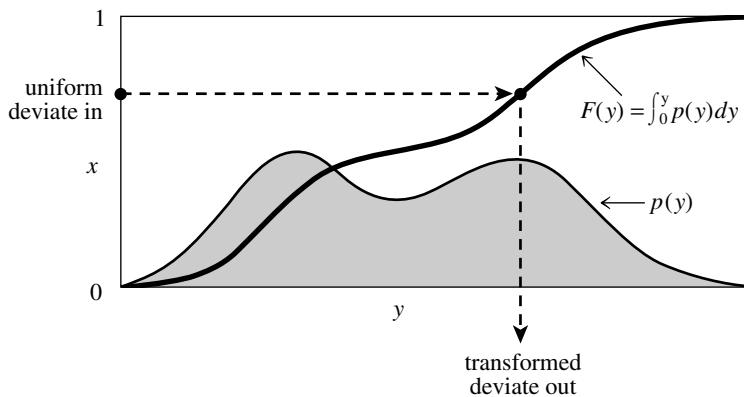
We can thus generate exponential deviates at a cost of about one uniform deviate, plus a logarithm, per call.

```
deviates.h struct Expondev : Ran {
    Structure for exponential deviates.
    Doub beta;
    Expondev(Doub bbeta, Ullong i) : Ran(i), beta(bbata) {}
    Constructor arguments are  $\beta$  and a random sequence seed.
    Doub dev() {
        Return an exponential deviate.
        Doub u;
        do u = doub(); while (u == 0.0);
        return -log(u)/beta;
    }
};
```

Our convention here and in the rest of this section is to derive the class for each kind of deviate from the uniform generator class `Ran`. We use the constructor to set the distribution's parameters and set the initial seed for the generator. We then provide a method `dev()` that returns a random deviate from the distribution.

### 7.3.2 Transformation Method in General

Let's see what is involved in using the above *transformation method* to generate some arbitrary desired distribution of  $y$ 's, say one with  $p(y) = f(y)$  for some positive function  $f$  whose integral is 1. According to (7.3.4), we need to solve the differential equation



**Figure 7.3.1.** Transformation method for generating a random deviate  $y$  from a known probability distribution  $p(y)$ . The indefinite integral of  $p(y)$  must be known and invertible. A uniform deviate  $x$  is chosen between 0 and 1. Its corresponding  $y$  on the definite-integral curve is the desired deviate.

$$\frac{dx}{dy} = f(y) \quad (7.3.8)$$

But the solution of this is just  $x = F(y)$ , where  $F(y)$  is the indefinite integral of  $f(y)$ . The desired transformation that takes a uniform deviate into one distributed as  $f(y)$  is therefore

$$y(x) = F^{-1}(x) \quad (7.3.9)$$

where  $F^{-1}$  is the inverse function to  $F$ . Whether (7.3.9) is feasible to implement depends on whether the *inverse function of the integral of  $f(y)$*  is itself feasible to compute, either analytically or numerically. Sometimes it is, and sometimes it isn't.

Incidentally, (7.3.9) has an immediate geometric interpretation: Since  $F(y)$  is the area under the probability curve to the left of  $y$ , (7.3.9) is just the prescription: Choose a uniform random  $x$ , then find the value  $y$  that has that fraction  $x$  of probability area to its left, and return the value  $y$ . (See Figure 7.3.1.)

### 7.3.3 Logistic Deviates

Deviates from the logistic distribution, as discussed in §6.14.4, are readily generated by the transformation method, using equation (6.14.15). The cost is again dominated by one uniform deviate, and a logarithm, per logistic deviate.

```
struct Logisticdev : Ran {
    Structure for logistic deviates.
    Doub mu,sig;
    Logisticdev(Doub mmu, Doub ssig, Ullong i) : Ran(i), mu(mmu), sig(ssig) {}
    Constructor arguments are  $\mu$ ,  $\sigma$ , and a random sequence seed.
    Doub dev() {
        Return a logistic deviate.
        Doub u;
        do u = doub(); while (u*(1.-u) == 0.);
        return mu + 0.551328895421792050*sig*log(u/(1.-u));
    }
};
```

deviates.h

### 7.3.4 Normal Deviates by Transformation (Box-Muller)

Transformation methods generalize to more than one dimension. If  $x_1, x_2, \dots$  are random deviates with a *joint* probability distribution  $p(x_1, x_2, \dots)dx_1dx_2\dots$ , and if  $y_1, y_2, \dots$  are each functions of all the  $x$ 's (same number of  $y$ 's as  $x$ 's), then the joint probability distribution of the  $y$ 's is

$$p(y_1, y_2, \dots)dy_1dy_2 \dots = p(x_1, x_2, \dots) \left| \frac{\partial(x_1, x_2, \dots)}{\partial(y_1, y_2, \dots)} \right| dy_1dy_2 \dots \quad (7.3.10)$$

where  $|\partial(\quad)/\partial(\quad)|$  is the Jacobian determinant of the  $x$ 's with respect to the  $y$ 's (or the reciprocal of the Jacobian determinant of the  $y$ 's with respect to the  $x$ 's).

An important historical example of the use of (7.3.10) is the *Box-Muller* method for generating random deviates with a normal (Gaussian) distribution (§6.14.1):

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \quad (7.3.11)$$

Consider the transformation between two uniform deviates on  $(0,1)$ ,  $x_1, x_2$ , and two quantities  $y_1, y_2$ ,

$$\begin{aligned} y_1 &= \sqrt{-2 \ln x_1} \cos 2\pi x_2 \\ y_2 &= \sqrt{-2 \ln x_1} \sin 2\pi x_2 \end{aligned} \quad (7.3.12)$$

Equivalently we can write

$$\begin{aligned} x_1 &= \exp \left[ -\frac{1}{2}(y_1^2 + y_2^2) \right] \\ x_2 &= \frac{1}{2\pi} \arctan \frac{y_2}{y_1} \end{aligned} \quad (7.3.13)$$

Now the Jacobian determinant can readily be calculated (try it!):

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = - \left[ \frac{1}{\sqrt{2\pi}} e^{-y_1^2/2} \right] \left[ \frac{1}{\sqrt{2\pi}} e^{-y_2^2/2} \right] \quad (7.3.14)$$

Since this is the product of a function of  $y_2$  alone and a function of  $y_1$  alone, we see that each  $y$  is independently distributed according to the normal distribution (7.3.11).

One further trick is useful in applying (7.3.12). Suppose that, instead of picking uniform deviates  $x_1$  and  $x_2$  in the unit square, we instead pick  $v_1$  and  $v_2$  as the ordinate and abscissa of a random point inside the unit circle around the origin. Then the sum of their squares,  $R^2 \equiv v_1^2 + v_2^2$ , is a uniform deviate, which can be used for  $x_1$ , while the angle that  $(v_1, v_2)$  defines with respect to the  $v_1$ -axis can serve as the random angle  $2\pi x_2$ . What's the advantage? It's that the cosine and sine in (7.3.12) can now be written as  $v_1/\sqrt{R^2}$  and  $v_2/\sqrt{R^2}$ , obviating the trigonometric function calls! (In the next section we will generalize this trick considerably.)

Code for generating normal deviates by the Box-Muller method follows. Consider it for pedagogical use only, because a significantly faster method for generating normal deviates is coming, below, in §7.3.9.

```

struct Normaldev_BM : Ran {
Structure for normal deviates.                                         deviates.h
    Doub mu,sig;
    Doub storedval;
    Normaldev_BM(Doub mmu, Doub ssig, Ullong i)
        : Ran(i), mu(mmu), sig(ssig), storedval(0.) {}
    Constructor arguments are  $\mu$ ,  $\sigma$ , and a random sequence seed.
    Doub dev() {
        Return a normal deviate.
        Doub v1,v2,rsq,fac;
        if (storedval == 0.) {                               We don't have an extra deviate handy, so
            do {
                v1=2.0*doub()-1.0;                         pick two uniform numbers in the square ex-
                v2=2.0*doub()-1.0;                         tending from -1 to +1 in each direction,
                rsq=v1*v1+v2*v2;                           see if they are in the unit circle,
            } while (rsq >= 1.0 || rsq == 0.0);           or try again.
            fac=sqrt(-2.0*log(rsq)/rsq);                 Now make the Box-Muller transformation to
            storedval = v1*fac;                          get two normal deviates. Return one and
            return mu + sig*v2*fac;                      save the other for next time.
        } else {                                         We have an extra deviate handy,
            fac = storedval;                           so return it.
            storedval = 0.;
            return mu + sig*fac;
        }
    }
};
```

### 7.3.5 Rayleigh Deviates

The *Rayleigh distribution* is defined for positive  $z$  by

$$p(z)dz = z \exp\left(-\frac{1}{2}z^2\right) dz \quad (z > 0) \quad (7.3.15)$$

Since the indefinite integral can be done analytically, and the result easily inverted, a simple transformation method from a uniform deviate  $x$  results:

$$z = \sqrt{-2 \ln x}, \quad x \sim U(0, 1) \quad (7.3.16)$$

A Rayleigh deviate  $z$  can also be generated from two normal deviates  $y_1$  and  $y_2$  by

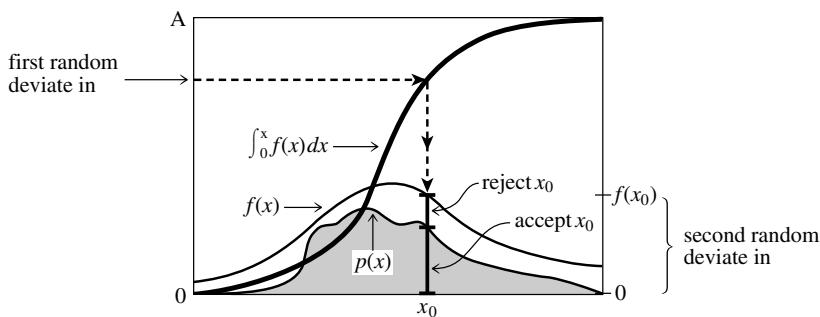
$$z = \sqrt{y_1^2 + y_2^2}, \quad y_1, y_2 \sim N(0, 1) \quad (7.3.17)$$

Indeed, the relation between equations (7.3.16) and (7.3.17) is immediately evident in the equation for the Box-Muller method, equation (7.3.12), if we square and sum that method's two normal deviates  $y_1$  and  $y_2$ .

### 7.3.6 Rejection Method

The *rejection method* is a powerful, general technique for generating random deviates whose distribution function  $p(x)dx$  (probability of a value occurring between  $x$  and  $x + dx$ ) is known and computable. The rejection method does not require that the cumulative distribution function (indefinite integral of  $p(x)$ ) be readily computable, much less the inverse of that function — which was required for the transformation method in the previous section.

The rejection method is based on a simple geometrical argument (Figure 7.3.2):



**Figure 7.3.2.** Rejection method for generating a random deviate  $x$  from a known probability distribution  $p(x)$  that is everywhere less than some other function  $f(x)$ . The transformation method is first used to generate a random deviate  $x$  of the distribution  $f$  (compare Figure 7.3.1). A second uniform deviate is used to decide whether to accept or reject that  $x$ . If it is rejected, a new deviate of  $f$  is found, and so on. The ratio of accepted to rejected points is the ratio of the area under  $p$  to the area between  $p$  and  $f$ .

Draw a graph of the probability distribution  $p(x)$  that you wish to generate, so that the area under the curve in any range of  $x$  corresponds to the desired probability of generating an  $x$  in that range. If we had some way of choosing a random point *in two dimensions*, with uniform probability in the *area* under your curve, then the  $x$  value of that random point would have the desired distribution.

Now, on the same graph, draw any other curve  $f(x)$  that has finite (not infinite) area and lies everywhere *above* your original probability distribution. (This is always possible, because your original curve encloses only unit area, by definition of probability.) We will call this  $f(x)$  the *comparison function*. Imagine now that you have some way of choosing a random point in two dimensions that is uniform in the area under the comparison function. Whenever that point lies outside the area under the original probability distribution, we will *reject* it and choose another random point. Whenever it lies inside the area under the original probability distribution, we will *accept* it.

It should be obvious that the accepted points are uniform in the accepted area, so that their  $x$  values have the desired distribution. It should also be obvious that the fraction of points rejected just depends on the ratio of the area of the comparison function to the area of the probability distribution function, not on the details of shape of either function. For example, a comparison function whose area is less than 2 will reject fewer than half the points, even if it approximates the probability function very badly at some values of  $x$ , e.g., remains finite in some region where  $p(x)$  is zero.

It remains only to suggest how to choose a uniform random point in two dimensions under the comparison function  $f(x)$ . A variant of the transformation method (§7.3) does nicely: Be sure to have chosen a comparison function whose indefinite integral is known analytically, and is also analytically invertible to give  $x$  as a function of “area under the comparison function to the left of  $x$ .” Now pick a uniform deviate between 0 and  $A$ , where  $A$  is the total area under  $f(x)$ , and use it to get a corresponding  $x$ . Then pick a uniform deviate between 0 and  $f(x)$  as the  $y$  value for the two-dimensional point. Finally, accept or reject according to whether it is respectively less than or greater than  $p(x)$ .

So, to summarize, the rejection method for some given  $p(x)$  requires that one find, once and for all, some reasonably good comparison function  $f(x)$ . Thereafter,

each deviate generated requires two uniform random deviates, one evaluation of  $f$  (to get the coordinate  $y$ ) and one evaluation of  $p$  (to decide whether to accept or reject the point  $x, y$ ). Figure 7.3.1 illustrates the whole process. Then, of course, this process may need to be repeated, on the average,  $A$  times before the final deviate is obtained.

### 7.3.7 Cauchy Deviates

The “further trick” described following equation (7.3.14) in the context of the Box-Muller method is now seen to be a rejection method for getting trigonometric functions of a uniformly random angle. If we combine this with the explicit formula, equation (6.14.6), for the inverse cdf of the Cauchy distribution (see §6.14.2), we can generate Cauchy deviates quite efficiently.

```
struct Cauchydev : Ran {  
    Structure for Cauchy deviates.  
    Doub mu,sig;  
    Cauchydev(Doub mmu, Doub ssig, Ullong i) : Ran(i), mu(mmu), sig(ssig) {}  
    Constructor arguments are  $\mu$ ,  $\sigma$ , and a random sequence seed.  
    Doub dev() {  
        Return a Cauchy deviate.  
        Doub v1,v2;  
        do {  
            v1=2.0*doub()-1.0;           Find a random point in the unit semicircle.  
            v2=doub();  
        } while (SQR(v1)+SQR(v2) >= 1. || v2 == 0.);  
        return mu + sig*v1/v2;         Ratio of its coordinates is the tangent of a  
        }                                random angle.  
};
```

deviates.h

### 7.3.8 Ratio-of-Uniforms Method

In finding Cauchy deviates, we took the ratio of two uniform deviates chosen to lie within the unit circle. If we generalize to shapes other than the unit circle, and combine it with the principle of the rejection method, a powerful variant emerges. Kinderman and Monahan [1] showed that deviates of virtually *any* probability distribution  $p(x)$  can be generated by the following rather amazing prescription:

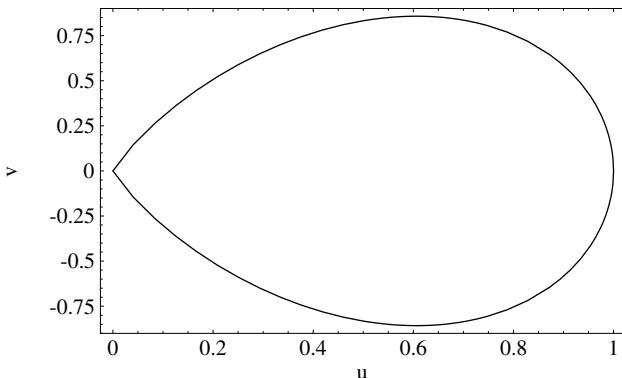
- Construct the region in the  $(u, v)$  plane bounded by  $0 \leq u \leq [p(v/u)]^{1/2}$ .
- Choose two deviates,  $u$  and  $v$ , that lie uniformly in this region.
- Return  $v/u$  as the deviate.

Proof: We can represent the ordinary rejection method by the equation in the  $(x, p)$  plane,

$$p(x)dx = \int_{p'=0}^{p'=p(x)} dp' dx \quad (7.3.18)$$

Since the integrand is 1, we are justified in sampling uniformly in  $(x, p')$  as long as  $p'$  is within the limits of the integral (that is,  $0 < p' < p(x)$ ). Now make the change of variable

$$\begin{aligned} \frac{v}{u} &= x \\ u^2 &= p \end{aligned} \quad (7.3.19)$$



**Figure 7.3.3.** Ratio-of-uniforms method. The interior of this teardrop shape is the acceptance region for the normal distribution: If a random point is chosen inside this region, then the ratio  $v/u$  will be a normal deviate.

Then equation (7.3.18) becomes

$$p(x)dx = \int_{p'=0}^{p'=p(x)} dp' dx = \int_{u=0}^{u=\sqrt{p(x)}} \frac{\partial(p, x)}{\partial(u, v)} du dv = 2 \int_{u=0}^{u=\sqrt{p(v/u)}} du dv \quad (7.3.20)$$

because (as you can work out) the Jacobian determinant is the constant 2. Since the new integrand is constant, uniform sampling in  $(u, v)$  with the limits indicated for  $u$  is equivalent to the rejection method in  $(x, p)$ .

The above limits on  $u$  very often define a region that is “teardrop” shaped. To see why, note that the locii of constant  $x = v/u$  are radial lines. Along each radial, the acceptance region goes from the origin to a point where  $u^2 = p(x)$ . Since most probability distributions go to zero for both large and small  $x$ , the acceptance region accordingly shrinks toward the origin along radials, producing a teardrop. Of course, it is the exact shape of this teardrop that matters. Figure 7.3.3 shows the shape of the acceptance region for the case of the normal distribution.

Typically this *ratio-of-uniforms* method is used when the desired region can be closely bounded by a rectangle, parallelogram, or some other shape that is easy to sample uniformly. Then, we go from sampling the easy shape to sampling the desired region by rejection of points outside the desired region.

An important adjunct to the ratio-of-uniforms method is the idea of a *squeeze*. A squeeze is any easy-to-compute shape that tightly bounds the region of acceptance of a rejection method, either from the inside or from the outside. Best of all is when you have squeezes on both sides. Then you can immediately reject points that are outside the outer squeeze and immediately accept points that are inside the inner squeeze. Only when you have the bad luck of drawing a point between the two squeezes do you actually have to do the more lengthy computation of comparing with the actual rejection boundary. Squeezes are useful both in the ordinary rejection method and in the ratio-of-uniforms method.

### 7.3.9 Normal Deviates by Ratio-of-Uniforms

Leva [2] has given an algorithm for normal deviates that uses the ratio-of-uniforms method with great success. He uses quadratic curves to provide both inner

and outer squeezes that hug the desired region in the  $(u, v)$  plane (Figure 7.3.3). Only about 1% of the time is it necessary to calculate an exact boundary (requiring a logarithm).

The resulting code looks so simple and “un-transcendental” that it may be hard to believe that exact normal deviates are generated. But they are!

```
struct Normaldev : Ran {
    Doub mu,sig;
    Normaldev(Doub mmu, Doub ssig, Ullong i)
        : Ran(i), mu(mmu), sig(ssig){}
    Constructor arguments are  $\mu$ ,  $\sigma$ , and a random sequence seed.
    Doub dev() {
        Return a normal deviate.
        Doub u,v,x,y,q;
        do {
            u = doub();
            v = 1.7156*(doub()-0.5);
            x = u - 0.449871;
            y = abs(v) + 0.386595;
            q = SQR(x) + y*(0.19600*y-0.25472*x);
        } while (q > 0.27597
            && (q > 0.27846 || SQR(v) > -4.*log(u)*SQR(u)));
        return mu + sig*v/u;
    }
};
```

deviates.h

Note that the `while` clause makes use of C’s (and C++’s) guarantee that logical expressions are evaluated conditionally: If the first operand is sufficient to determine the outcome, the second is not evaluated at all. With these rules, the logarithm is evaluated only when  $q$  is between 0.27597 and 0.27846.

On average, each normal deviate uses 2.74 uniform deviates. By the way, even though the various constants are given only to six digits, the method is exact (to full double precision). Small perturbations of the bounding curves are of no consequence. The accuracy is implicit in the (rare) evaluations of the exact boundary.

### 7.3.10 Gamma Deviates

The distribution  $\text{Gamma}(\alpha, \beta)$  was described in §6.14.9. The  $\beta$  parameter enters only as a scaling,

$$\text{Gamma}(\alpha, \beta) \cong \frac{1}{\beta} \text{Gamma}(\alpha, 1) \quad (7.3.21)$$

(Translation: To generate a  $\text{Gamma}(\alpha, \beta)$  deviate, generate a  $\text{Gamma}(\alpha, 1)$  deviate and divide it by  $\beta$ .)

If  $\alpha$  is a small positive integer, a fast way to generate  $x \sim \text{Gamma}(\alpha, 1)$  is to use the fact that it is distributed as the waiting time to the  $\alpha$ th event in a Poisson random process of unit mean. Since the time between two consecutive events is just the exponential distribution  $\text{Exponential}(1)$ , you can simply add up  $\alpha$  exponentially distributed waiting times, i.e., logarithms of uniform deviates. Even better, since the sum of logarithms is the logarithm of the product, you really only have to compute the product of  $\alpha$  uniform deviates and then take the log. Because this is such a special case, however, we don’t include it in the code below.

When  $\alpha < 1$ , the gamma distribution's density function is not bounded, which is inconvenient. However, it turns out [4] that if

$$y \sim \text{Gamma}(\alpha + 1, 1), \quad u \sim \text{Uniform}(0, 1) \quad (7.3.22)$$

then

$$yu^{1/\alpha} \sim \text{Gamma}(\alpha, 1) \quad (7.3.23)$$

We will use this in the code below.

For  $\alpha > 1$ , Marsaglia and Tsang [5] give an elegant rejection method based on a simple transformation of the gamma distribution combined with a squeeze. After transformation, the gamma distribution can be bounded by a Gaussian curve whose area is never more than 5% greater than that of the gamma curve. The cost of a gamma deviate is thus only a little more than the cost of the normal deviate that is used to sample the comparison function. The following code gives the precise formulation; see the original paper for a full explanation.

deviates.h

```
struct Gammadev : Normaldev {
    Structure for gamma deviates.
    Doub alph, oalph, bet;
    Doub a1,a2;
    Gammadev(Doub aalph, Doub bbet, Ullong i)
        : Normaldev(0.,1.,i), alph(aalph), oalph(aalph), bet(bbet) {
        Constructor arguments are  $\alpha$ ,  $\beta$ , and a random sequence seed.
        if (alph <= 0.) throw("bad alph in Gammadev");
        if (alph < 1.) alph += 1.;
        a1 = alph-1./3.;
        a2 = 1./sqrt(9.*a1);
    }
    Doub dev() {
        Return a gamma deviate by the method of Marsaglia and Tsang.
        Doub u,v,x;
        do {
            do {
                x = Normaldev::dev();
                v = 1. + a2*x;
            } while (v <= 0.);
            v = v*v*v;
            u = doub();
        } while (u > 1. - 0.331*SQR(SQR(x)) &&
            log(u) > 0.5*SQR(x) + a1*(1.-v+log(v))); Rarely evaluated.
        if (alph == oalph) return a1*v/bet;
        else {                                     Case where  $\alpha < 1$ , per Ripley.
            do u=doub(); while (u == 0.);           }
            return pow(u,1./alph)*a1*v/bet;
        }
    };
};
```

There exists a sum rule for gamma deviates. If we have a set of independent deviates  $y_i$  with possibly different  $\alpha_i$ 's, but sharing a common value of  $\beta$ ,

$$y_i \sim \text{Gamma}(\alpha_i, \beta) \quad (7.3.24)$$

then their sum is also a gamma deviate,

$$y \equiv \sum_i y_i \sim \text{Gamma}(\alpha_T, \beta), \quad \alpha_T = \sum_i \alpha_i \quad (7.3.25)$$

If the  $\alpha_i$ 's are integers, you can see how this relates to the discussion of Poisson waiting times above.

### 7.3.11 Distributions Easily Generated by Other Deviates

From normal, gamma and uniform deviates, we get a bunch of other distributions for free. Important: When you are going to combine their results, be sure that all distinct instances of `Normaldist`, `Gammadist`, and `Ran` have different random seeds! (`Ran` and its derived classes are sufficiently robust that seeds  $i, i + 1, \dots$  are fine.)

#### Chi-Square Deviates (cf. §6.14.8)

This one is easy:

$$\text{Chisquare}(\nu) \cong \text{Gamma}\left(\frac{\nu}{2}, \frac{1}{2}\right) \cong 2 \text{Gamma}\left(\frac{\nu}{2}, 1\right) \quad (7.3.26)$$

#### Student-t Deviates (cf. §6.14.3)

Deviates from the Student-t distribution can be generated by a method very similar to the Box-Muller method. The analog of equation (7.3.12) is

$$y = \sqrt{\nu(u_1^{-2/\nu} - 1)} \cos 2\pi u_2 \quad (7.3.27)$$

If  $u_1$  and  $u_2$  are independently uniform,  $U(0, 1)$ , then

$$y \sim \text{Student}(\nu, 0, 1) \quad (7.3.28)$$

or

$$\mu + \sigma y \sim \text{Student}(\nu, \mu, \sigma) \quad (7.3.29)$$

Unfortunately, you can't do the Box-Muller trick of getting two deviates at a time, because the Jacobian determinant analogous to equation (7.3.14) does not factorize. You might want to use the polar method anyway, just to get  $\cos 2\pi u_2$ , but its advantage is now not so large.

An alternative method uses the quotients of normal and gamma deviates. If we have

$$x \sim N(0, 1), \quad y \sim \text{Gamma}\left(\frac{\nu}{2}, \frac{1}{2}\right) \quad (7.3.30)$$

then

$$x \sqrt{\nu/y} \sim \text{Student}(\nu, 0, 1) \quad (7.3.31)$$

#### Beta Deviates (cf. §6.14.11)

If

$$x \sim \text{Gamma}(\alpha, 1), \quad y \sim \text{Gamma}(\beta, 1) \quad (7.3.32)$$

then

$$\frac{x}{x + y} \sim \text{Beta}(\alpha, \beta) \quad (7.3.33)$$

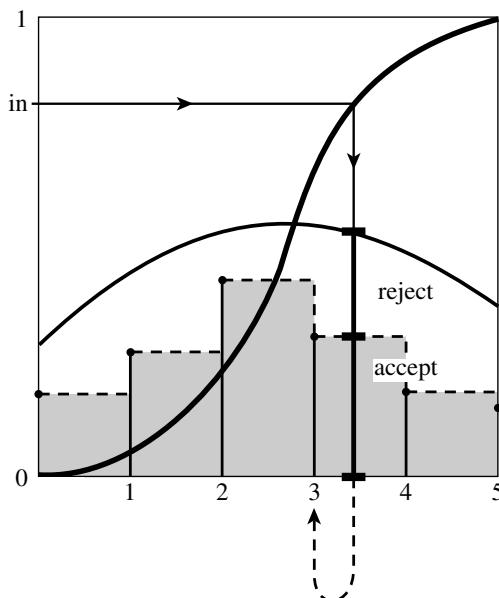
#### F-Distribution Deviates (cf. §6.14.10)

If

$$x \sim \text{Beta}\left(\frac{1}{2}\nu_1, \frac{1}{2}\nu_2\right) \quad (7.3.34)$$

(see equation 7.3.33), then

$$\frac{\nu_2 x}{\nu_1(1 - x)} \sim F(\nu_1, \nu_2) \quad (7.3.35)$$



**Figure 7.3.4.** Rejection method as applied to an integer-valued distribution. The method is performed on the step function shown as a dashed line, yielding a real-valued deviate. This deviate is rounded down to the next lower integer, which is output.

### 7.3.12 Poisson Deviates

The Poisson distribution,  $\text{Poisson}(\lambda)$ , previously discussed in §6.14.13, is a discrete distribution, so its deviates will be integers,  $k$ . To use the methods already discussed, it is convenient to convert the Poisson distribution into a continuous distribution by the following trick: Consider the finite probability  $p(k)$  as being spread out uniformly into the interval from  $k$  to  $k + 1$ . This defines a continuous distribution  $q_\lambda(k)dk$  given by

$$q_\lambda(k)dk = \frac{\lambda^{\lfloor k \rfloor} e^{-\lambda}}{\lfloor k \rfloor!} dk \quad (7.3.36)$$

where  $\lfloor k \rfloor$  represents the largest integer  $\leq k$ . If we now use a rejection method, or any other method, to generate a (noninteger) deviate from (7.3.36), and then take the integer part of that deviate, it will be as if drawn from the discrete Poisson distribution. (See Figure 7.3.4.) This trick is general for any integer-valued probability distribution. Instead of the “floor” operator, one can equally well use “ceiling” or “nearest” — anything that spreads the probability over a unit interval.

For  $\lambda$  large enough, the distribution (7.3.36) is qualitatively bell-shaped (albeit with a bell made out of small, square steps). In that case, the ratio-of-uniforms method works well. It is not hard to find simple inner and outer squeezes in the  $(u, v)$  plane of the form  $v^2 = Q(u)$ , where  $Q(u)$  is a simple polynomial in  $u$ . The only trick is to allow a big enough gap between the squeezes to enclose the true, jagged, boundaries for all values of  $\lambda$ . (Look ahead to Figure 7.3.5 for a similar example.)

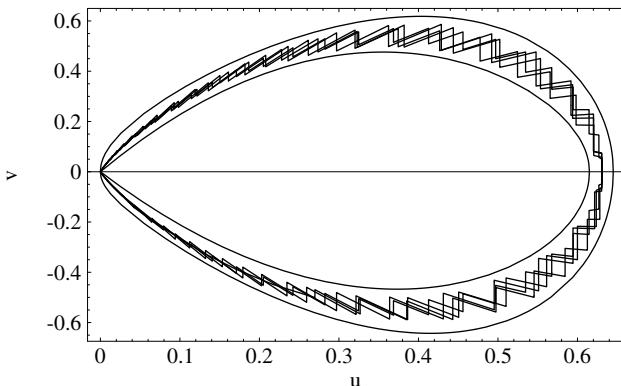
For intermediate values of  $\lambda$ , the jaggedness is so large as to render squeezes impractical, but the ratio-of-uniforms method, unadorned, still works pretty well.

For small  $\lambda$ , we can use an idea similar to that mentioned above for the gamma distribution in the case of integer  $a$ . When the sum of independent exponential

deviates first exceeds  $\lambda$ , their number (less 1) is a Poisson deviate  $k$ . Also, as explained for the gamma distribution, we can multiply uniform deviates from  $U(0, 1)$  instead of adding deviates from Exponential (1).

These ideas produce the following routine.

```
struct Poissondev : Ran {  
    Doub lambda, sqlam, loglam, lamexp, lambold;  
    VecDoub logfact;  
    Int swch;  
    Poissondev(Doub llambda, Ullong i) : Ran(i), lambda(llambda),  
        logfact(1024,-1.), lambold(-1.) {}  
    Constructor arguments are  $\lambda$  and a random sequence seed.  
    Int dev() {  
        Return a Poisson deviate using the most recently set value of  $\lambda$ .  
        Doub u,u2,v,v2,p,t,lfac;  
        Int k;  
        if (lambda < 5.) {  
            if (lambda != lambold) lamexp=exp(-lambda);  
            k = -1;  
            t=1.;  
            do {  
                ++k;  
                t *= doub();  
            } while (t > lamexp);  
        } else {  
            if (lambda != lambold) {  
                sqlam = sqrt(lambda);  
                loglam = log(lambda);  
            }  
            for (;;) {  
                u = 0.64*doub();  
                v = -0.68 + 1.28*doub();  
                if (lambda > 13.5) {  
                    v2 = SQR(v);  
                    if (v >= 0.) {if (v2 > 6.5*u*(0.64-u)*(u+0.2)) continue;}  
                    else {if (v2 > 9.6*u*(0.66-u)*(u+0.07)) continue;}  
                }  
                k = Int(floor(sqlam*(v/u)+lambda+0.5));  
                if (k < 0) continue;  
                u2 = SQR(u);  
                if (lambda > 13.5) {  
                    Inner squeeze for fast acceptance.  
                    if (v >= 0.) {if (v2 < 15.2*u2*(0.61-u)*(0.8-u)) break;}  
                    else {if (v2 < 6.76*u2*(0.62-u)*(1.4-u)) break;}  
                }  
                if (k < 1024) {  
                    if (logfact[k] < 0.) logfact[k] = gammeln(k+1.);  
                    lfac = logfact[k];  
                } else lfac = gammeln(k+1.);  
                p = sqlam*exp(-lambda + k*loglam - lfac);  
                Only when we must.  
                if (u2 < p) break;  
            }  
        }  
        lambold = lambda;  
        return k;  
    }  
    Int dev(Doub llambda) {  
        Reset  $\lambda$  and then return a Poisson deviate.  
        lambda = llambda;  
        return dev();  
    }  
};
```



**Figure 7.3.5.** Ratio-of-uniforms method as applied to the generation of binomial deviates. Points are chosen randomly in the  $(u, v)$ -plane. The smooth curves are inner and outer squeezes. The jagged curves correspond to various binomial distributions with  $n > 64$  and  $np > 30$ . An evaluation of the binomial probability is required only when the random point falls between the smooth curves.

In the regime  $\lambda > 13.5$ , the above code uses about 3.3 uniform deviates per output Poisson deviate and does 0.4 evaluations of the exact probability (costing an exponential and, for large  $k$ , a call to `gammaln`).

`Poissondev` is slightly faster if you draw many deviates with the same value  $\lambda$ , using the `dev` function with no arguments, than if you vary  $\lambda$  on each call, using the one-argument overloaded form of `dev` (which is provided for just that purpose). The difference is just an extra exponential ( $\lambda < 5$ ) or square root and logarithm ( $\lambda \geq 5$ ). Note also the object's table of previously computed log-factorials. If your  $\lambda$ 's are as large as  $\sim 10^3$ , you might want to make the table larger.

### 7.3.13 Binomial Deviates

The generation of binomial deviates  $k \sim \text{Binomial}(n, p)$  involves many of the same ideas as for Poisson deviates. The distribution is again integer-valued, so we use the same trick to convert it into a stepped continuous distribution. We can always restrict attention to the case  $p \leq 0.5$ , since the distribution's symmetries let us trivially recover the case  $p > 0.5$ .

When  $n > 64$  and  $np > 30$ , we use the ratio-of-uniforms method, with squeezes shown in Figure 7.3.5. The cost is about 3.2 uniform deviates, plus 0.4 evaluations of the exact probability, per binomial deviate.

It would be foolish to waste much thought on the case where  $n > 64$  and  $np < 30$ , because it is so easy simply to tabulate the cdf, say for  $0 \leq k < 64$ , and then loop over  $k$ 's until the right one is found. (A bisection search, implemented below, is even better.) With a cdf table of length 64, the neglected probability at the end of the table is never larger than  $\sim 10^{-20}$ . (At  $10^9$  deviates per second, you could run 3000 years before losing a deviate.)

What is left is the interesting case  $n < 64$ , which we will explore in some detail, because it demonstrates the important concept of *bit-parallel random comparison*.

Analogous to the methods for gamma deviates with small integer  $a$  and for Poisson deviates with small  $\lambda$ , is this direct method for binomial deviates: Generate  $n$  uniform deviates in  $U(0, 1)$ . Count the number of them  $< p$ . Return the count as

$k \sim \text{Binomial}(n, p)$ . Indeed this is essentially the definition of a binomial process!

The problem with the direct method is that it seems to require  $n$  uniform deviates, even when the mean value of  $k$  is much smaller. Would you be surprised if we told you that for  $n \leq 64$  you can achieve the same goal with at most *seven* 64-bit uniform deviates, on average? Here is how.

Expand  $p < 1$  into its first 5 bits, plus a residual,

$$p = b_1 2^{-1} + b_2 2^{-2} + \cdots + b_5 2^{-5} + p_r 2^{-5} \quad (7.3.37)$$

where each  $b_i$  is 0 or 1, and  $0 \leq p_r \leq 1$ .

Now imagine that you have generated and stored 64 uniform  $U(0, 1)$  deviates, and that the 64-bit word  $P$  displays just the first bit of each of the 64. Compare each bit of  $P$  to  $b_1$ . If the bits are the same, then we don't yet know whether that uniform deviate is less than or greater than  $p$ . But if the bits are *different*, then we know that the generator is less than  $p$  (in the case that  $b_1 = 1$ ) or greater than  $p$  (in the case that  $b_1 = 0$ ). If we keep a mask of “known” versus “unknown” cases, we can do these comparisons in a bit-parallel manner by bitwise logical operations (see code below to learn how). Now move on to the second bit,  $b_2$ , in the same way. At each stage we change half the remaining unknowns to knowns. After five stages (for  $n = 64$ ) there will be two remaining unknowns, on average, each of which we finish off by generating a new uniform and comparing it to  $p_r$ . (This requires a loop through the 64 bits; but since C++ has no bitwise “popcount” operation, we are stuck doing such a loop anyway. If you can do popcounts, you may be better off just doing more stages until the unknowns mask is zero.)

The trick is that the bits used in the five stages are not actually the leading five bits of 64 generators, they are just five independent 64-bit random integers. The number five was chosen because it minimizes  $64 \times 2^{-j} + j$ , the expected number of deviates needed.

So, the code for binomial deviates ends up with three separate methods: bit-parallel direct, cdf lookup (by bisection), and squeezed ratio-of-uniforms.

```
struct Binomialdev : Ran {
    Structure for binomial deviates.
    Doub pp,p,pb,expnp,np,glnp,plog,pclog,sq;
    Int n,swch;
    Ullong uz,uo,unfin,diff,rntp;
    Int pbits[5];
    Doub cdf[64];
    Doub logfact[1024];
    Binomialdev(Int nn, Doub ppp, Ullong i) : Ran(i), pp(ppp), n(nn) {
        Constructor arguments are n, p, and a random sequence seed.
        Int j;
        pb = p = (pp <= 0.5 ? pp : 1.0-pp);
        if (n <= 64) {                                Will use bit-parallel direct method.
            uz=0;
            uo=0xffffffffffffffffLL;
            rntp = 0;
            for (j=0;j<5;j++) pbits[j] = 1 & ((Int)(pb *= 2));
            pb -= floor(pb);                         Leading bits of p (above) and remaining
            swch = 0;                                  fraction.
        } else if (n*p < 30.) {                      Will use precomputed cdf table.
            cdf[0] = exp(n*log(1-p));
            for (j=1;j<64;j++) cdf[j] = cdf[j-1] + exp(gammln(n+1.)
                -gammln(j+1.)-gammln(n-j+1.)+j*log(p)+(n-j)*log(1.-p));
            swch = 1;
        }
    }
}
```

deviates.h

```

} else {                                Will use ratio-of-uniforms method.
    np = n*p;
    glnp=gammln(n+1.);
    plog=log(p);
    pclog=log(1.-p);
    sq=sqrt(np*(1.-p));
    if (n < 1024) for (j=0;j<=n;j++) logfact[j] = gammln(j+1.);
    swch = 2;
}
}

Int dev() {
Return a binomial deviate.
    Int j,k,k1,km;
    Doub y,u,v,u2,v2,b;
    if (swch == 0) {
        unfin = uo;                         Mark all bits as "unfinished."
        for (j=0;j<5;j++) {                  Compare with first five bits of  $p$ .
            diff = unfin & (int64()^(pbits[j]? uo : uz)); Mask of diff.
            if (pbits[j] rltp |= diff);       Set bits to 1, meaning ran <  $p$ .
            else rltp = rltp & ~diff;        Set bits to 0, meaning ran >  $p$ .
            unfin = unfin & ~diff;          Update unfinished status.
        }
        k=0;                                 Now we just count the events.
        for (j=0;j<n;j++) {
            if (unfin & 1) {if (doub() < pb) ++k;}     Clean up unresolved cases,
            else if (rltp & 1) ++k;}                     or use bit answer.
            unfin >>= 1;
            rltp >>= 1;
        }
    } else if (swch == 1) {                Use stored cdf.
        y = doub();
        k1 = -1;
        k = 64;
        while (k-k1>1) {
            km = (k1+k)/2;
            if (y < cdf[km]) k = km;
            else k1 = km;
        }
    } else {                                Use ratio-of-uniforms method.
        for (;;) {
            u = 0.645*doub();
            v = -0.63 + 1.25*doub();
            v2 = SQR(v);
            Try squeeze for fast rejection:
            if (v >= 0.) {if (v2 > 6.5*u*(0.645-u)*(u+0.2)) continue;}
            else {if (v2 > 8.4*u*(0.645-u)*(u+0.1)) continue;}
            k = Int(floor(sq*(v/u)+np+0.5));
            if (k < 0) continue;
            u2 = SQR(u);
            Try squeeze for fast acceptance:
            if (v >= 0.) {if (v2 < 12.25*u2*(0.615-u)*(0.92-u)) break;}
            else {if (v2 < 7.84*u2*(0.615-u)*(1.2-u)) break;}
            b = sq*exp(glnp+k*plog+(n-k)*pclog) Only when we must.
            - (n < 1024 ? logfact[k]+logfact[n-k]
                         : gammln(k+1.)+gammln(n-k+1.)));
            if (u2 < b) break;
        }
    }
    if (p != pp) k = n - k;
    return k;
}
};

```

If you are in a situation where you are drawing only one or a few deviates each for many different values of  $n$  and/or  $p$ , you'll need to restructure the code so that  $n$  and  $p$  can be changed without creating a new instance of the object and without reinitializing the underlying Ran generator.

### 7.3.14 When You Need Greater Speed

In particular situations you can cut some corners to gain greater speed. Here are some suggestions.

- All of the algorithms in this section can be speeded up significantly by using Ranq1 in §7.1 instead of Ran. We know of no reason not to do this. You can gain some further speed by coding Ranq1's algorithm inline, thus eliminating the function calls.
- If you are using Poissondev or Binomialdev with large values of  $\lambda$  or  $n$ , then the above codes revert to calling gammln, which is slow. You can instead increase the length of the stored tables.
- For Poisson deviates with  $\lambda < 20$ , you may want to use a stored table of cdfs combined with bisection to find the value of  $k$ . The code in Binomialdev shows how to do this.
- If your need is for binomial deviates with small  $n$ , you can easily modify the code in Binomialdev to get multiple deviates ( $\sim 64/n$ , in fact) from each execution of the bit-parallel code.
- Do you need exact deviates, or would an approximation do? If your distribution of interest can be approximated by a normal distribution, consider substituting Normaldev, above, especially if you also code the uniform random generation inline.
- If you sum exactly 12 uniform deviates  $U(0, 1)$  and then subtract 6, you get a pretty good approximation of a normal deviate  $N(0, 1)$ . This is definitely slower than Normaldev (not to mention less accurate) on a general-purpose CPU. However, there are reported to be some special-purpose signal processing chips in which all the operations can be done with integer arithmetic and in parallel.

See Gentle [3], Ripley [4], Devroye [6], Bratley [7], and Knuth [8] for many additional algorithms.

#### CITED REFERENCES AND FURTHER READING:

- Kinderman, A.J. and Monahan, J.F 1977, "Computer Generation of Random Variables Using the Ratio of Uniform Deviates," *ACM Transactions on Mathematical Software*, vol. 3, pp. 257–260.[1]
- Leva, J.L. 1992. "A Fast Normal Random Number Generator," *ACM Transactions on Mathematical Software*, vol. 18, no. 4, pp. 449-453.[2]
- Gentle, J.E. 2003, *Random Number Generation and Monte Carlo Methods*, 2nd ed. (New York: Springer), Chapters 4–5.[3]
- Ripley, B.D. 1987, *Stochastic Simulation* (New York: Wiley).[4]
- Marsaglia, G. and Tsang W-W. 2000, "A Simple Method for Generating Gamma Variables," *ACM Transactions on Mathematical Software*, vol. 26, no. 3, pp. 363–372.[5]
- Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer).[6]

- Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation*, 2nd ed. (New York: Springer).[7].
- Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 125ff.[8]

## 7.4 Multivariate Normal Deviates

A multivariate random deviate of dimension  $M$  is a point in  $M$ -dimensional space. Its coordinates are a vector, each of whose  $M$  components are random — but not, in general, independently so, or identically distributed. The special case of *multivariate normal deviates* is defined by the multidimensional Gaussian density function

$$N(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{M/2} \det(\boldsymbol{\Sigma})^{1/2}} \exp[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}) \cdot \boldsymbol{\Sigma}^{-1} \cdot (\mathbf{x} - \boldsymbol{\mu})] \quad (7.4.1)$$

where the parameter  $\boldsymbol{\mu}$  is a vector that is the mean of the distribution, and the parameter  $\boldsymbol{\Sigma}$  is a symmetrical, positive-definite matrix that is the distribution's covariance.

There is a quite general way to construct a vector deviate  $\mathbf{x}$  with a specified covariance  $\boldsymbol{\Sigma}$  and mean  $\boldsymbol{\mu}$ , starting with a vector  $\mathbf{y}$  of independent random deviates of zero mean and unit variance: First, use Cholesky decomposition (§2.9) to factor  $\boldsymbol{\Sigma}$  into a left triangular matrix  $\mathbf{L}$  times its transpose,

$$\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^T \quad (7.4.2)$$

This is always possible because  $\boldsymbol{\Sigma}$  is positive-definite, and you need do it only once for each distinct  $\boldsymbol{\Sigma}$  of interest. Next, whenever you want a new deviate  $\mathbf{x}$ , fill  $\mathbf{y}$  with independent deviates of unit variance and then construct

$$\mathbf{x} = \mathbf{L}\mathbf{y} + \boldsymbol{\mu} \quad (7.4.3)$$

The proof is straightforward, with angle brackets denoting expectation values: Since the components  $y_i$  are independent with unit variance, we have

$$\langle \mathbf{y} \otimes \mathbf{y} \rangle = \mathbf{1} \quad (7.4.4)$$

where  $\mathbf{1}$  is the identity matrix. Then,

$$\begin{aligned} \langle (\mathbf{x} - \boldsymbol{\mu}) \otimes (\mathbf{x} - \boldsymbol{\mu}) \rangle &= \langle (\mathbf{L}\mathbf{y}) \otimes (\mathbf{L}\mathbf{y}) \rangle \\ &= \left\langle \mathbf{L}(\mathbf{y} \otimes \mathbf{y})\mathbf{L}^T \right\rangle = \mathbf{L} \langle \mathbf{y} \otimes \mathbf{y} \rangle \mathbf{L}^T \\ &= \mathbf{L}\mathbf{L}^T = \boldsymbol{\Sigma} \end{aligned} \quad (7.4.5)$$

As general as this procedure is, it is, however, rarely useful for anything except multivariate *normal* deviates. The reason is that while the components of  $\mathbf{x}$  indeed have the right mean and covariance structure, their detailed distribution is not anything “nice.” The  $x_i$ ’s are linear combinations of the  $y_i$ ’s, and, in general, a linear combination of random variables is distributed as a complicated convolution of their individual distributions.

For Gaussians, however, we do have “nice.” All linear combinations of normal deviates are themselves normally distributed, and completely defined by their mean and covariance structure. Thus, if we always fill the components of  $\mathbf{y}$  with normal deviates,

$$y_i \sim N(0, 1) \quad (7.4.6)$$

then the deviate (7.4.3) will be distributed according to equation (7.4.1).

Implementation is straightforward, since the Cholesky structure both accomplishes the decomposition and provides a method for doing the matrix multiplication efficiently, taking advantage of  $\mathbf{L}$ ’s triangular structure. The generation of normal deviates is inline for efficiency, identical to `Normaldev` in §7.3.

```
struct Multinormaldev : Ran {
    Structure for multivariate normal deviates.
    Int mm;
    VecDoub mean;
    MatDoub var;
    Cholesky chol;
    VecDoub spt, pt;

    Multinormaldev(Ullong j, VecDoub &mmean, MatDoub &vvar) :
        Ran(j), mm(mmean.size()), mean(mmean), var(vvar), chol(var),
        spt(mm), pt(mm) {
        Constructor. Arguments are the random generator seed, the (vector) mean, and the (matrix) covariance. Cholesky decomposition of the covariance is done here.
        if (var.ncols() != mm || var.nrows() != mm) throw("bad sizes");
    }

    VecDoub &dev() {
        Return a multivariate normal deviate.
        Int i;
        Doub u,v,x,y,q;
        for (i=0;i<mm;i++) {                               Fill a vector of independent normal deviates.
            do {
                u = doub();
                v = 1.7156*(doub()-0.5);
                x = u - 0.449871;
                y = abs(v) + 0.386595;
                q = SQR(x) + y*(0.19600*y-0.25472*x);
            } while (q > 0.27597
                    && (q > 0.27846 || SQR(v) > -4.*log(u)*SQR(u)));
            spt[i] = v/u;
        }
        chol.elmult(spt,pt);                                Apply equation (7.4.3).
        for (i=0;i<mm;i++) {pt[i] += mean[i];}
        return pt;
    }

};
```

multinormaldev.h

### 7.4.1 Decorrelating Multiple Random Variables

Although not directly related to the generation of random deviates, this is a convenient place to point out how Cholesky decomposition can be used in the reverse manner, namely to find linear combinations of correlated random variables that have no correlation. In this application we are given a vector  $\mathbf{x}$  whose components have a known covariance  $\Sigma$  and mean  $\mu$ . Decomposing  $\Sigma$  as in equation (7.4.2), we assert

that

$$\mathbf{y} = \mathbf{L}^{-1}(\mathbf{x} - \boldsymbol{\mu}) \quad (7.4.7)$$

has uncorrelated components, each of unit variance. Proof:

$$\begin{aligned}\langle \mathbf{y} \otimes \mathbf{y} \rangle &= \langle (\mathbf{L}^{-1}[\mathbf{x} - \boldsymbol{\mu}]) \otimes (\mathbf{L}^{-1}[\mathbf{x} - \boldsymbol{\mu}]) \rangle \\ &= \mathbf{L}^{-1} \langle (\mathbf{x} - \boldsymbol{\mu}) \otimes (\mathbf{x} - \boldsymbol{\mu}) \rangle \mathbf{L}^{-1T} \\ &= \mathbf{L}^{-1} \boldsymbol{\Sigma} \mathbf{L}^{-1T} = \mathbf{L}^{-1} \mathbf{L} \mathbf{L}^T \mathbf{L}^{-1T} = \mathbf{1}\end{aligned} \quad (7.4.8)$$

Be aware that this linear combination is not unique. In fact, once you have obtained a vector  $\mathbf{y}$  of uncorrelated components, you can perform any rotation on it and still have uncorrelated components. In particular, if  $\mathbf{K}$  is an orthogonal matrix, so that

$$\mathbf{K}^T \mathbf{K} = \mathbf{K} \mathbf{K}^T = \mathbf{1} \quad (7.4.9)$$

then

$$\langle (\mathbf{K}\mathbf{y}) \otimes (\mathbf{K}\mathbf{y}) \rangle = \mathbf{K} \langle \mathbf{y} \otimes \mathbf{y} \rangle \mathbf{K}^T = \mathbf{K} \mathbf{K}^T = \mathbf{1} \quad (7.4.10)$$

A common (though slower) alternative to Cholesky decomposition is to use the Jacobi transformation (§11.1) to decompose  $\boldsymbol{\Sigma}$  as

$$\boldsymbol{\Sigma} = \mathbf{V} \text{diag}(\sigma_i^2) \mathbf{V}^T \quad (7.4.11)$$

where  $\mathbf{V}$  is the orthogonal matrix of eigenvectors, and the  $\sigma_i$ 's are the standard deviations of the (new) uncorrelated variables. Then  $\mathbf{V} \text{diag}(\sigma_i)$  plays the role of  $\mathbf{L}$  in the proofs above.

Section §16.1.1 discusses some further applications of Cholesky decomposition relating to multivariate random variables.

## 7.5 Linear Feedback Shift Registers

A *linear feedback shift register* (LFSR) consists of a *state vector* and a certain kind of *update rule*. The state vector is often the set of bits in a 32- or 64-bit word, but it can sometimes be a set of words in an array. To qualify as an LFSR, the update rule must generate a *linear* combination of the bits (or words) in the current state, and then shift that result onto one end of the state vector. The oldest value, at the other end of the state vector, falls off and is gone. The output of an LFSR consists of the sequence of new bits (or words) as they are shifted in.

For single bits, “linear” means arithmetic modulo 2, which is the same as using the logical XOR operation for  $+$  and the logical AND operation for  $\times$ . It is convenient, however, to write equations using the arithmetic notation. So, for an LFSR of length  $n$ , the words in the paragraph above translate to

$$\begin{aligned}a'_1 &= \left( \sum_{j=1}^{n-1} c_j a_j \right) + a_n \\ a'_i &= a_{i-1}, \quad i = 2, \dots, n\end{aligned} \quad (7.5.1)$$