Numerical Recipes in C

Second Edition

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 1 of 181

Numerical Recipes in C

The Art of Scientific Computing

Second Edition

William H. Press

Harvard-Smithsonian Center for Astrophysics

Saul A. Teukolsky

Department of Physics, Cornell University

William T. Vetterling

Polaroid Corporation

Brian P. Flannery

EXXON Research and Engineering Company

CAMBRIDGE UNIVERSITY PRESS

Cambridge New York Port Chester Melbourne Sydney

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 3 of 181

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC, 3207, Australia

Copyright © Cambridge University Press 1988, 1992
except for §13.10 and Appendix B, which are placed into the public domain,
and except for all other computer programs and procedures, which are
Copyright © Numerical Recipes Software 1987, 1988, 1992, 1997, 2002
All Rights Reserved.

Some sections of this book were originally published, in different form, in *Computers in Physics* magazine, Copyright © American Institute of Physics, 1988–1992.

First Edition originally published 1988; Second Edition originally published 1992.
Reprinted with corrections, 1993, 1994, 1995, 1997, 2002.
This reprinting is corrected to software version 2.10

Printed in the United States of America
Typeset in TEX

Without an additional license to use the contained software, this book is intended as a text and reference book, for reading purposes only. A free license for limited use of the software by the individual owner of a copy of this book who personally types one or more routines into a single computer is granted under terms described on p. xvii. See the section "License Information" (pp. xvi–xviii) for information on obtaining more general licenses at low cost.

Machine-readable media containing the software in this book, with included licenses for use on a single screen, are available from Cambridge University Press. See the order form at the back of the book, email to "orders@cup.org" (North America) or "directcustserv@cambridge.org" (rest of world), or write to Cambridge University Press, 110 Midland Avenue, Port Chester, NY 10573 (USA), for further information.

The software may also be downloaded, with immediate purchase of a license also possible, from the Numerical Recipes Software Web Site (`http://www.nr.com`). Unlicensed transfer of Numerical Recipes programs to any other format, or to any computer except one that is specifically licensed, is strictly prohibited. Technical questions, corrections, and requests for information should be addressed to Numerical Recipes Software, P.O. Box 380243, Cambridge, MA 02238-0243 (USA), email "info@nr.com", or fax 781 863-1739.

Library of Congress Cataloging in Publication Data

Numerical recipes in C : the art of scientific computing / William H. Press
. . . [et al.]. – 2nd ed.

Includes bibliographical references (p.) and index.

ISBN 0-521-43108-5

1. Numerical analysis–Computer programs. 2. Science–Mathematics–Computer programs.
3. C (Computer program language) I. Press, William H.
QA297.N866 1992
519.4′0285′53–dc20 92-8876

A catalog record for this book is available from the British Library.

ISBN 0 521 43108 5 Book
ISBN 0 521 43720 2 Example book in C
ISBN 0 521 75037 7 C/C++ CDROM (Windows/Macintosh)
ISBN 0 521 75035 0 Complete CDROM (Windows/Macintosh)
ISBN 0 521 75036 9 Complete CDROM (UNIX/Linux)

Contents

Preface to the Second Edition ***xi***

Preface to the First Edition ***xiv***

License Information ***xvi***

Computer Programs by Chapter and Section ***xix***

1 Preliminaries ***1***

1.0 Introduction 1
1.1 Program Organization and Control Structures 5
1.2 Some C Conventions for Scientific Computing 15
1.3 Error, Accuracy, and Stability 28

2 Solution of Linear Algebraic Equations ***32***

2.0 Introduction 32
2.1 Gauss-Jordan Elimination 36
2.2 Gaussian Elimination with Backsubstitution 41
2.3 LU Decomposition and Its Applications 43
2.4 Tridiagonal and Band Diagonal Systems of Equations 50
2.5 Iterative Improvement of a Solution to Linear Equations 55
2.6 Singular Value Decomposition 59
2.7 Sparse Linear Systems 71
2.8 Vandermonde Matrices and Toeplitz Matrices 90
2.9 Cholesky Decomposition 96
2.10 QR Decomposition 98
2.11 Is Matrix Inversion an N^3 Process? 102

3 Interpolation and Extrapolation ***105***

3.0 Introduction 105
3.1 Polynomial Interpolation and Extrapolation 108
3.2 Rational Function Interpolation and Extrapolation 111
3.3 Cubic Spline Interpolation 113
3.4 How to Search an Ordered Table 117
3.5 Coefficients of the Interpolating Polynomial 120
3.6 Interpolation in Two or More Dimensions 123

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 5 of 181

4 Integration of Functions 129

4.0 Introduction 129
4.1 Classical Formulas for Equally Spaced Abscissas 130
4.2 Elementary Algorithms 136
4.3 Romberg Integration 140
4.4 Improper Integrals 141
4.5 Gaussian Quadratures and Orthogonal Polynomials 147
4.6 Multidimensional Integrals 161

5 Evaluation of Functions 165

5.0 Introduction 165
5.1 Series and Their Convergence 165
5.2 Evaluation of Continued Fractions 169
5.3 Polynomials and Rational Functions 173
5.4 Complex Arithmetic 176
5.5 Recurrence Relations and Clenshaw's Recurrence Formula 178
5.6 Quadratic and Cubic Equations 183
5.7 Numerical Derivatives 186
5.8 Chebyshev Approximation 190
5.9 Derivatives or Integrals of a Chebyshev-approximated Function 195
5.10 Polynomial Approximation from Chebyshev Coefficients 197
5.11 Economization of Power Series 198
5.12 Padé Approximants 200
5.13 Rational Chebyshev Approximation 204
5.14 Evaluation of Functions by Path Integration 208

6 Special Functions 212

6.0 Introduction 212
6.1 Gamma Function, Beta Function, Factorials, Binomial Coefficients 213
6.2 Incomplete Gamma Function, Error Function, Chi-Square Probability Function, Cumulative Poisson Function 216
6.3 Exponential Integrals 222
6.4 Incomplete Beta Function, Student's Distribution, F-Distribution, Cumulative Binomial Distribution 226
6.5 Bessel Functions of Integer Order 230
6.6 Modified Bessel Functions of Integer Order 236
6.7 Bessel Functions of Fractional Order, Airy Functions, Spherical Bessel Functions 240
6.8 Spherical Harmonics 252
6.9 Fresnel Integrals, Cosine and Sine Integrals 255
6.10 Dawson's Integral 259
6.11 Elliptic Integrals and Jacobian Elliptic Functions 261
6.12 Hypergeometric Functions 271

7 Random Numbers 274

7.0 Introduction 274
7.1 Uniform Deviates 275

7.2 Transformation Method: Exponential and Normal Deviates 287
7.3 Rejection Method: Gamma, Poisson, Binomial Deviates 290
7.4 Generation of Random Bits 296
7.5 Random Sequences Based on Data Encryption 300
7.6 Simple Monte Carlo Integration 304
7.7 Quasi- (that is, Sub-) Random Sequences 309
7.8 Adaptive and Recursive Monte Carlo Methods 316

8 Sorting 329

8.0 Introduction 329
8.1 Straight Insertion and Shell's Method 330
8.2 Quicksort 332
8.3 Heapsort 336
8.4 Indexing and Ranking 338
8.5 Selecting the Mth Largest 341
8.6 Determination of Equivalence Classes 345

9 Root Finding and Nonlinear Sets of Equations 347

9.0 Introduction 347
9.1 Bracketing and Bisection 350
9.2 Secant Method, False Position Method, and Ridders' Method 354
9.3 Van Wijngaarden–Dekker–Brent Method 359
9.4 Newton-Raphson Method Using Derivative 362
9.5 Roots of Polynomials 369
9.6 Newton-Raphson Method for Nonlinear Systems of Equations 379
9.7 Globally Convergent Methods for Nonlinear Systems of Equations 383

10 Minimization or Maximization of Functions 394

10.0 Introduction 394
10.1 Golden Section Search in One Dimension 397
10.2 Parabolic Interpolation and Brent's Method in One Dimension 402
10.3 One-Dimensional Search with First Derivatives 405
10.4 Downhill Simplex Method in Multidimensions 408
10.5 Direction Set (Powell's) Methods in Multidimensions 412
10.6 Conjugate Gradient Methods in Multidimensions 420
10.7 Variable Metric Methods in Multidimensions 425
10.8 Linear Programming and the Simplex Method 430
10.9 Simulated Annealing Methods 444

11 Eigensystems 456

11.0 Introduction 456
11.1 Jacobi Transformations of a Symmetric Matrix 463
11.2 Reduction of a Symmetric Matrix to Tridiagonal Form: Givens and Householder Reductions 469
11.3 Eigenvalues and Eigenvectors of a Tridiagonal Matrix 475
11.4 Hermitian Matrices 481
11.5 Reduction of a General Matrix to Hessenberg Form 482

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 7 of 181

11.6 The QR Algorithm for Real Hessenberg Matrices 486
11.7 Improving Eigenvalues and/or Finding Eigenvectors by Inverse Iteration 493

12 Fast Fourier Transform 496

12.0 Introduction 496
12.1 Fourier Transform of Discretely Sampled Data 500
12.2 Fast Fourier Transform (FFT) 504
12.3 FFT of Real Functions, Sine and Cosine Transforms 510
12.4 FFT in Two or More Dimensions 521
12.5 Fourier Transforms of Real Data in Two and Three Dimensions 525
12.6 External Storage or Memory-Local FFTs 532

13 Fourier and Spectral Applications 537

13.0 Introduction 537
13.1 Convolution and Deconvolution Using the FFT 538
13.2 Correlation and Autocorrelation Using the FFT 545
13.3 Optimal (Wiener) Filtering with the FFT 547
13.4 Power Spectrum Estimation Using the FFT 549
13.5 Digital Filtering in the Time Domain 558
13.6 Linear Prediction and Linear Predictive Coding 564
13.7 Power Spectrum Estimation by the Maximum Entropy (All Poles) Method 572
13.8 Spectral Analysis of Unevenly Sampled Data 575
13.9 Computing Fourier Integrals Using the FFT 584
13.10 Wavelet Transforms 591
13.11 Numerical Use of the Sampling Theorem 606

14 Statistical Description of Data 609

14.0 Introduction 609
14.1 Moments of a Distribution: Mean, Variance, Skewness, and So Forth 610
14.2 Do Two Distributions Have the Same Means or Variances? 615
14.3 Are Two Distributions Different? 620
14.4 Contingency Table Analysis of Two Distributions 628
14.5 Linear Correlation 636
14.6 Nonparametric or Rank Correlation 639
14.7 Do Two-Dimensional Distributions Differ? 645
14.8 Savitzky-Golay Smoothing Filters 650

15 Modeling of Data 656

15.0 Introduction 656
15.1 Least Squares as a Maximum Likelihood Estimator 657
15.2 Fitting Data to a Straight Line 661
15.3 Straight-Line Data with Errors in Both Coordinates 666
15.4 General Linear Least Squares 671
15.5 Nonlinear Models 681

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 8 of 181

15.6 Confidence Limits on Estimated Model Parameters 689
15.7 Robust Estimation 699

16 Integration of Ordinary Differential Equations 707

16.0 Introduction 707
16.1 Runge-Kutta Method 710
16.2 Adaptive Stepsize Control for Runge-Kutta 714
16.3 Modified Midpoint Method 722
16.4 Richardson Extrapolation and the Bulirsch-Stoer Method 724
16.5 Second-Order Conservative Equations 732
16.6 Stiff Sets of Equations 734
16.7 Multistep, Multivalue, and Predictor-Corrector Methods 747

17 Two Point Boundary Value Problems 753

17.0 Introduction 753
17.1 The Shooting Method 757
17.2 Shooting to a Fitting Point 760
17.3 Relaxation Methods 762
17.4 A Worked Example: Spheroidal Harmonics 772
17.5 Automated Allocation of Mesh Points 783
17.6 Handling Internal Boundary Conditions or Singular Points 784

18 Integral Equations and Inverse Theory 788

18.0 Introduction 788
18.1 Fredholm Equations of the Second Kind 791
18.2 Volterra Equations 794
18.3 Integral Equations with Singular Kernels 797
18.4 Inverse Problems and the Use of A Priori Information 804
18.5 Linear Regularization Methods 808
18.6 Backus-Gilbert Method 815
18.7 Maximum Entropy Image Restoration 818

19 Partial Differential Equations 827

19.0 Introduction 827
19.1 Flux-Conservative Initial Value Problems 834
19.2 Diffusive Initial Value Problems 847
19.3 Initial Value Problems in Multidimensions 853
19.4 Fourier and Cyclic Reduction Methods for Boundary Value Problems 857
19.5 Relaxation Methods for Boundary Value Problems 863
19.6 Multigrid Methods for Boundary Value Problems 871

20 Less-Numerical Algorithms 889

20.0 Introduction 889
20.1 Diagnosing Machine Parameters 889
20.2 Gray Codes 894

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 9 of 181

20.3 Cyclic Redundancy and Other Checksums 896
20.4 Huffman Coding and Compression of Data 903
20.5 Arithmetic Coding 910
20.6 Arithmetic at Arbitrary Precision 915

References ***926***

Appendix A: Table of Prototype Declarations ***930***

Appendix B: Utility Routines ***940***

Appendix C: Complex Arithmetic ***948***

Index of Programs and Dependencies ***951***

General Index ***965***

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 10 of 181

Preface to the Second Edition

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 both in 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

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 11 of 181

- simulated annealing minimization for continuous control spaces
- fast Fourier transform (FFT) for real data in two and three dimensions
- fast Fourier transform (FFT) 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 Table of 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 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. Deuflhard, 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 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. (See §1.0 for a list of additional computers used in

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 12 of 181

program tests.) We enthusiastically recommend the principal software used: GNU Emacs, TEX, Perl, Adobe Illustrator, and PostScript. Also used were a variety of C compilers – too numerous (and sometimes too buggy) for individual acknowledgment. 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. D.A.R.P.A. support is acknowledged for §13.10 on wavelets.

June, 1992

William H. Press
Saul A. Teukolsky
William T. Vetterling
Brian P. Flannery

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 13 of 181

Preface to the First Edition

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 towards more in-depth discussion of the nature of the computational algorithms, or towards 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 (Chapter 19). Second, we obviously cannot include *everything* else. All the so-called "standard" topics of a numerical analysis course have been included in this book:

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 14 of 181

linear equations (Chapter 2), interpolation and extrapolation (Chaper 3), integration (Chaper 4), nonlinear root-finding (Chapter 9), eigensystems (Chapter 11), and ordinary differential equations (Chapter 16). Most of these topics have been taken beyond their standard treatments into some advanced material which 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 (Chapters 5 and 6); random numbers and Monte Carlo methods (Chapter 7); sorting (Chapter 8); optimization, including multidimensional methods (Chapter 10); Fourier transform methods, including FFT methods and other spectral methods (Chapters 12 and 13); two chapters on the statistical description and modeling of data (Chapters 14 and 15); and two-point boundary value problems, both shooting and relaxation methods (Chapter 17).

The programs in this book are included in ANSI-standard `C`. Versions of the book in `FORTRAN`, `Pascal`, and `BASIC` are available separately. We have more to say about the `C` language, and the computational environment assumed by our routines, in §1.1 (Introduction).

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 in 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 which immensely aided production of this book.

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

October, 1985

William H. Press
Brian P. Flannery
Saul A. Teukolsky
William T. Vetterling

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 15 of 181

License Information

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

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.

How to Get the Code onto Your Computer

Pick one of the following methods:

- You can type the programs from this book directly into your computer. In this case, the *only* kind of license available to you is the free "immediate license" (see below). You are not authorized to transfer or distribute a machine-readable copy to any other person, nor to have any other person type the programs into a computer on your behalf. We do not want to hear bug reports from you if you choose this option, because experience has shown that *virtually all* reported bugs in such cases are typing errors!
- You can download the Numerical Recipes programs electronically from the Numerical Recipes On-Line Software Store, located at `http://www.nr.com`, our Web site. All the files (Recipes and demonstration programs) are packaged as a single compressed file. You'll need to purchase a license to download and unpack them. Any number of single-screen licenses can be purchased instantly (with discount for multiple screens) from the On-Line Store, with fees that depend on your operating system (Windows or Macintosh versus Linux or UNIX) and whether you are affiliated with an educational institution. Purchasing a single-screen license is also the way to start if you want to acquire a more general (site or corporate) license; your single-screen cost will be subtracted from the cost of any later license upgrade.
- You can purchase media containing the programs from Cambridge University Press. A CD-ROM version in ISO-9660 format for Windows and Macintosh systems contains the complete C software, and also the C++ version. More extensive CD-ROMs in ISO-9660 format for Windows, Macintosh, and UNIX/Linux systems are also available; these include the C, C++, and Fortran versions on a single CD-ROM (as well as versions in Pascal and BASIC from the first edition). These CD-ROMs are available with a single-screen license for Windows or Macintosh (order ISBN 0 521 750350), or (at a slightly higher price) with a single-screen license for UNIX/Linux workstations (order ISBN 0 521 750369). Orders for media from Cambridge University Press can be placed at 800 872-7423 (North America only) or by email to orders@cup.org (North America) or directcustserv@cambridge.org (rest of world). Or, visit the Web site `http://www.cambridge.org`.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 16 of 181

Types of License Offered

Here are the types of licenses that we offer. Note that some types are automatically acquired with the purchase of media from Cambridge University Press, or of an unlocking password from the Numerical Recipes On-Line Software Store, while other types of licenses require that you communicate specifically with Numerical Recipes Software (email: orders@nr.com or fax: 781 863-1739). Our Web site `http://www.nr.com` has additional information.

- [“Immediate License”] If you are the individual owner of a copy of this book and you type one or more of its routines into your computer, we authorize you to use them on that computer for your own personal and noncommercial purposes. You are not authorized to transfer or distribute machine-readable copies to any other person, or to use the routines on more than one machine, or to distribute executable programs containing our routines. This is the only free license.
- [“Single-Screen License”] This is the most common type of low-cost license, with terms governed by our Single Screen (Shrinkwrap) License document (complete terms available through our Web site). Basically, this license lets you use Numerical Recipes routines on any one screen (PC, workstation, X-terminal, etc.). You may also, under this license, transfer pre-compiled, executable programs incorporating our routines to other, unlicensed, screens 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 on a licensed screen, 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. Conditions for other types of commercial or noncommercial distribution may be found on our Web site (`http://www.nr.com`).
- [“Multi-Screen, Server, Site, and Corporate Licenses”] The terms of the Single Screen License can be extended to designated groups of machines, defined by number of screens, number of machines, locations, or ownership. Significant discounts from the corresponding single-screen prices are available when the estimated number of screens exceeds 40. Contact Numerical Recipes Software (email: orders@nr.com or fax: 781 863-1739) for details.
- [“Course Right-to-Copy License”] Instructors at accredited educational institutions who have adopted this book for a course, and who have already purchased a Single Screen License (either acquired with the purchase of media, or from the Numerical Recipes On-Line Software Store), may license the programs for use in that course as follows: Mail your name, title, and address; the course name, number, dates, and estimated enrollment; and advance payment of $5 per (estimated) student to Numerical Recipes Software, at this address: P.O. Box 243, Cambridge, MA 02238 (USA). You will receive by return mail a license authorizing you to make copies of the programs for use by your students, and/or to transfer the programs to a machine accessible to your students (but only for the duration of the course).

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

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 17 of 181

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: Sun is a trademark of Sun Microsystems, Inc. SPARC and SPARCstation are trademarks of SPARC International, Inc. Microsoft, Windows 95, Windows NT, PowerStation, and MS are trademarks of Microsoft Corporation. DEC, VMS, Alpha AXP, and ULTRIX are trademarks of Digital Equipment Corporation. IBM is a trademark of International Business Machines Corporation. Apple and Macintosh are trademarks of Apple Computer, Inc. UNIX is a trademark licensed exclusively through X/Open Co. Ltd. IMSL is a trademark of Visual Numerics, Inc. NAG refers to proprietary computer software of Numerical Algorithms Group (USA) Inc. PostScript and Adobe Illustrator are trademarks of Adobe Systems Incorporated. Last, and no doubt least, Numerical Recipes (when identifying products) is a trademark 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.

Computer Programs by Chapter and Section

1.0	`flmoon`	calculate phases of the moon by date
1.1	`julday`	Julian Day number from calendar date
1.1	`badluk`	Friday the 13th when the moon is full
1.1	`caldat`	calendar date from Julian day number
2.1	`gaussj`	Gauss-Jordan matrix inversion and linear equation solution
2.3	`ludcmp`	linear equation solution, LU decomposition
2.3	`lubksb`	linear equation solution, backsubstitution
2.4	`tridag`	solution of tridiagonal systems
2.4	`banmul`	multiply vector by band diagonal matrix
2.4	`bandec`	band diagonal systems, decomposition
2.4	`banbks`	band diagonal systems, backsubstitution
2.5	`mprove`	linear equation solution, iterative improvement
2.6	`svbksb`	singular value backsubstitution
2.6	`svdcmp`	singular value decomposition of a matrix
2.6	`pythag`	calculate $(a^2 + b^2)^{1/2}$ without overflow
2.7	`cyclic`	solution of cyclic tridiagonal systems
2.7	`sprsin`	convert matrix to sparse format
2.7	`sprsax`	product of sparse matrix and vector
2.7	`sprstx`	product of transpose sparse matrix and vector
2.7	`sprstp`	transpose of sparse matrix
2.7	`sprspm`	pattern multiply two sparse matrices
2.7	`sprstm`	threshold multiply two sparse matrices
2.7	`linbcg`	biconjugate gradient solution of sparse systems
2.7	`snrm`	used by `linbcg` for vector norm
2.7	`atimes`	used by `linbcg` for sparse multiplication
2.7	`asolve`	used by `linbcg` for preconditioner
2.8	`vander`	solve Vandermonde systems
2.8	`toeplz`	solve Toeplitz systems
2.9	`choldc`	Cholesky decomposition
2.9	`cholsl`	Cholesky backsubstitution
2.10	`qrdcmp`	QR decomposition
2.10	`qrsolv`	QR backsubstitution
2.10	`rsolv`	right triangular backsubstitution
2.10	`qrupdt`	update a QR decomposition
2.10	`rotate`	Jacobi rotation used by `qrupdt`
3.1	`polint`	polynomial interpolation
3.2	`ratint`	rational function interpolation
3.3	`spline`	construct a cubic spline
3.3	`splint`	cubic spline interpolation
3.4	`locate`	search an ordered table by bisection

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 19 of 181

3.4	`hunt`	search a table when calls are correlated
3.5	`polcoe`	polynomial coefficients from table of values
3.5	`polcof`	polynomial coefficients from table of values
3.6	`polin2`	two-dimensional polynomial interpolation
3.6	`bcucof`	construct two-dimensional bicubic
3.6	`bcuint`	two-dimensional bicubic interpolation
3.6	`splie2`	construct two-dimensional spline
3.6	`splin2`	two-dimensional spline interpolation
4.2	`trapzd`	trapezoidal rule
4.2	`qtrap`	integrate using trapezoidal rule
4.2	`qsimp`	integrate using Simpson's rule
4.3	`qromb`	integrate using Romberg adaptive method
4.4	`midpnt`	extended midpoint rule
4.4	`qromo`	integrate using open Romberg adaptive method
4.4	`midinf`	integrate a function on a semi-infinite interval
4.4	`midsql`	integrate a function with lower square-root singularity
4.4	`midsqu`	integrate a function with upper square-root singularity
4.4	`midexp`	integrate a function that decreases exponentially
4.5	`qgaus`	integrate a function by Gaussian quadratures
4.5	`gauleg`	Gauss-Legendre weights and abscissas
4.5	`gaulag`	Gauss-Laguerre weights and abscissas
4.5	`gauher`	Gauss-Hermite weights and abscissas
4.5	`gaujac`	Gauss-Jacobi weights and abscissas
4.5	`gaucof`	quadrature weights from orthogonal polynomials
4.5	`orthog`	construct nonclassical orthogonal polynomials
4.6	`quad3d`	integrate a function over a three-dimensional space
5.1	`eulsum`	sum a series by Euler–van Wijngaarden algorithm
5.3	`ddpoly`	evaluate a polynomial and its derivatives
5.3	`poldiv`	divide one polynomial by another
5.3	`ratval`	evaluate a rational function
5.7	`dfridr`	numerical derivative by Ridders' method
5.8	`chebft`	fit a Chebyshev polynomial to a function
5.8	`chebev`	Chebyshev polynomial evaluation
5.9	`chder`	derivative of a function already Chebyshev fitted
5.9	`chint`	integrate a function already Chebyshev fitted
5.10	`chebpc`	polynomial coefficients from a Chebyshev fit
5.10	`pcshft`	polynomial coefficients of a shifted polynomial
5.11	`pccheb`	inverse of `chebpc`; use to economize power series
5.12	`pade`	Padé approximant from power series coefficients
5.13	`ratlsq`	rational fit by least-squares method
6.1	`gammln`	logarithm of gamma function
6.1	`factrl`	factorial function
6.1	`bico`	binomial coefficients function
6.1	`factln`	logarithm of factorial function

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 20 of 181

6.1	`beta`	beta function
6.2	`gammp`	incomplete gamma function
6.2	`gammq`	complement of incomplete gamma function
6.2	`gser`	series used by `gammp` and `gammq`
6.2	`gcf`	continued fraction used by `gammp` and `gammq`
6.2	`erff`	error function
6.2	`erffc`	complementary error function
6.2	`erfcc`	complementary error function, concise routine
6.3	`expint`	exponential integral E_n
6.3	`ei`	exponential integral Ei
6.4	`betai`	incomplete beta function
6.4	`betacf`	continued fraction used by `betai`
6.5	`bessj0`	Bessel function J_0
6.5	`bessy0`	Bessel function Y_0
6.5	`bessj1`	Bessel function J_1
6.5	`bessy1`	Bessel function Y_1
6.5	`bessy`	Bessel function Y of general integer order
6.5	`bessj`	Bessel function J of general integer order
6.6	`bessi0`	modified Bessel function I_0
6.6	`bessk0`	modified Bessel function K_0
6.6	`bessi1`	modified Bessel function I_1
6.6	`bessk1`	modified Bessel function K_1
6.6	`bessk`	modified Bessel function K of integer order
6.6	`bessi`	modified Bessel function I of integer order
6.7	`bessjy`	Bessel functions of fractional order
6.7	`beschb`	Chebyshev expansion used by `bessjy`
6.7	`bessik`	modified Bessel functions of fractional order
6.7	`airy`	Airy functions
6.7	`sphbes`	spherical Bessel functions j_n and y_n
6.8	`plgndr`	Legendre polynomials, associated (spherical harmonics)
6.9	`frenel`	Fresnel integrals $S(x)$ and $C(x)$
6.9	`cisi`	cosine and sine integrals Ci and Si
6.10	`dawson`	Dawson's integral
6.11	`rf`	Carlson's elliptic integral of the first kind
6.11	`rd`	Carlson's elliptic integral of the second kind
6.11	`rj`	Carlson's elliptic integral of the third kind
6.11	`rc`	Carlson's degenerate elliptic integral
6.11	`ellf`	Legendre elliptic integral of the first kind
6.11	`elle`	Legendre elliptic integral of the second kind
6.11	`ellpi`	Legendre elliptic integral of the third kind
6.11	`sncndn`	Jacobian elliptic functions
6.12	`hypgeo`	complex hypergeometric function
6.12	`hypser`	complex hypergeometric function, series evaluation
6.12	`hypdrv`	complex hypergeometric function, derivative of
7.1	`ran0`	random deviate by Park and Miller minimal standard
7.1	`ran1`	random deviate, minimal standard plus shuffle

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 21 of 181

7.1	`ran2`	random deviate by L'Ecuyer long period plus shuffle
7.1	`ran3`	random deviate by Knuth subtractive method
7.2	`expdev`	exponential random deviates
7.2	`gasdev`	normally distributed random deviates
7.3	`gamdev`	gamma-law distribution random deviates
7.3	`poidev`	Poisson distributed random deviates
7.3	`bnldev`	binomial distributed random deviates
7.4	`irbit1`	random bit sequence
7.4	`irbit2`	random bit sequence
7.5	`psdes`	"pseudo-DES" hashing of 64 bits
7.5	`ran4`	random deviates from DES-like hashing
7.7	`sobseq`	Sobol's quasi-random sequence
7.8	`vegas`	adaptive multidimensional Monte Carlo integration
7.8	`rebin`	sample rebinning used by `vegas`
7.8	`miser`	recursive multidimensional Monte Carlo integration
7.8	`ranpt`	get random point, used by `miser`
8.1	`piksrt`	sort an array by straight insertion
8.1	`piksr2`	sort two arrays by straight insertion
8.1	`shell`	sort an array by Shell's method
8.2	`sort`	sort an array by quicksort method
8.2	`sort2`	sort two arrays by quicksort method
8.3	`hpsort`	sort an array by heapsort method
8.4	`indexx`	construct an index for an array
8.4	`sort3`	sort, use an index to sort 3 or more arrays
8.4	`rank`	construct a rank table for an array
8.5	`select`	find the Nth largest in an array
8.5	`selip`	find the Nth largest, without altering an array
8.5	`hpsel`	find M largest values, without altering an array
8.6	`eclass`	determine equivalence classes from list
8.6	`eclazz`	determine equivalence classes from procedure
9.0	`scrsho`	graph a function to search for roots
9.1	`zbrac`	outward search for brackets on roots
9.1	`zbrak`	inward search for brackets on roots
9.1	`rtbis`	find root of a function by bisection
9.2	`rtflsp`	find root of a function by false-position
9.2	`rtsec`	find root of a function by secant method
9.2	`zriddr`	find root of a function by Ridders' method
9.3	`zbrent`	find root of a function by Brent's method
9.4	`rtnewt`	find root of a function by Newton-Raphson
9.4	`rtsafe`	find root of a function by Newton-Raphson and bisection
9.5	`laguer`	find a root of a polynomial by Laguerre's method
9.5	`zroots`	roots of a polynomial by Laguerre's method with deflation
9.5	`zrhqr`	roots of a polynomial by eigenvalue methods
9.5	`qroot`	complex or double root of a polynomial, Bairstow

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 22 of 181

9.6	`mnewt`	Newton's method for systems of equations
9.7	`lnsrch`	search along a line, used by `newt`
9.7	`newt`	globally convergent multi-dimensional Newton's method
9.7	`fdjac`	finite-difference Jacobian, used by `newt`
9.7	`fmin`	norm of a vector function, used by `newt`
9.7	`broydn`	secant method for systems of equations
10.1	`mnbrak`	bracket the minimum of a function
10.1	`golden`	find minimum of a function by golden section search
10.2	`brent`	find minimum of a function by Brent's method
10.3	`dbrent`	find minimum of a function using derivative information
10.4	`amoeba`	minimize in N-dimensions by downhill simplex method
10.4	`amotry`	evaluate a trial point, used by `amoeba`
10.5	`powell`	minimize in N-dimensions by Powell's method
10.5	`linmin`	minimum of a function along a ray in N-dimensions
10.5	`f1dim`	function used by `linmin`
10.6	`frprmn`	minimize in N-dimensions by conjugate gradient
10.6	`dlinmin`	minimum of a function along a ray using derivatives
10.6	`df1dim`	function used by `dlinmin`
10.7	`dfpmin`	minimize in N-dimensions by variable metric method
10.8	`simplx`	linear programming maximization of a linear function
10.8	`simp1`	linear programming, used by `simplx`
10.8	`simp2`	linear programming, used by `simplx`
10.8	`simp3`	linear programming, used by `simplx`
10.9	`anneal`	traveling salesman problem by simulated annealing
10.9	`revcst`	cost of a reversal, used by `anneal`
10.9	`reverse`	do a reversal, used by `anneal`
10.9	`trncst`	cost of a transposition, used by `anneal`
10.9	`trnspt`	do a transposition, used by `anneal`
10.9	`metrop`	Metropolis algorithm, used by `anneal`
10.9	`amebsa`	simulated annealing in continuous spaces
10.9	`amotsa`	evaluate a trial point, used by `amebsa`
11.1	`jacobi`	eigenvalues and eigenvectors of a symmetric matrix
11.1	`eigsrt`	eigenvectors, sorts into order by eigenvalue
11.2	`tred2`	Householder reduction of a real, symmetric matrix
11.3	`tqli`	eigensolution of a symmetric tridiagonal matrix
11.5	`balanc`	balance a nonsymmetric matrix
11.5	`elmhes`	reduce a general matrix to Hessenberg form
11.6	`hqr`	eigenvalues of a Hessenberg matrix
12.2	`four1`	fast Fourier transform (FFT) in one dimension
12.3	`twofft`	fast Fourier transform of two real functions
12.3	`realft`	fast Fourier transform of a single real function
12.3	`sinft`	fast sine transform
12.3	`cosft1`	fast cosine transform with endpoints
12.3	`cosft2`	"staggered" fast cosine transform

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 23 of 181

12.4	`fourn`	fast Fourier transform in multidimensions
12.5	`rlft3`	FFT of real data in two or three dimensions
12.6	`fourfs`	FFT for huge data sets on external media
12.6	`fourew`	rewind and permute files, used by `fourfs`
13.1	`convlv`	convolution or deconvolution of data using FFT
13.2	`correl`	correlation or autocorrelation of data using FFT
13.4	`spctrm`	power spectrum estimation using FFT
13.6	`memcof`	evaluate maximum entropy (MEM) coefficients
13.6	`fixrts`	reflect roots of a polynomial into unit circle
13.6	`predic`	linear prediction using MEM coefficients
13.7	`evlmem`	power spectral estimation from MEM coefficients
13.8	`period`	power spectrum of unevenly sampled data
13.8	`fasper`	power spectrum of unevenly sampled larger data sets
13.8	`spread`	extirpolate value into array, used by `fasper`
13.9	`dftcor`	compute endpoint corrections for Fourier integrals
13.9	`dftint`	high-accuracy Fourier integrals
13.10	`wt1`	one-dimensional discrete wavelet transform
13.10	`daub4`	Daubechies 4-coefficient wavelet filter
13.10	`pwtset`	initialize coefficients for `pwt`
13.10	`pwt`	partial wavelet transform
13.10	`wtn`	multidimensional discrete wavelet transform
14.1	`moment`	calculate moments of a data set
14.2	`ttest`	Student's t-test for difference of means
14.2	`avevar`	calculate mean and variance of a data set
14.2	`tutest`	Student's t-test for means, case of unequal variances
14.2	`tptest`	Student's t-test for means, case of paired data
14.2	`ftest`	F-test for difference of variances
14.3	`chsone`	chi-square test for difference between data and model
14.3	`chstwo`	chi-square test for difference between two data sets
14.3	`ksone`	Kolmogorov-Smirnov test of data against model
14.3	`kstwo`	Kolmogorov-Smirnov test between two data sets
14.3	`probks`	Kolmogorov-Smirnov probability function
14.4	`cntab1`	contingency table analysis using chi-square
14.4	`cntab2`	contingency table analysis using entropy measure
14.5	`pearsn`	Pearson's correlation between two data sets
14.6	`spear`	Spearman's rank correlation between two data sets
14.6	`crank`	replaces array elements by their rank
14.6	`kendl1`	correlation between two data sets, Kendall's tau
14.6	`kendl2`	contingency table analysis using Kendall's tau
14.7	`ks2d1s`	K–S test in two dimensions, data vs. model
14.7	`quadct`	count points by quadrants, used by `ks2d1s`
14.7	`quadvl`	quadrant probabilities, used by `ks2d1s`
14.7	`ks2d2s`	K–S test in two dimensions, data vs. data
14.8	`savgol`	Savitzky-Golay smoothing coefficients

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 24 of 181

15.2	`fit`	least-squares fit data to a straight line
15.3	`fitexy`	fit data to a straight line, errors in both x and y
15.3	`chixy`	used by `fitexy` to calculate a χ^2
15.4	`lfit`	general linear least-squares fit by normal equations
15.4	`covsrt`	rearrange covariance matrix, used by `lfit`
15.4	`svdfit`	linear least-squares fit by singular value decomposition
15.4	`svdvar`	variances from singular value decomposition
15.4	`fpoly`	fit a polynomial using `lfit` or `svdfit`
15.4	`fleg`	fit a Legendre polynomial using `lfit` or `svdfit`
15.5	`mrqmin`	nonlinear least-squares fit, Marquardt's method
15.5	`mrqcof`	used by `mrqmin` to evaluate coefficients
15.5	`fgauss`	fit a sum of Gaussians using `mrqmin`
15.7	`medfit`	fit data to a straight line robustly, least absolute deviation
15.7	`rofunc`	fit data robustly, used by `medfit`
16.1	`rk4`	integrate one step of ODEs, fourth-order Runge-Kutta
16.1	`rkdumb`	integrate ODEs by fourth-order Runge-Kutta
16.2	`rkqs`	integrate one step of ODEs with accuracy monitoring
16.2	`rkck`	Cash-Karp-Runge-Kutta step used by `rkqs`
16.2	`odeint`	integrate ODEs with accuracy monitoring
16.3	`mmid`	integrate ODEs by modified midpoint method
16.4	`bsstep`	integrate ODEs, Bulirsch-Stoer step
16.4	`pzextr`	polynomial extrapolation, used by `bsstep`
16.4	`rzextr`	rational function extrapolation, used by `bsstep`
16.5	`stoerm`	integrate conservative second-order ODEs
16.6	`stiff`	integrate stiff ODEs by fourth-order Rosenbrock
16.6	`jacobn`	sample Jacobian routine for `stiff`
16.6	`derivs`	sample derivatives routine for `stiff`
16.6	`simpr`	integrate stiff ODEs by semi-implicit midpoint rule
16.6	`stifbs`	integrate stiff ODEs, Bulirsch-Stoer step
17.1	`shoot`	solve two point boundary value problem by shooting
17.2	`shootf`	ditto, by shooting to a fitting point
17.3	`solvde`	two point boundary value problem, solve by relaxation
17.3	`bksub`	backsubstitution, used by `solvde`
17.3	`pinvs`	diagonalize a sub-block, used by `solvde`
17.3	`red`	reduce columns of a matrix, used by `solvde`
17.4	`sfroid`	spheroidal functions by method of `solvde`
17.4	`difeq`	spheroidal matrix coefficients, used by `sfroid`
17.4	`sphoot`	spheroidal functions by method of `shoot`
17.4	`sphfpt`	spheroidal functions by method of `shootf`
18.1	`fred2`	solve linear Fredholm equations of the second kind
18.1	`fredin`	interpolate solutions obtained with `fred2`
18.2	`voltra`	linear Volterra equations of the second kind
18.3	`wwghts`	quadrature weights for an arbitrarily singular kernel
18.3	`kermom`	sample routine for moments of a singular kernel

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 25 of 181

18.3	`quadmx`	sample routine for a quadrature matrix
18.3	`fredex`	example of solving a singular Fredholm equation
19.5	`sor`	elliptic PDE solved by successive overrelaxation method
19.6	`mglin`	linear elliptic PDE solved by multigrid method
19.6	`rstrct`	half-weighting restriction, used by `mglin`, `mgfas`
19.6	`interp`	bilinear prolongation, used by `mglin`, `mgfas`
19.6	`addint`	interpolate and add, used by `mglin`
19.6	`slvsml`	solve on coarsest grid, used by `mglin`
19.6	`relax`	Gauss-Seidel relaxation, used by `mglin`
19.6	`resid`	calculate residual, used by `mglin`
19.6	`copy`	utility used by `mglin`, `mgfas`
19.6	`fill0`	utility used by `mglin`
19.6	`mgfas`	nonlinear elliptic PDE solved by multigrid method
19.6	`relax2`	Gauss-Seidel relaxation, used by `mgfas`
19.6	`slvsm2`	solve on coarsest grid, used by `mgfas`
19.6	`lop`	applies nonlinear operator, used by `mgfas`
19.6	`matadd`	utility used by `mgfas`
19.6	`matsub`	utility used by `mgfas`
19.6	`anorm2`	utility used by `mgfas`
20.1	`machar`	diagnose computer's floating arithmetic
20.2	`igray`	Gray code and its inverse
20.3	`icrc1`	cyclic redundancy checksum, used by `icrc`
20.3	`icrc`	cyclic redundancy checksum
20.3	`decchk`	decimal check digit calculation or verification
20.4	`hufmak`	construct a Huffman code
20.4	`hufapp`	append bits to a Huffman code, used by `hufmak`
20.4	`hufenc`	use Huffman code to encode and compress a character
20.4	`hufdec`	use Huffman code to decode and decompress a character
20.5	`arcmak`	construct an arithmetic code
20.5	`arcode`	encode or decode a character using arithmetic coding
20.5	`arcsum`	add integer to byte string, used by `arcode`
20.6	`mpops`	multiple precision arithmetic, simpler operations
20.6	`mpmul`	multiple precision multiply, using FFT methods
20.6	`mpinv`	multiple precision reciprocal
20.6	`mpdiv`	multiple precision divide and remainder
20.6	`mpsqrt`	multiple precision square root
20.6	`mp2dfr`	multiple precision conversion to decimal base
20.6	`mppi`	multiple precision example, compute many digits of π

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 26 of 181

Chapter 9. Root Finding and Nonlinear Sets of Equations

9.0 Introduction

We now consider that most basic of tasks, solving equations numerically. While most equations are born with both a right-hand side and a left-hand side, one traditionally moves all terms to the left, leaving

$$f(x) = 0 \tag{9.0.1}$$

whose solution or solutions are desired. When there is only one independent variable, the problem is *one-dimensional*, namely to find the root or roots of a function.

With more than one independent variable, more than one equation can be satisfied simultaneously. You likely once learned the *implicit function theorem* which (in this context) gives us the hope of satisfying N equations in N unknowns simultaneously. Note that we have only hope, not certainty. A nonlinear set of equations may have no (real) solutions at all. Contrariwise, it may have more than one solution. The implicit function theorem tells us that "generically" the solutions will be distinct, pointlike, and separated from each other. If, however, life is so unkind as to present you with a nongeneric, i.e., degenerate, case, then you can get a continuous family of solutions. In vector notation, we want to find one or more N-dimensional solution vectors $\mathbf{x}$ such that

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{9.0.2}$$

where $\mathbf{f}$ is the N-dimensional vector-valued function whose components are the individual equations to be satisfied simultaneously.

Don't be fooled by the apparent notational similarity of equations (9.0.2) and (9.0.1). Simultaneous solution of equations in N dimensions is *much* more difficult than finding roots in the one-dimensional case. The principal difference between one and many dimensions is that, in one dimension, it is possible to bracket or "trap" a root between bracketing values, and then hunt it down like a rabbit. In multidimensions, you can never be sure that the root is there at all until you have found it.

Except in linear problems, root finding invariably proceeds by iteration, and this is equally true in one or in many dimensions. Starting from some approximate trial solution, a useful algorithm will improve the solution until some predetermined convergence criterion is satisfied. For smoothly varying functions, good algorithms

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 27 of 181

will always converge, *provided* that the initial guess is good enough. Indeed one can even determine in advance the rate of convergence of most algorithms.

It cannot be overemphasized, however, how crucially success depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numerics. Carefully crafted initial estimates reward you not only with reduced computational effort, but also with understanding and increased self-esteem. Hamming's motto, "the purpose of computing is insight, not numbers," is particularly apt in the area of finding roots. You should repeat this motto aloud whenever your program converges, with ten-digit accuracy, to the wrong root of a problem, or whenever it fails to converge because there is actually *no* root, or because there is a root but your initial estimate was not sufficiently close to it.

"This talk of insight is all very well, but what do I actually do?" For one-dimensional root finding, it is possible to give some straightforward answers: You should try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like. Next, you should always bracket a root, that is, know that the function changes sign in an identified interval, before trying to converge to the root's value.

Finally (this is advice with which some daring souls might disagree, but we give it nonetheless) never let your iteration method get outside of the best bracketing bounds obtained at any stage. We will see below that some pedagogically important algorithms, such as *secant method* or *Newton-Raphson*, can violate this last constraint, and are thus not recommended unless certain fixups are implemented.

Multiple roots, or very close roots, are a real problem, especially if the multiplicity is an even number. In that case, there may be no readily apparent sign change in the function, so the notion of bracketing a root — and maintaining the bracket — becomes difficult. We are hard-liners: we nevertheless insist on bracketing a root, even if it takes the minimum-searching techniques of Chapter 10 to determine whether a tantalizing dip in the function really does cross zero or not. (You can easily modify the simple golden section routine of §10.1 to return early if it detects a sign change in the function. And, if the minimum of the function is exactly zero, then you have found a *double* root.)

As usual, we want to discourage you from using routines as black boxes without understanding them. However, as a guide to beginners, here are some reasonable starting points:

- Brent's algorithm in §9.3 is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function's derivative. Ridders' method (§9.2) is concise, and a close competitor.
- When you can compute the function's derivative, the routine `rtsafe` in §9.4, which combines the Newton-Raphson method with some bookkeeping on bounds, is recommended. Again, you must first bracket your root.
- Roots of polynomials are a special case. Laguerre's method, in §9.5, is recommended as a starting point. Beware: Some polynomials are ill-conditioned!
- Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), which works *very* well if you can supply a

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 28 of 181

good first guess of the solution. Try it. Then read the more advanced material in §9.7 for some more complicated, but globally more convergent, alternatives.

Avoiding implementations for specific computers, this book must generally steer clear of interactive or graphics-related routines. We make an exception right now. The following routine, which produces a crude function plot with interactively scaled axes, can save you a lot of grief as you enter the world of root finding.

```
#include <stdio.h>
#define ISCR 60                 Number of horizontal and vertical positions in display.
#define JSCR 21
#define BLANK ' '
#define ZERO '-'
#define YY 'l'
#define XX '-'
#define FF 'x'

void scrsho(float (*fx)(float))
For interactive CRT terminal use. Produce a crude graph of the function fx over the prompted-
for interval x1,x2. Query for another plot until the user signals satisfaction.
{
    int jz,j,i;
    float ysml,ybig,x2,x1,x,dyj,dx,y[ISCR+1];
    char scr[ISCR+1][JSCR+1];

    for (;;) {
        printf("\nEnter x1 x2 (x1=x2 to stop):\n");      Query for another plot, quit
        scanf("%f %f",&x1,&x2);                              if x1=x2.
        if (x1 == x2) break;
        for (j=1;j<=JSCR;j++)                   Fill vertical sides with character 'l'.
            scr[1][j]=scr[ISCR][j]=YY;
        for (i=2;i<=(ISCR-1);i++) {
            scr[i][1]=scr[i][JSCR]=XX;          Fill top, bottom with character '-'.
            for (j=2;j<=(JSCR-1);j++)           Fill interior with blanks.
                scr[i][j]=BLANK;
        }
        dx=(x2-x1)/(ISCR-1);
        x=x1;
        ysml=ybig=0.0;                          Limits will include 0.
        for (i=1;i<=ISCR;i++) {                 Evaluate the function at equal intervals.
            y[i]=(*fx)(x);                          Find the largest and smallest val-
            if (y[i] < ysml) ysml=y[i];             ues.
            if (y[i] > ybig) ybig=y[i];
            x += dx;
        }
        if (ybig == ysml) ybig=ysml+1.0;        Be sure to separate top and bottom.
        dyj=(JSCR-1)/(ybig-ysml);
        jz=1-(int) (ysml*dyj);                  Note which row corresponds to 0.
        for (i=1;i<=ISCR;i++) {                 Place an indicator at function height and
            scr[i][jz]=ZERO;                        0.
            j=1+(int) ((y[i]-ysml)*dyj);
            scr[i][j]=FF;
        }
        printf(" %10.3f ",ybig);
        for (i=1;i<=ISCR;i++) printf("%c",scr[i][JSCR]);
        printf("\n");
        for (j=(JSCR-1);j>=2;j--) {             Display.
            printf("%12s"," ");
            for (i=1;i<=ISCR;i++) printf("%c",scr[i][j]);
            printf("\n");
        }
        printf(" %10.3f ",ysml);
```

```
        for (i=1;i<=ISCR;i++) printf("%c",scr[i][1]);
        printf("\n");
        printf("%8s %10.3f %44s %10.3f\n"," ",x1," ",x2);
    }
}
```

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 5.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapters 2, 7, and 14.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 8.

Householder, A.S. 1970, *The Numerical Treatment of a Single Nonlinear Equation* (New York: McGraw-Hill).

9.1 Bracketing and Bisection

We will say that a root is *bracketed* in the interval (a, b) if $f(a)$ and $f(b)$ have opposite signs. If the function is continuous, then at least one root must lie in that interval (the *intermediate value theorem*). If the function is discontinuous, but bounded, then instead of a root there might be a step discontinuity which crosses zero (see Figure 9.1.1). For numerical purposes, that might as well be a root, since the behavior is indistinguishable from the case of a continuous function whose zero crossing occurs in between two "adjacent" floating-point numbers in a machine's finite-precision representation. Only for functions with singularities is there the possibility that a bracketed root is not really there, as for example

$$f(x) = \frac{1}{x - c} \tag{9.1.1}$$

Some root-finding algorithms (e.g., bisection in this section) will readily converge to c in (9.1.1). Luckily there is not much possibility of your mistaking c, or any number x close to it, for a root, since mere evaluation of $|f(x)|$ will give a very large, rather than a very small, result.

If you are given a function in a black box, there is no sure way of bracketing its roots, or of even determining that it has roots. If you like pathological examples, think about the problem of locating the two real roots of equation (3.0.1), which dips below zero only in the ridiculously small interval of about $x = \pi \pm 10^{-667}$.

In the next chapter we will deal with the related problem of bracketing a function's minimum. There it is possible to give a procedure that always succeeds; in essence, "Go downhill, taking steps of increasing size, until your function starts back uphill." There is no analogous procedure for roots. The procedure "go downhill until your function changes sign," can be foiled by a function that has a simple extremum. Nevertheless, if you are prepared to deal with a "failure" outcome, this procedure is often a good first start; success is usual if your function has opposite signs in the limit $x \to \pm\infty$.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 30 of 181

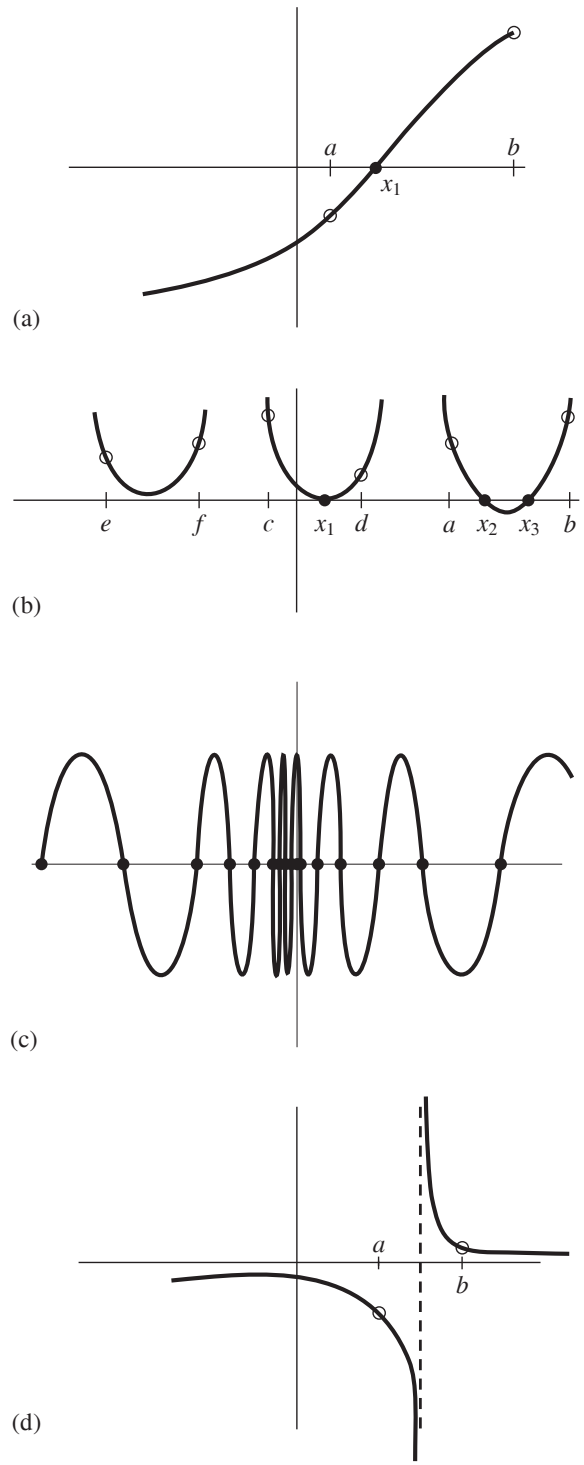

Figure 9.1.1. Some situations encountered while root finding: (a) shows an isolated root x_1 bracketed by two points a and b at which the function has opposite signs; (b) illustrates that there is not necessarily a sign change in the function near a double root (in fact, there is not necessarily a root!); (c) is a pathological function with many roots; in (d) the function has opposite signs at points a and b, but the points bracket a singularity, not a root.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 31 of 181

```
#include <math.h>
#define FACTOR 1.6
#define NTRY 50

int zbrac(float (*func)(float), float *x1, float *x2)
```

Given a function func and an initial guessed range x1 to x2, the routine expands the range geometrically until a root is bracketed by the returned values x1 and x2 (in which case zbrac returns 1) or until the range becomes unacceptably large (in which case zbrac returns 0).

```
{
    void nrerror(char error_text[]);
    int j;
    float f1,f2;

    if (*x1 == *x2) nrerror("Bad initial range in zbrac");
    f1=(*func)(*x1);
    f2=(*func)(*x2);
    for (j=1;j<=NTRY;j++) {
        if (f1*f2 < 0.0) return 1;
        if (fabs(f1) < fabs(f2))
            f1=(*func)(*x1 += FACTOR*(*x1-*x2));
        else
            f2=(*func)(*x2 += FACTOR*(*x2-*x1));
    }
    return 0;
}
```

Alternatively, you might want to "look inward" on an initial interval, rather than "look outward" from it, asking if there are any roots of the function $f(x)$ in the interval from x_1 to x_2 when a search is carried out by subdivision into n equal intervals. The following function calculates brackets for up to nb distinct intervals which each contain one or more roots.

```
void zbrak(float (*fx)(float), float x1, float x2, int n, float xb1[],
    float xb2[], int *nb)
```

Given a function fx defined on the interval from x1−x2 subdivide the interval into n equally spaced segments, and search for zero crossings of the function. nb is input as the maximum number of roots sought, and is reset to the number of bracketing pairs xb1[1..nb], xb2[1..nb] that are found.

```
{
    int nbb,i;
    float x,fp,fc,dx;

    nbb=0;
    dx=(x2-x1)/n;                    Determine the spacing appropriate to the mesh.
    fp=(*fx)(x=x1);
    for (i=1;i<=n;i++) {             Loop over all intervals
        fc=(*fx)(x += dx);
        if (fc*fp <= 0.0) {          If a sign change occurs then record values for the
            xb1[++nbb]=x-dx;             bounds.
            xb2[nbb]=x;
            if(*nb == nbb) return;

        }
        fp=fc;
    }
    *nb = nbb;
}
```

Bisection Method

Once we know that an interval contains a root, several classical procedures are available to refine it. These proceed with varying degrees of speed and sureness towards the answer. Unfortunately, the methods that are guaranteed to converge plod along most slowly, while those that rush to the solution in the best cases can also dash rapidly to infinity without warning if measures are not taken to avoid such behavior.

The *bisection method* is one that cannot fail. It is thus not to be sneered at as a method for otherwise badly behaved problems. The idea is simple. Over some interval the function is known to pass through zero because it changes sign. Evaluate the function at the interval's midpoint and examine its sign. Use the midpoint to replace whichever limit has the same sign. After each iteration the bounds containing the root decrease by a factor of two. If after n iterations the root is known to be within an interval of size ϵ_n, then after the next iteration it will be bracketed within an interval of size

$$\epsilon_{n+1} = \epsilon_n/2 \tag{9.1.2}$$

neither more nor less. Thus, we know in advance the number of iterations required to achieve a given tolerance in the solution,

$$n = \log_2 \frac{\epsilon_0}{\epsilon} \tag{9.1.3}$$

where ϵ_0 is the size of the initially bracketing interval, ϵ is the desired ending tolerance.

Bisection *must* succeed. If the interval happens to contain two or more roots, bisection will find one of them. If the interval contains no roots and merely straddles a singularity, it will converge on the singularity.

When a method converges as a factor (less than 1) times the previous uncertainty to the first power (as is the case for bisection), it is said to converge *linearly*. Methods that converge as a higher power,

$$\epsilon_{n+1} = \text{constant} \times (\epsilon_n)^m \qquad m > 1 \tag{9.1.4}$$

are said to converge superlinearly. In other contexts "linear" convergence would be termed "exponential," or "geometrical." That is not too bad at all: Linear convergence means that successive significant figures are won linearly with computational effort.

It remains to discuss practical criteria for convergence. It is crucial to keep in mind that computers use a fixed number of binary digits to represent floating-point numbers. While your function might analytically pass through zero, it is possible that its computed value is never zero, for any floating-point argument. One must decide what accuracy on the root is attainable: Convergence to within 10^{-6} in absolute value is reasonable when the root lies near 1, but certainly unachievable if the root lies near 10^{26}. One might thus think to specify convergence by a relative (fractional) criterion, but this becomes unworkable for roots near zero. To be most general, the routines below will require you to specify an absolute tolerance, such that iterations continue until the interval becomes smaller than this tolerance in absolute units. Usually you may wish to take the tolerance to be $\epsilon(|x_1| + |x_2|)/2$ where ϵ is the machine precision and x_1 and x_2 are the initial brackets. When the root lies near zero you ought to consider carefully what reasonable tolerance means for your function. The following routine quits after 40 bisections in any event, with $2^{-40} \approx 10^{-12}$.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 33 of 181

```
#include <math.h>
#define JMAX 40                    Maximum allowed number of bisections.

float rtbis(float (*func)(float), float x1, float x2, float xacc)
Using bisection, find the root of a function func known to lie between x1 and x2. The root,
returned as rtbis, will be refined until its accuracy is ±xacc.
{
    void nrerror(char error_text[]);
    int j;
    float dx,f,fmid,xmid,rtb;

    f=(*func)(x1);
    fmid=(*func)(x2);
    if (f*fmid >= 0.0) nrerror("Root must be bracketed for bisection in rtbis");
    rtb = f < 0.0 ? (dx=x2-x1,x1) : (dx=x1-x2,x2);      Orient the search so that f>0
    for (j=1;j<=JMAX;j++) {                                 lies at x+dx.
        fmid=(*func)(xmid=rtb+(dx *= 0.5));             Bisection loop.
        if (fmid <= 0.0) rtb=xmid;
        if (fabs(dx) < xacc || fmid == 0.0) return rtb;
    }
    nrerror("Too many bisections in rtbis");
    return 0.0;                                         Never get here.
}
```

9.2 Secant Method, False Position Method, and Ridders' Method

For functions that are smooth near a root, the methods known respectively as *false position* (or *regula falsi*) and *secant method* generally converge faster than bisection. In both of these methods the function is assumed to be approximately linear in the local region of interest, and the next improvement in the root is taken as the point where the approximating line crosses the axis. After each iteration one of the previous boundary points is discarded in favor of the latest estimate of the root.

The *only* difference between the methods is that secant retains the most recent of the prior estimates (Figure 9.2.1; this requires an arbitrary choice on the first iteration), while false position retains that prior estimate for which the function value has opposite sign from the function value at the current best estimate of the root, so that the two points continue to bracket the root (Figure 9.2.2). Mathematically, the secant method converges more rapidly near a root of a sufficiently continuous function. Its order of convergence can be shown to be the "golden ratio" $1.618\ldots$, so that

$$\lim_{k\to\infty} |\epsilon_{k+1}| \approx \text{const} \times |\epsilon_k|^{1.618} \tag{9.2.1}$$

The secant method has, however, the disadvantage that the root does not necessarily remain bracketed. For functions that are *not* sufficiently continuous, the algorithm can therefore not be guaranteed to converge: Local behavior might send it off towards infinity.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 34 of 181

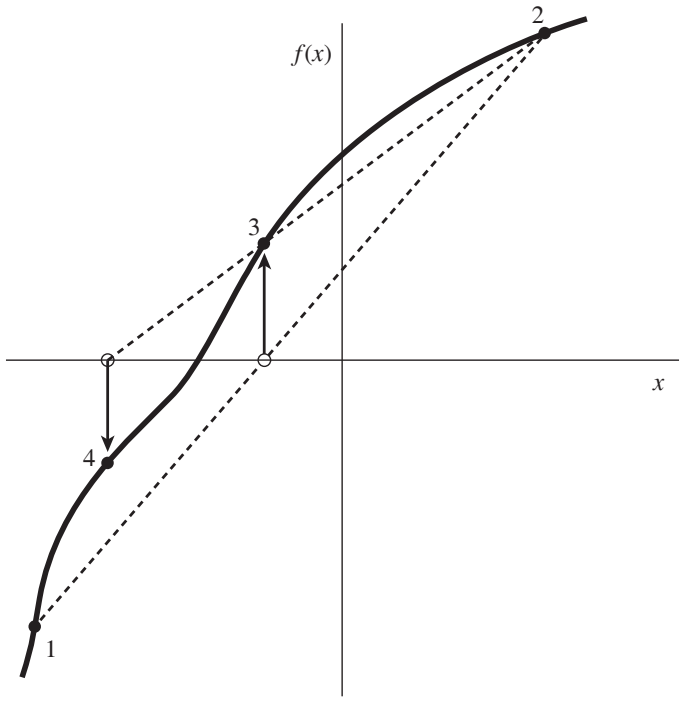

Figure 9.2.1. Secant method. Extrapolation or interpolation lines (dashed) are drawn through the two most recently evaluated points, whether or not they bracket the function. The points are numbered in the order that they are used.

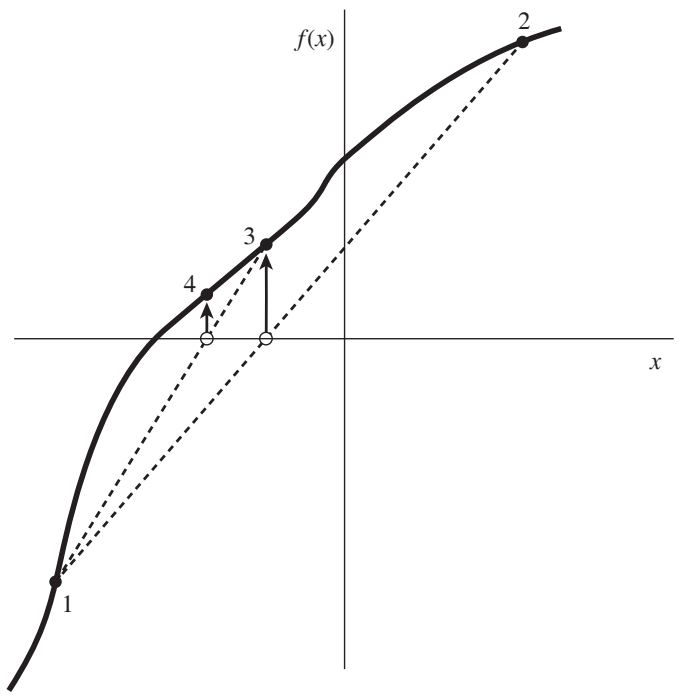

Figure 9.2.2. False position method. Interpolation lines (dashed) are drawn through the most recent points *that bracket the root*. In this example, point 1 thus remains "active" for many steps. False position converges less rapidly than the secant method, but it is more certain.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 35 of 181

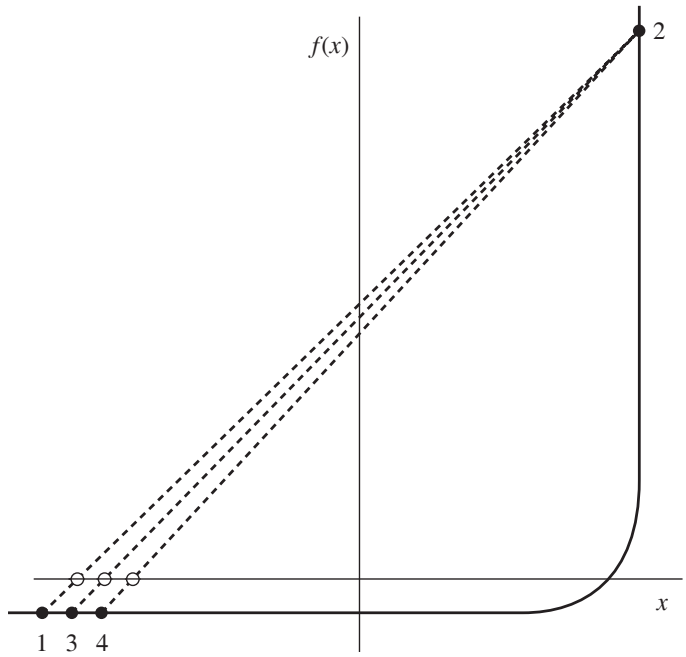

Figure 9.2.3. Example where both the secant and false position methods will take many iterations to arrive at the true root. This function would be difficult for many other root-finding methods.

False position, since it sometimes keeps an older rather than newer function evaluation, has a lower order of convergence. Since the newer function value will *sometimes* be kept, the method is often superlinear, but estimation of its exact order is not so easy.

Here are sample implementations of these two related methods. While these methods are standard textbook fare, *Ridders' method*, described below, or *Brent's method*, in the next section, are almost always better choices. Figure 9.2.3 shows the behavior of secant and false-position methods in a difficult situation.

```
#include <math.h>
#define MAXIT 30                    Set to the maximum allowed number of iterations.

float rtflsp(float (*func)(float), float x1, float x2, float xacc)
Using the false position method, find the root of a function func known to lie between x1 and
x2. The root, returned as rtflsp, is refined until its accuracy is ±xacc.
{
    void nrerror(char error_text[]);
    int j;
    float fl,fh,xl,xh,swap,dx,del,f,rtf;

    fl=(*func)(x1);
    fh=(*func)(x2);                 Be sure the interval brackets a root.
    if (fl*fh > 0.0) nrerror("Root must be bracketed in rtflsp");
    if (fl < 0.0) {                 Identify the limits so that xl corresponds to the low
        xl=x1;                          side.
        xh=x2;
    } else {
        xl=x2;
        xh=x1;
        swap=fl;
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 36 of 181

```
        fl=fh;
        fh=swap;
    }
    dx=xh-xl;
    for (j=1;j<=MAXIT;j++) {        False position loop.
        rtf=xl+dx*fl/(fl-fh);       Increment with respect to latest value.
        f=(*func)(rtf);
        if (f < 0.0) {              Replace appropriate limit.
            del=xl-rtf;
            xl=rtf;
            fl=f;
        } else {
            del=xh-rtf;
            xh=rtf;
            fh=f;
        }
        dx=xh-xl;
        if (fabs(del) < xacc || f == 0.0) return rtf;       Convergence.
    }
    nrerror("Maximum number of iterations exceeded in rtflsp");
    return 0.0;                     Never get here.
}
```

```
#include <math.h>
#define MAXIT 30                    Maximum allowed number of iterations.

float rtsec(float (*func)(float), float x1, float x2, float xacc)
```

Using the secant method, find the root of a function `func` thought to lie between `x1` and `x2`. The root, returned as `rtsec`, is refined until its accuracy is ±`xacc`.

```
{
    void nrerror(char error_text[]);
    int j;
    float fl,f,dx,swap,xl,rts;

    fl=(*func)(x1);
    f=(*func)(x2);
    if (fabs(fl) < fabs(f)) {       Pick the bound with the smaller function value as
        rts=x1;                         the most recent guess.
        xl=x2;
        swap=fl;
        fl=f;
        f=swap;
    } else {
        xl=x1;
        rts=x2;
    }
    for (j=1;j<=MAXIT;j++) {        Secant loop.
        dx=(xl-rts)*f/(f-fl);       Increment with respect to latest value.
        xl=rts;
        fl=f;
        rts += dx;
        f=(*func)(rts);
        if (fabs(dx) < xacc || f == 0.0) return rts;     Convergence.
    }
    nrerror("Maximum number of iterations exceeded in rtsec");
    return 0.0;                     Never get here.
}
```

Ridders' Method

A powerful variant on false position is due to Ridders [1]. When a root is bracketed between x_1 and x_2, Ridders' method first evaluates the function at the midpoint $x_3 = (x_1 + x_2)/2$. It then factors out that unique exponential function which turns the residual function into a straight line. Specifically, it solves for a factor e^Q that gives

$$f(x_1) - 2f(x_3)e^Q + f(x_2)e^{2Q} = 0 \tag{9.2.2}$$

This is a quadratic equation in e^Q, which can be solved to give

$$e^Q = \frac{f(x_3) + \text{sign}[f(x_2)]\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}{f(x_2)} \tag{9.2.3}$$

Now the false position method is applied, not to the values $f(x_1), f(x_3), f(x_2)$, but to the values $f(x_1), f(x_3)e^Q, f(x_2)e^{2Q}$, yielding a new guess for the root, x_4. The overall updating formula (incorporating the solution 9.2.3) is

$$x_4 = x_3 + (x_3 - x_1)\frac{\text{sign}[f(x_1) - f(x_2)]f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}} \tag{9.2.4}$$

Equation (9.2.4) has some very nice properties. First, x_4 is guaranteed to lie in the interval (x_1, x_2), so the method never jumps out of its brackets. Second, the convergence of successive applications of equation (9.2.4) is *quadratic*, that is, $m = 2$ in equation (9.1.4). Since each application of (9.2.4) requires two function evaluations, the actual order of the method is $\sqrt{2}$, not 2; but this is still quite respectably superlinear: the number of significant digits in the answer approximately *doubles* with each two function evaluations. Third, taking out the function's "bend" via exponential (that is, ratio) factors, rather than via a polynomial technique (e.g., fitting a parabola), turns out to give an extraordinarily robust algorithm. In both reliability and speed, Ridders' method is generally competitive with the more highly developed and better established (but more complicated) method of Van Wijngaarden, Dekker, and Brent, which we next discuss.

```
#include <math.h>
#include "nrutil.h"
#define MAXIT 60
#define UNUSED (-1.11e30)

float zriddr(float (*func)(float), float x1, float x2, float xacc)
```

Using Ridders' method, return the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `zriddr`, will be refined to an approximate accuracy `xacc`.

```
{
    int j;
    float ans,fh,fl,fm,fnew,s,xh,xl,xm,xnew;

    fl=(*func)(x1);
    fh=(*func)(x2);
    if ((fl > 0.0 && fh < 0.0) || (fl < 0.0 && fh > 0.0)) {
        xl=x1;
        xh=x2;
        ans=UNUSED;                    Any highly unlikely value, to simplify logic
                                           below.
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 38 of 181

```
        for (j=1;j<=MAXIT;j++) {
            xm=0.5*(xl+xh);
            fm=(*func)(xm);                    First of two function evaluations per it-
            s=sqrt(fm*fm-fl*fh);                   eration.
            if (s == 0.0) return ans;
            xnew=xm+(xm-xl)*((fl >= fh ? 1.0 : -1.0)*fm/s);     Updating formula.
            if (fabs(xnew-ans) <= xacc) return ans;
            ans=xnew;
            fnew=(*func)(ans);                 Second of two function evaluations per
            if (fnew == 0.0) return ans;           iteration.
            if (SIGN(fm,fnew) != fm) {         Bookkeeping to keep the root bracketed
                xl=xm;                             on next iteration.
                fl=fm;
                xh=ans;
                fh=fnew;
            } else if (SIGN(fl,fnew) != fl) {
                xh=ans;
                fh=fnew;
            } else if (SIGN(fh,fnew) != fh) {
                xl=ans;
                fl=fnew;
            } else nrerror("never get here.");
            if (fabs(xh-xl) <= xacc) return ans;
        }
        nrerror("zriddr exceed maximum iterations");
    }
    else {
        if (fl == 0.0) return x1;
        if (fh == 0.0) return x2;
        nrerror("root must be bracketed in zriddr.");
    }
    return 0.0;                                Never get here.
}
```

CITED REFERENCES AND FURTHER READING:

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.3.

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press), Chapter 12.

Ridders, C.J.F. 1979, *IEEE Transactions on Circuits and Systems*, vol. CAS-26, pp. 979–980. [1]

9.3 Van Wijngaarden–Dekker–Brent Method

While secant and false position formally converge faster than bisection, one finds in practice pathological functions for which bisection converges more rapidly. These can be choppy, discontinuous functions, or even smooth functions if the second derivative changes sharply near the root. Bisection always halves the interval, while secant and false position can sometimes spend many cycles slowly pulling distant bounds closer to a root. Ridders' method does a much better job, but it too can sometimes be fooled. Is there a way to combine superlinear convergence with the sureness of bisection?

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 39 of 181

Yes. We can keep track of whether a supposedly superlinear method is actually converging the way it is supposed to, and, if it is not, we can intersperse bisection steps so as to guarantee *at least* linear convergence. This kind of super-strategy requires attention to bookkeeping detail, and also careful consideration of how roundoff errors can affect the guiding strategy. Also, we must be able to determine reliably when convergence has been achieved.

An excellent algorithm that pays close attention to these matters was developed in the 1960s by van Wijngaarden, Dekker, and others at the Mathematical Center in Amsterdam, and later improved by Brent [1]. For brevity, we refer to the final form of the algorithm as *Brent's method*. The method is *guaranteed* (by Brent) to converge, so long as the function can be evaluated within the initial interval known to contain a root.

Brent's method combines root bracketing, bisection, and *inverse quadratic interpolation* to converge from the neighborhood of a zero crossing. While the false position and secant methods assume approximately linear behavior between two prior root estimates, inverse quadratic interpolation uses three prior points to fit an inverse quadratic function (x as a quadratic function of y) whose value at $y = 0$ is taken as the next estimate of the root x. Of course one must have contingency plans for what to do if the root falls outside of the brackets. Brent's method takes care of all that. If the three point pairs are $[a, f(a)], [b, f(b)], [c, f(c)]$ then the interpolation formula (cf. equation 3.1.1) is

$$x = \frac{[y - f(a)][y - f(b)]c}{[f(c) - f(a)][f(c) - f(b)]} + \frac{[y - f(b)][y - f(c)]a}{[f(a) - f(b)][f(a) - f(c)]} + \frac{[y - f(c)][y - f(a)]b}{[f(b) - f(c)][f(b) - f(a)]} \tag{9.3.1}$$

Setting y to zero gives a result for the next root estimate, which can be written as

$$x = b + P/Q \tag{9.3.2}$$

where, in terms of

$$R \equiv f(b)/f(c), \qquad S \equiv f(b)/f(a), \qquad T \equiv f(a)/f(c) \tag{9.3.3}$$

we have

$$P = S\,[T(R - T)(c - b) - (1 - R)(b - a)] \tag{9.3.4}$$

$$Q = (T - 1)(R - 1)(S - 1) \tag{9.3.5}$$

In practice b is the current best estimate of the root and P/Q ought to be a "small" correction. Quadratic methods work well only when the function behaves smoothly; they run the serious risk of giving very bad estimates of the next root or causing machine failure by an inappropriate division by a very small number ($Q \approx 0$). Brent's method guards against this problem by maintaining brackets on the root and checking where the interpolation would land before carrying out the division. When the correction P/Q would not land within the bounds, or when the bounds are not collapsing rapidly enough, the algorithm takes a bisection step. Thus,

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 40 of 181

Brent's method combines the sureness of bisection with the speed of a higher-order method when appropriate. We recommend it as the method of choice for general one-dimensional root finding where a function's values only (and not its derivative or functional form) are available.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100                          Maximum allowed number of iterations.
#define EPS 3.0e-8                         Machine floating-point precision.

float zbrent(float (*func)(float), float x1, float x2, float tol)
Using Brent's method, find the root of a function func known to lie between x1 and x2. The
root, returned as zbrent, will be refined until its accuracy is tol.
{
    int iter;
    float a=x1,b=x2,c=x2,d,e,min1,min2;
    float fa=(*func)(a),fb=(*func)(b),fc,p,q,r,s,tol1,xm;

    if ((fa > 0.0 && fb > 0.0) || (fa < 0.0 && fb < 0.0))
        nrerror("Root must be bracketed in zbrent");
    fc=fb;
    for (iter=1;iter<=ITMAX;iter++) {
        if ((fb > 0.0 && fc > 0.0) || (fb < 0.0 && fc < 0.0)) {
            c=a;                           Rename a, b, c and adjust bounding interval
            fc=fa;                             d.
            e=d=b-a;
        }
        if (fabs(fc) < fabs(fb)) {
            a=b;
            b=c;
            c=a;
            fa=fb;
            fb=fc;
            fc=fa;
        }
        tol1=2.0*EPS*fabs(b)+0.5*tol;      Convergence check.
        xm=0.5*(c-b);
        if (fabs(xm) <= tol1 || fb == 0.0) return b;
        if (fabs(e) >= tol1 && fabs(fa) > fabs(fb)) {
            s=fb/fa;                       Attempt inverse quadratic interpolation.
            if (a == c) {
                p=2.0*xm*s;
                q=1.0-s;
            } else {
                q=fa/fc;
                r=fb/fc;
                p=s*(2.0*xm*q*(q-r)-(b-a)*(r-1.0));
                q=(q-1.0)*(r-1.0)*(s-1.0);
            }
            if (p > 0.0) q = -q;           Check whether in bounds.
            p=fabs(p);
            min1=3.0*xm*q-fabs(tol1*q);
            min2=fabs(e*q);
            if (2.0*p < (min1 < min2 ? min1 : min2)) {
                e=d;                       Accept interpolation.
                d=p/q;
            } else {
                d=xm;                      Interpolation failed, use bisection.
                e=d;
            }
        } else {                           Bounds decreasing too slowly, use bisection.
            d=xm;
            e=d;
```

```
        }
        a=b;                                Move last best guess to a.
        fa=fb;
        if (fabs(d) > tol1)                 Evaluate new trial root.
            b += d;
        else
            b += SIGN(tol1,xm);
        fb=(*func)(b);
    }
    nrerror("Maximum number of iterations exceeded in zbrent");
    return 0.0;                             Never get here.
}
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 3, 4. [1]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §7.2.

9.4 Newton-Raphson Method Using Derivative

Perhaps the most celebrated of all one-dimensional root-finding routines is *Newton's method*, also called the *Newton-Raphson method*. This method is distinguished from the methods of previous sections by the fact that it requires the evaluation of both the function $f(x)$, *and* the derivative $f'(x)$, at arbitrary points x. The Newton-Raphson formula consists geometrically of extending the tangent line at a current point x_i until it crosses zero, then setting the next guess x_{i+1} to the abscissa of that zero-crossing (see Figure 9.4.1). Algebraically, the method derives from the familiar Taylor series expansion of a function in the neighborhood of a point,

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \dots . \tag{9.4.1}$$

For small enough values of δ, and for well-behaved functions, the terms beyond linear are unimportant, hence $f(x+\delta) = 0$ implies

$$\delta = -\frac{f(x)}{f'(x)}. \tag{9.4.2}$$

Newton-Raphson is not restricted to one dimension. The method readily generalizes to multiple dimensions, as we shall see in §9.6 and §9.7, below.

Far from a root, where the higher-order terms in the series *are* important, the Newton-Raphson formula can give grossly inaccurate, meaningless corrections. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. This can be death to the method (see Figure 9.4.2). If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson sends its solution off to limbo, with vanishingly small hope of recovery.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 42 of 181

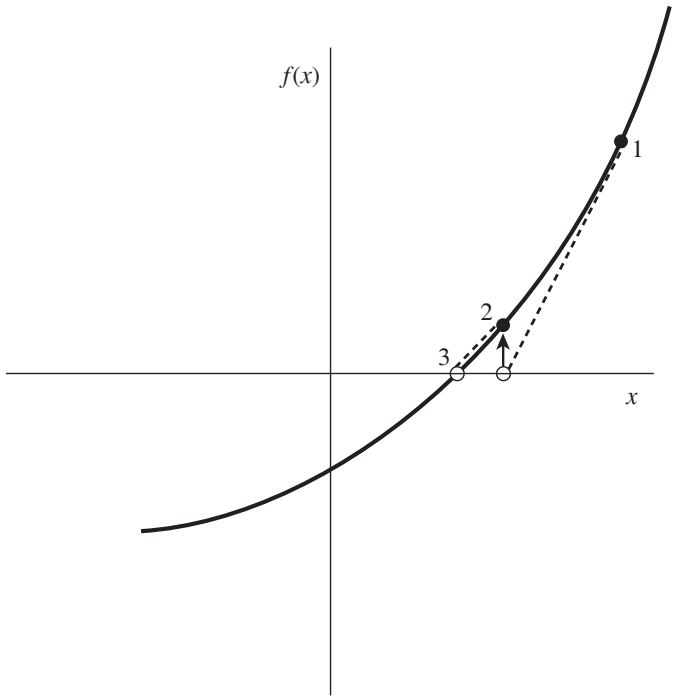

Figure 9.4.1. Newton's method extrapolates the local derivative to find the next estimate of the root. In this example it works well and converges quadratically.

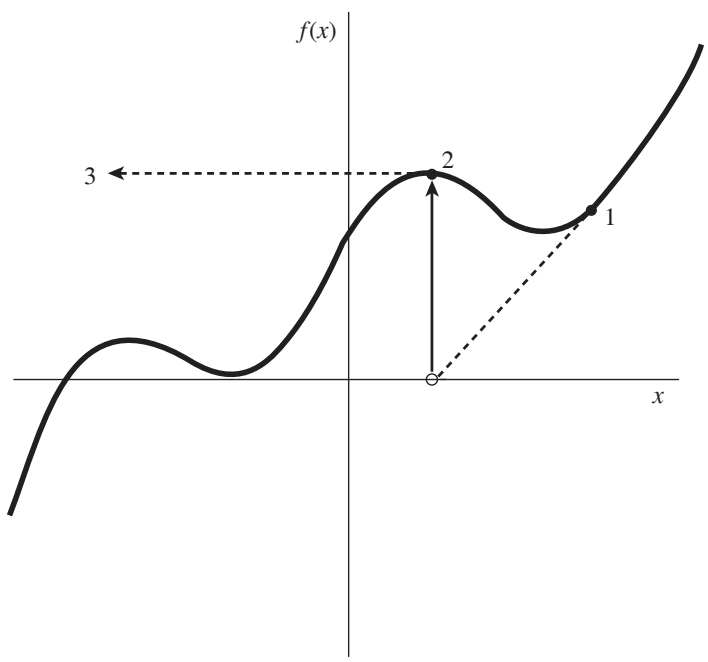

Figure 9.4.2. Unfortunate case where Newton's method encounters a local extremum and shoots off to outer space. Here bracketing bounds, as in `rtsafe`, would save the day.

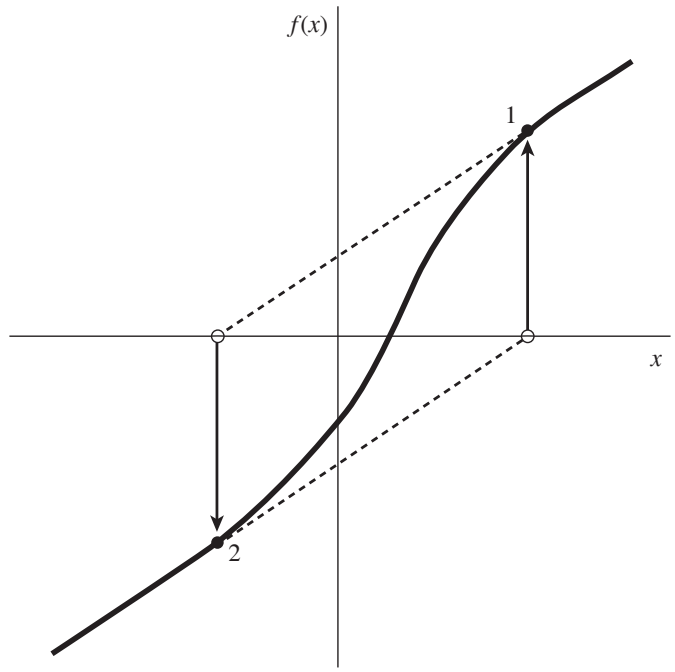

Figure 9.4.3. Unfortunate case where Newton's method enters a nonconvergent cycle. This behavior is often encountered when the function f is obtained, in whole or in part, by table interpolation. With a better initial guess, the method would have succeeded.

Like most powerful tools, Newton-Raphson can be destructive used in inappropriate circumstances. Figure 9.4.3 demonstrates another possible pathology.

Why do we call Newton-Raphson powerful? The answer lies in its rate of convergence: Within a small distance ϵ of x the function and its derivative are approximately:

$$\begin{aligned} f(x+\epsilon) &= f(x) + \epsilon f'(x) + \epsilon^2 \frac{f''(x)}{2} + \cdots, \\ f'(x+\epsilon) &= f'(x) + \epsilon f''(x) + \cdots \end{aligned} \tag{9.4.3}$$

By the Newton-Raphson formula,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{9.4.4}$$

so that

$$\epsilon_{i+1} = \epsilon_i - \frac{f(x_i)}{f'(x_i)}. \tag{9.4.5}$$

When a trial solution x_i differs from the true root by ϵ_i, we can use (9.4.3) to express $f(x_i), f'(x_i)$ in (9.4.4) in terms of ϵ_i and derivatives at the root itself. The result is a recurrence relation for the deviations of the trial solutions

$$\epsilon_{i+1} = -\epsilon_i^2 \frac{f''(x)}{2f'(x)}. \tag{9.4.6}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 44 of 181

Equation (9.4.6) says that Newton-Raphson converges *quadratically* (cf. equation 9.2.3). Near a root, the number of significant digits approximately *doubles* with each step. This very strong convergence property makes Newton-Raphson the method of choice for any function whose derivative can be evaluated efficiently, and whose derivative is continuous and nonzero in the neighborhood of a root.

Even where Newton-Raphson is rejected for the early stages of convergence (because of its poor global convergence properties), it is very common to "polish up" a root with one or two steps of Newton-Raphson, which can multiply by two or four its number of significant figures!

For an efficient realization of Newton-Raphson the user provides a routine that evaluates both $f(x)$ and its first derivative $f'(x)$ at the point x. The Newton-Raphson formula can also be applied using a numerical difference to approximate the true local derivative,

$$f'(x) \approx \frac{f(x+dx) - f(x)}{dx}. \tag{9.4.7}$$

This is not, however, a recommended procedure for the following reasons: (i) You are doing two function evaluations per step, so *at best* the superlinear order of convergence will be only $\sqrt{2}$. (ii) If you take dx too small you will be wiped out by roundoff, while if you take it too large your order of convergence will be only linear, no better than using the *initial* evaluation $f'(x_0)$ for all subsequent steps. Therefore, Newton-Raphson with numerical derivatives is (in one dimension) always dominated by the secant method of §9.2. (In multidimensions, where there is a paucity of available methods, Newton-Raphson with numerical derivatives must be taken more seriously. See §§9.6–9.7.)

The following function calls a user supplied function `funcd(x,fn,df)` which supplies the function value as `fn` and the derivative as `df`. We have included input bounds on the root simply to be consistent with previous root-finding routines: Newton does not adjust bounds, and works only on local information at the point `x`. The bounds are used only to pick the midpoint as the first guess, and to reject the solution if it wanders outside of the bounds.

```
#include <math.h>
#define JMAX 20                                  Set to maximum number of iterations.

float rtnewt(void (*funcd)(float, float *, float *), float x1, float x2,
    float xacc)
Using the Newton-Raphson method, find the root of a function known to lie in the interval
[x1, x2]. The root rtnewt will be refined until its accuracy is known within ±xacc. funcd
is a user-supplied routine that returns both the function value and the first derivative of the
function at the point x.
{
    void nrerror(char error_text[]);
    int j;
    float df,dx,f,rtn;

    rtn=0.5*(x1+x2);                             Initial guess.
    for (j=1;j<=JMAX;j++) {
        (*funcd)(rtn,&f,&df);
        dx=f/df;
        rtn -= dx;
        if ((x1-rtn)*(rtn-x2) < 0.0)
            nrerror("Jumped out of brackets in rtnewt");
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 45 of 181

```
        if (fabs(dx) < xacc) return rtn;      Convergence.
    }
    nrerror("Maximum number of iterations exceeded in rtnewt");
    return 0.0;                               Never get here.
}
```

While Newton-Raphson's global convergence properties are poor, it is fairly easy to design a fail-safe routine that utilizes a combination of bisection and Newton-Raphson. The hybrid algorithm takes a bisection step whenever Newton-Raphson would take the solution out of bounds, or whenever Newton-Raphson is not reducing the size of the brackets rapidly enough.

```
#include <math.h>
#define MAXIT 100                             Maximum allowed number of iterations.

float rtsafe(void (*funcd)(float, float *, float *), float x1, float x2,
    float xacc)
Using a combination of Newton-Raphson and bisection, find the root of a function bracketed
between x1 and x2. The root, returned as the function value rtsafe, will be refined until
its accuracy is known within ±xacc. funcd is a user-supplied routine that returns both the
function value and the first derivative of the function.
{
    void nrerror(char error_text[]);
    int j;
    float df,dx,dxold,f,fh,fl;
    float temp,xh,xl,rts;

    (*funcd)(x1,&fl,&df);
    (*funcd)(x2,&fh,&df);
    if ((fl > 0.0 && fh > 0.0) || (fl < 0.0 && fh < 0.0))
        nrerror("Root must be bracketed in rtsafe");
    if (fl == 0.0) return x1;
    if (fh == 0.0) return x2;
    if (fl < 0.0) {                           Orient the search so that f(xl) < 0.
        xl=x1;
        xh=x2;
    } else {
        xh=x1;
        xl=x2;
    }
    rts=0.5*(x1+x2);                          Initialize the guess for root,
    dxold=fabs(x2-x1);                        the "stepsize before last,"
    dx=dxold;                                 and the last step.
    (*funcd)(rts,&f,&df);
    for (j=1;j<=MAXIT;j++) {                  Loop over allowed iterations.
        if ((((rts-xh)*df-f)*((rts-xl)*df-f) > 0.0)      Bisect if Newton out of range,
            || (fabs(2.0*f) > fabs(dxold*df))) {         or not decreasing fast enough.
            dxold=dx;
            dx=0.5*(xh-xl);
            rts=xl+dx;
            if (xl == rts) return rts;        Change in root is negligible.
        } else {                              Newton step acceptable. Take it.
            dxold=dx;
            dx=f/df;
            temp=rts;
            rts -= dx;
            if (temp == rts) return rts;
        }
        if (fabs(dx) < xacc) return rts;      Convergence criterion.
        (*funcd)(rts,&f,&df);
        The one new function evaluation per iteration.
```

```
        if (f < 0.0)                          Maintain the bracket on the root.
            xl=rts;
        else
            xh=rts;
    }
    nrerror("Maximum number of iterations exceeded in rtsafe");
    return 0.0;                               Never get here.
}
```

For many functions the derivative $f'(x)$ often converges to machine accuracy before the function $f(x)$ itself does. When that is the case one need not subsequently update $f'(x)$. This shortcut is recommended only when you confidently understand the generic behavior of your function, but it speeds computations when the derivative calculation is laborious. (Formally this makes the convergence only linear, but if the derivative isn't changing anyway, you can do no better.)

Newton-Raphson and Fractals

An interesting sidelight to our repeated warnings about Newton-Raphson's unpredictable global convergence properties — its very rapid local convergence notwithstanding — is to investigate, for some particular equation, the set of starting values from which the method does, or doesn't converge to a root.

Consider the simple equation

$$z^3 - 1 = 0 \tag{9.4.8}$$

whose single real root is $z = 1$, but which also has complex roots at the other two cube roots of unity, $\exp(\pm 2\pi i/3)$. Newton's method gives the iteration

$$z_{j+1} = z_j - \frac{z_j^3 - 1}{3z_j^2} \tag{9.4.9}$$

Up to now, we have applied an iteration like equation (9.4.9) only for real starting values z_0, but in fact all of the equations in this section also apply in the complex plane. We can therefore map out the complex plane into regions from which a starting value z_0, iterated in equation (9.4.9), will, or won't, converge to $z = 1$. Naively, we might expect to find a "basin of convergence" somehow surrounding the root $z = 1$. We surely do not expect the basin of convergence to fill the whole plane, because the plane must also contain regions that converge to each of the two complex roots. In fact, by symmetry, the three regions must have identical shapes. Perhaps they will be three symmetric $120°$ wedges, with one root centered in each?

Now take a look at Figure 9.4.4, which shows the result of a numerical exploration. The basin of convergence does indeed cover $1/3$ the area of the complex plane, but its boundary is highly irregular — in fact, *fractal*. (A fractal, so called, has self-similar structure that repeats on all scales of magnification.) How does this fractal emerge from something as simple as Newton's method, and an equation as simple as (9.4.8)? The answer is already implicit in Figure 9.4.2, which showed how, on the real line, a local extremum causes Newton's method to shoot off to infinity. Suppose one is *slightly* removed from such a point. Then one might be shot off not to infinity, but — by luck — right into the basin of convergence of the desired

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 47 of 181

Figure 9.4.4. The complex z plane with real and imaginary components in the range $(-2, 2)$. The black region is the set of points from which Newton's method converges to the root $z = 1$ of the equation $z^3 - 1 = 0$. Its shape is fractal.

root. But that means that in the neighborhood of an extremum there must be a tiny, perhaps distorted, copy of the basin of convergence — a kind of "one-bounce away" copy. Similar logic shows that there can be "two-bounce" copies, "three-bounce" copies, and so on. A fractal thus emerges.

Notice that, for equation (9.4.8), almost the whole real axis is in the domain of convergence for the root $z = 1$. We say "almost" because of the peculiar discrete points on the negative real axis whose convergence is indeterminate (see figure). What happens if you start Newton's method from one of these points? (Try it.)

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.4.

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press).

Mandelbrot, B.B. 1983, *The Fractal Geometry of Nature* (San Francisco: W.H. Freeman).

Peitgen, H.-O., and Saupe, D. (eds.) 1988, *The Science of Fractal Images* (New York: Springer-Verlag).

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 48 of 181

9.5 Roots of Polynomials

Here we present a few methods for finding roots of polynomials. These will serve for most practical problems involving polynomials of low-to-moderate degree or for well-conditioned polynomials of higher degree. Not as well appreciated as it ought to be is the fact that some polynomials are exceedingly ill-conditioned. The tiniest changes in a polynomial's coefficients can, in the worst case, send its roots sprawling all over the complex plane. (An infamous example due to Wilkinson is detailed by Acton [1].)

Recall that a polynomial of degree n will have n roots. The roots can be real or complex, and they might not be distinct. If the coefficients of the polynomial are real, then complex roots will occur in pairs that are conjugate, i.e., if $x_1 = a + bi$ is a root then $x_2 = a - bi$ will also be a root. When the coefficients are complex, the complex roots need not be related.

Multiple roots, or closely spaced roots, produce the most difficulty for numerical algorithms (see Figure 9.5.1). For example, $P(x) = (x - a)^2$ has a double real root at $x = a$. However, we cannot bracket the root by the usual technique of identifying neighborhoods where the function changes sign, nor will slope-following methods such as Newton-Raphson work well, because both the function and its derivative vanish at a multiple root. Newton-Raphson *may* work, but slowly, since large roundoff errors can occur. When a root is known in advance to be multiple, then special methods of attack are readily devised. Problems arise when (as is generally the case) we do not know in advance what pathology a root will display.

Deflation of Polynomials

When seeking several or all roots of a polynomial, the total effort can be significantly reduced by the use of *deflation*. As each root r is found, the polynomial is factored into a product involving the root and a reduced polynomial of degree one less than the original, i.e., $P(x) = (x - r)Q(x)$. Since the roots of Q are exactly the remaining roots of P, the effort of finding additional roots decreases, because we work with polynomials of lower and lower degree as we find successive roots. Even more important, with deflation we can avoid the blunder of having our iterative method converge twice to the same (nonmultiple) root instead of separately to two different roots.

Deflation, which amounts to synthetic division, is a simple operation that acts on the array of polynomial coefficients. The concise code for synthetic division by a monomial factor was given in §5.3 above. You can deflate complex roots either by converting that code to complex data type, or else — in the case of a polynomial with real coefficients but possibly complex roots — by deflating by a quadratic factor,

$$[x - (a + ib)]\,[x - (a - ib)] = x^2 - 2ax + (a^2 + b^2) \tag{9.5.1}$$

The routine `poldiv` in §5.3 can be used to divide the polynomial by this factor.

Deflation must, however, be utilized with care. Because each new root is known with only finite accuracy, errors creep into the determination of the coefficients of the successively deflated polynomial. Consequently, the roots can become more and more inaccurate. It matters a lot whether the inaccuracy creeps in stably (plus or

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 49 of 181

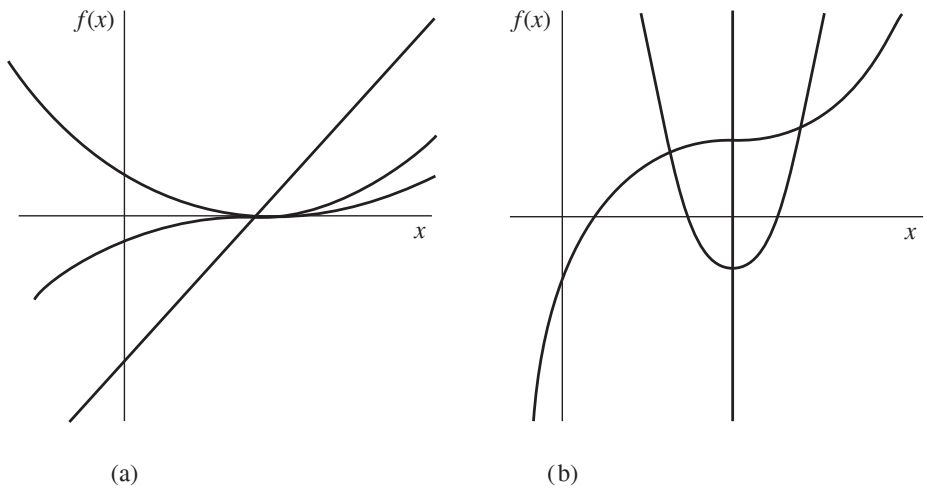

Figure 9.5.1. (a) Linear, quadratic, and cubic behavior at the roots of polynomials. Only under high magnification (b) does it become apparent that the cubic has one, not three, roots, and that the quadratic has two roots rather than none.

minus a few multiples of the machine precision at each stage) or unstably (erosion of successive significant figures until the results become meaningless). Which behavior occurs depends on just how the root is divided out. *Forward deflation*, where the new polynomial coefficients are computed in the order from the highest power of x down to the constant term, was illustrated in §5.3. This turns out to be stable if the root of smallest absolute value is divided out at each stage. Alternatively, one can do *backward deflation*, where new coefficients are computed in order from the constant term up to the coefficient of the highest power of x. This is stable if the remaining root of *largest* absolute value is divided out at each stage.

A polynomial whose coefficients are interchanged "end-to-end," so that the constant becomes the highest coefficient, etc., has its roots mapped into their reciprocals. (Proof: Divide the whole polynomial by its highest power x^n and rewrite it as a polynomial in $1/x$.) The algorithm for backward deflation is therefore virtually identical to that of forward deflation, except that the original coefficients are taken in reverse order and the reciprocal of the deflating root is used. Since we will use forward deflation below, we leave to you the exercise of writing a concise coding for backward deflation (as in §5.3). For more on the stability of deflation, consult [2].

To minimize the impact of increasing errors (even stable ones) when using deflation, it is advisable to treat roots of the successively deflated polynomials as only *tentative* roots of the original polynomial. One then *polishes* these tentative roots by taking them as initial guesses that are to be re-solved for, using the *nondeflated* original polynomial P. Again you must beware lest two deflated roots are inaccurate enough that, under polishing, they both converge to the same undeflated root; in that case you gain a spurious root-multiplicity and lose a distinct root. This is detectable, since you can compare each polished root for equality to previous ones from distinct tentative roots. When it happens, you are advised to deflate the polynomial just once (and for this root only), then again polish the tentative root, or to use Maehly's procedure (see equation 9.5.29 below).

Below we say more about techniques for polishing real and complex-conjugate

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 50 of 181

tentative roots. First, let's get back to overall strategy.

There are two schools of thought about how to proceed when faced with a polynomial of real coefficients. One school says to go after the easiest quarry, the real, distinct roots, by the same kinds of methods that we have discussed in previous sections for general functions, i.e., trial-and-error bracketing followed by a safe Newton-Raphson as in `rtsafe`. Sometimes you are *only* interested in real roots, in which case the strategy is complete. Otherwise, you then go after quadratic factors of the form (9.5.1) by any of a variety of methods. One such is Bairstow's method, which we will discuss below in the context of root polishing. Another is Muller's method, which we here briefly discuss.

Muller's Method

Muller's method generalizes the secant method, but uses quadratic interpolation among three points instead of linear interpolation between two. Solving for the zeros of the quadratic allows the method to find complex pairs of roots. Given *three* previous guesses for the root x_{i-2}, x_{i-1}, x_i, and the values of the polynomial $P(x)$ at those points, the next approximation x_{i+1} is produced by the following formulas,

$$\begin{aligned} q &\equiv \frac{x_i - x_{i-1}}{x_{i-1} - x_{i-2}} \\ A &\equiv qP(x_i) - q(1+q)P(x_{i-1}) + q^2 P(x_{i-2}) \\ B &\equiv (2q+1)P(x_i) - (1+q)^2 P(x_{i-1}) + q^2 P(x_{i-2}) \\ C &\equiv (1+q)P(x_i) \end{aligned} \tag{9.5.2}$$

followed by

$$x_{i+1} = x_i - (x_i - x_{i-1}) \left[\frac{2C}{B \pm \sqrt{B^2 - 4AC}} \right] \tag{9.5.3}$$

where the sign in the denominator is chosen to make its absolute value or modulus as large as possible. You can start the iterations with any three values of x that you like, e.g., three equally spaced values on the real axis. Note that you must allow for the possibility of a complex denominator, and subsequent complex arithmetic, in implementing the method.

Muller's method is sometimes also used for finding complex zeros of analytic functions (not just polynomials) in the complex plane, for example in the IMSL routine `ZANLY` [3].

Laguerre's Method

The second school regarding overall strategy happens to be the one to which we belong. That school advises you to use one of a very small number of methods that will converge (though with greater or lesser efficiency) to all types of roots: real, complex, single, or multiple. Use such a method to get tentative values for all n roots of your nth degree polynomial. Then go back and polish them as you desire.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 51 of 181

Laguerre's method is by far the most straightforward of these general, complex methods. It does require complex arithmetic, even while converging to real roots; however, for polynomials with all real roots, it is guaranteed to converge to a root from any starting point. For polynomials with some complex roots, little is theoretically proved about the method's convergence. Much empirical experience, however, suggests that nonconvergence is extremely unusual, and, further, can almost always be fixed by a simple scheme to break a nonconverging limit cycle. (This is implemented in our routine, below.) An example of a polynomial that requires this cycle-breaking scheme is one of high degree ($\gtrsim 20$), with all its roots just outside of the complex unit circle, approximately equally spaced around it. When the method converges on a simple complex zero, it is known that its convergence is third order.

In some instances the complex arithmetic in the Laguerre method is no disadvantage, since the polynomial itself may have complex coefficients.

To motivate (although not rigorously derive) the Laguerre formulas we can note the following relations between the polynomial and its roots and derivatives

$$P_n(x) = (x - x_1)(x - x_2)\dots(x - x_n) \tag{9.5.4}$$

$$\ln|P_n(x)| = \ln|x - x_1| + \ln|x - x_2| + \dots + \ln|x - x_n| \tag{9.5.5}$$

$$\frac{d\ln|P_n(x)|}{dx} = +\frac{1}{x - x_1} + \frac{1}{x - x_2} + \dots + \frac{1}{x - x_n} = \frac{P_n'}{P_n} \equiv G \tag{9.5.6}$$

$$\begin{aligned} -\frac{d^2\ln|P_n(x)|}{dx^2} &= +\frac{1}{(x - x_1)^2} + \frac{1}{(x - x_2)^2} + \dots + \frac{1}{(x - x_n)^2} \\ &= \left[\frac{P_n'}{P_n}\right]^2 - \frac{P_n''}{P_n} \equiv H \end{aligned} \tag{9.5.7}$$

Starting from these relations, the Laguerre formulas make what Acton [1] nicely calls "a rather drastic set of assumptions": The root x_1 that we seek is assumed to be located some distance a from our current guess x, while *all other roots* are assumed to be located at a distance b

$$x - x_1 = a \quad ; \quad x - x_i = b \quad i = 2, 3, \dots, n \tag{9.5.8}$$

Then we can express (9.5.6), (9.5.7) as

$$\frac{1}{a} + \frac{n-1}{b} = G \tag{9.5.9}$$

$$\frac{1}{a^2} + \frac{n-1}{b^2} = H \tag{9.5.10}$$

which yields as the solution for a

$$a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}} \tag{9.5.11}$$

where the sign should be taken to yield the largest magnitude for the denominator. Since the factor inside the square root can be negative, a can be complex. (A more rigorous justification of equation 9.5.11 is in [4].)

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 52 of 181

The method operates iteratively: For a trial value x, a is calculated by equation (9.5.11). Then $x - a$ becomes the next trial value. This continues until a is sufficiently small.

The following routine implements the Laguerre method to find one root of a given polynomial of degree m, whose coefficients can be complex. As usual, the first coefficient a[0] is the constant term, while a[m] is the coefficient of the highest power of x. The routine implements a simplified version of an elegant stopping criterion due to Adams [5], which neatly balances the desire to achieve full machine accuracy, on the one hand, with the danger of iterating forever in the presence of roundoff error, on the other.

```
#include <math.h>
#include "complex.h"
#include "nrutil.h"
#define EPSS 1.0e-7
#define MR 8
#define MT 10
#define MAXIT (MT*MR)
```

Here EPSS is the estimated fractional roundoff error. We try to break (rare) limit cycles with MR different fractional values, once every MT steps, for MAXIT total allowed iterations.

```
void laguer(fcomplex a[], int m, fcomplex *x, int *its)
```

Given the degree m and the m+1 complex coefficients a[0..m] of the polynomial $\sum_{i=0}^{\mathtt{m}} \mathtt{a}[i]x^i$, and given a complex value x, this routine improves x by Laguerre's method until it converges, within the achievable roundoff limit, to a root of the given polynomial. The number of iterations taken is returned as its.

```
{
    int iter,j;
    float abx,abp,abm,err;
    fcomplex dx,x1,b,d,f,g,h,sq,gp,gm,g2;
    static float frac[MR+1] = {0.0,0.5,0.25,0.75,0.13,0.38,0.62,0.88,1.0};
    Fractions used to break a limit cycle.

    for (iter=1;iter<=MAXIT;iter++) {       Loop over iterations up to allowed maximum.
        *its=iter;
        b=a[m];
        err=Cabs(b);
        d=f=Complex(0.0,0.0);
        abx=Cabs(*x);
        for (j=m-1;j>=0;j--) {              Efficient computation of the polynomial and
            f=Cadd(Cmul(*x,f),d);               its first two derivatives. f stores P''/2.
            d=Cadd(Cmul(*x,d),b);
            b=Cadd(Cmul(*x,b),a[j]);
            err=Cabs(b)+abx*err;
        }
        err *= EPSS;
        Estimate of roundoff error in evaluating polynomial.
        if (Cabs(b) <= err) return;         We are on the root.
        g=Cdiv(d,b);                        The generic case: use Laguerre's formula.
        g2=Cmul(g,g);
        h=Csub(g2,RCmul(2.0,Cdiv(f,b)));
        sq=Csqrt(RCmul((float) (m-1),Csub(RCmul((float) m,h),g2)));
        gp=Cadd(g,sq);
        gm=Csub(g,sq);
        abp=Cabs(gp);
        abm=Cabs(gm);
        if (abp < abm) gp=gm;
        dx=((FMAX(abp,abm) > 0.0 ? Cdiv(Complex((float) m,0.0),gp)
            : RCmul(1+abx,Complex(cos((float)iter),sin((float)iter)))));
        x1=Csub(*x,dx);
```

```
        if (x->r == x1.r && x->i == x1.i) return;        Converged.
        if (iter % MT) *x=x1;
        else *x=Csub(*x,RCmul(frac[iter/MT],dx));
        Every so often we take a fractional step, to break any limit cycle (itself a rare occur-
        rence).
    }
    nrerror("too many iterations in laguer");
    Very unusual — can occur only for complex roots. Try a different starting guess for the
    root.
    return;
}
```

Here is a driver routine that calls `laguer` in succession for each root, performs the deflation, optionally polishes the roots by the same Laguerre method — if you are not going to polish in some other way — and finally sorts the roots by their real parts. (We will use this routine in Chapter 13.)

```
#include <math.h>
#include "complex.h"
#define EPS 2.0e-6
#define MAXM 100
A small number, and maximum anticipated value of m.

void zroots(fcomplex a[], int m, fcomplex roots[], int polish)
Given the degree m and the m+1 complex coefficients a[0..m] of the polynomial Σ_{i=0}^{m} a(i)x^i,
this routine successively calls laguer and finds all m complex roots in roots[1..m]. The
boolean variable polish should be input as true (1) if polishing (also by Laguerre's method)
is desired, false (0) if the roots will be subsequently polished by other means.
{
    void laguer(fcomplex a[], int m, fcomplex *x, int *its);
    int i,its,j,jj;
    fcomplex x,b,c,ad[MAXM];

    for (j=0;j<=m;j++) ad[j]=a[j];          Copy of coefficients for successive deflation.
    for (j=m;j>=1;j--) {                    Loop over each root to be found.
        x=Complex(0.0,0.0);                 Start at zero to favor convergence to small-
        laguer(ad,j,&x,&its);                   est remaining root, and find the root.
        if (fabs(x.i) <= 2.0*EPS*fabs(x.r)) x.i=0.0;
        roots[j]=x;
        b=ad[j];                            Forward deflation.
        for (jj=j-1;jj>=0;jj--) {
            c=ad[jj];
            ad[jj]=b;
            b=Cadd(Cmul(x,b),c);
        }
    }
    if (polish)
        for (j=1;j<=m;j++)                  Polish the roots using the undeflated coeffi-
            laguer(a,m,&roots[j],&its);         cients.
    for (j=2;j<=m;j++) {                    Sort roots by their real parts by straight in-
        x=roots[j];                             sertion.
        for (i=j-1;i>=1;i--) {
            if (roots[i].r <= x.r) break;
            roots[i+1]=roots[i];
        }
        roots[i+1]=x;
    }
}
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 54 of 181

Eigenvalue Methods

The eigenvalues of a matrix $\mathbf{A}$ are the roots of the "characteristic polynomial" $P(x) = \det[\mathbf{A} - x\mathbf{I}]$. However, as we will see in Chapter 11, root-finding is not generally an efficient way to find eigenvalues. Turning matters around, we can use the more efficient eigenvalue methods that are discussed in Chapter 11 to find the roots of arbitrary polynomials. You can easily verify (see, e.g., [6]) that the characteristic polynomial of the special $m \times m$ *companion matrix*

$$\mathbf{A} = \begin{pmatrix} -\frac{a_{m-1}}{a_m} & -\frac{a_{m-2}}{a_m} & \cdots & -\frac{a_1}{a_m} & -\frac{a_0}{a_m} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \tag{9.5.12}$$

is equivalent to the general polynomial

$$P(x) = \sum_{i=0}^{m} a_i x^i \tag{9.5.13}$$

If the coefficients a_i are real, rather than complex, then the eigenvalues of $\mathbf{A}$ can be found using the routines `balanc` and `hqr` in §§11.5–11.6 (see discussion there). This method, implemented in the routine `zrhqr` following, is typically about a factor 2 slower than `zroots` (above). However, for some classes of polynomials, it is a more robust technique, largely because of the fairly sophisticated convergence methods embodied in `hqr`. If your polynomial has real coefficients, and you are having trouble with `zroots`, then `zrhqr` is a recommended alternative.

```
#include "nrutil.h"
#define MAXM 50

void zrhqr(float a[], int m, float rtr[], float rti[])
```

Find all the roots of a polynomial with real coefficients, $\sum_{i=0}^{\mathtt{m}} \mathtt{a}(i)x^i$, given the degree `m` and the coefficients `a[0..m]`. The method is to construct an upper Hessenberg matrix whose eigenvalues are the desired roots, and then use the routines `balanc` and `hqr`. The real and imaginary parts of the roots are returned in `rtr[1..m]` and `rti[1..m]`, respectively.

```
{
    void balanc(float **a, int n);
    void hqr(float **a, int n, float wr[], float wi[]);
    int j,k;
    float **hess,xr,xi;

    hess=matrix(1,MAXM,1,MAXM);
    if (m > MAXM || a[m] == 0.0) nrerror("bad args in zrhqr");
    for (k=1;k<=m;k++) {        Construct the matrix.
        hess[1][k] = -a[m-k]/a[m];
        for (j=2;j<=m;j++) hess[j][k]=0.0;
        if (k != m) hess[k+1][k]=1.0;
    }
    balanc(hess,m);             Find its eigenvalues.
    hqr(hess,m,rtr,rti);
    for (j=2;j<=m;j++) {        Sort roots by their real parts by straight insertion.
        xr=rtr[j];
        xi=rti[j];
```

```
        for (k=j-1;k>=1;k--) {
            if (rtr[k] <= xr) break;
            rtr[k+1]=rtr[k];
            rti[k+1]=rti[k];
        }
        rtr[k+1]=xr;
        rti[k+1]=xi;
    }
    free_matrix(hess,1,MAXM,1,MAXM);
}
```

Other Sure-Fire Techniques

The *Jenkins-Traub method* has become practically a standard in black-box polynomial root-finders, e.g., in the IMSL library [3]. The method is too complicated to discuss here, but is detailed, with references to the primary literature, in [4].

The *Lehmer-Schur algorithm* is one of a class of methods that isolate roots in the complex plane by generalizing the notion of one-dimensional bracketing. It is possible to determine efficiently whether there are any polynomial roots within a circle of given center and radius. From then on it is a matter of bookkeeping to hunt down all the roots by a series of decisions regarding where to place new trial circles. Consult [1] for an introduction.

Techniques for Root-Polishing

Newton-Raphson works very well for real roots once the neighborhood of a root has been identified. The polynomial and its derivative can be efficiently simultaneously evaluated as in §5.3. For a polynomial of degree `n` with coefficients `c[0]...c[n]`, the following segment of code embodies one cycle of Newton-Raphson:

```
p=c[n]*x+c[n-1];
p1=c[n];
for(i=n-2;i>=0;i--) {
    p1=p+p1*x;
    p=c[i]+p*x;
}
if (p1 == 0.0) nrerror("derivative should not vanish");
x -= p/p1;
```

Once all real roots of a polynomial have been polished, one must polish the complex roots, either directly, or by looking for quadratic factors.

Direct polishing by Newton-Raphson is straightforward for complex roots if the above code is converted to complex data types. With real polynomial coefficients, note that your starting guess (tentative root) *must* be off the real axis, otherwise you will never get off that axis — and may get shot off to infinity by a minimum or maximum of the polynomial.

For real polynomials, the alternative means of polishing complex roots (or, for that matter, double real roots) is *Bairstow's method*, which seeks quadratic factors. The advantage

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 56 of 181

of going after quadratic factors is that it avoids all complex arithmetic. Bairstow's method seeks a quadratic factor that embodies the two roots $x = a \pm ib$, namely

$$x^2 - 2ax + (a^2 + b^2) \equiv x^2 + Bx + C \tag{9.5.14}$$

In general if we divide a polynomial by a quadratic factor, there will be a linear remainder

$$P(x) = (x^2 + Bx + C)Q(x) + Rx + S. \tag{9.5.15}$$

Given B and C, R and S can be readily found, by polynomial division (§5.3). We can consider R and S to be adjustable functions of B and C, and they will be zero if the quadratic factor is a divisor of $P(x)$.

In the neighborhood of a root a first-order Taylor series expansion approximates the variation of R, S with respect to small changes in B, C

$$R(B + \delta B, C + \delta C) \approx R(B, C) + \frac{\partial R}{\partial B}\delta B + \frac{\partial R}{\partial C}\delta C \tag{9.5.16}$$

$$S(B + \delta B, C + \delta C) \approx S(B, C) + \frac{\partial S}{\partial B}\delta B + \frac{\partial S}{\partial C}\delta C \tag{9.5.17}$$

To evaluate the partial derivatives, consider the derivative of (9.5.15) with respect to C. Since $P(x)$ is a fixed polynomial, it is independent of C, hence

$$0 = (x^2 + Bx + C)\frac{\partial Q}{\partial C} + Q(x) + \frac{\partial R}{\partial C}x + \frac{\partial S}{\partial C} \tag{9.5.18}$$

which can be rewritten as

$$-Q(x) = (x^2 + Bx + C)\frac{\partial Q}{\partial C} + \frac{\partial R}{\partial C}x + \frac{\partial S}{\partial C} \tag{9.5.19}$$

Similarly, $P(x)$ is independent of B, so differentiating (9.5.15) with respect to B gives

$$-xQ(x) = (x^2 + Bx + C)\frac{\partial Q}{\partial B} + \frac{\partial R}{\partial B}x + \frac{\partial S}{\partial B} \tag{9.5.20}$$

Now note that equation (9.5.19) matches equation (9.5.15) in form. Thus if we perform a second synthetic division of $P(x)$, i.e., a division of $Q(x)$, yielding a remainder R_1x+S_1, then

$$\frac{\partial R}{\partial C} = -R_1 \qquad \frac{\partial S}{\partial C} = -S_1 \tag{9.5.21}$$

To get the remaining partial derivatives, evaluate equation (9.5.20) at the two roots of the quadratic, x_+ and x_-. Since

$$Q(x_\pm) = R_1x_\pm + S_1 \tag{9.5.22}$$

we get

$$\frac{\partial R}{\partial B}x_+ + \frac{\partial S}{\partial B} = -x_+(R_1x_+ + S_1) \tag{9.5.23}$$

$$\frac{\partial R}{\partial B}x_- + \frac{\partial S}{\partial B} = -x_-(R_1x_- + S_1) \tag{9.5.24}$$

Solve these two equations for the partial derivatives, using

$$x_+ + x_- = -B \qquad x_+x_- = C \tag{9.5.25}$$

and find

$$\frac{\partial R}{\partial B} = BR_1 - S_1 \qquad \frac{\partial S}{\partial B} = CR_1 \tag{9.5.26}$$

Bairstow's method now consists of using Newton-Raphson in two dimensions (which is actually the subject of the *next* section) to find a simultaneous zero of R and S. Synthetic division is used twice per cycle to evaluate R, S and their partial derivatives with respect to B, C. Like one-dimensional Newton-Raphson, the method works well in the vicinity of a root pair (real or complex), but it can fail miserably when started at a random point. We therefore recommend it only in the context of polishing tentative complex roots.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 57 of 181

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 20                          At most ITMAX iterations.
#define TINY 1.0e-6

void qroot(float p[], int n, float *b, float *c, float eps)
Given n+1 coefficients p[0..n] of a polynomial of degree n, and trial values for the coefficients
of a quadratic factor x*x+b*x+c, improve the solution until the coefficients b,c change by less
than eps. The routine poldiv §5.3 is used.
{
    void poldiv(float u[], int n, float v[], int nv, float q[], float r[]);
    int iter;
    float sc,sb,s,rc,rb,r,dv,delc,delb;
    float *q,*qq,*rem;
    float d[3];

    q=vector(0,n);
    qq=vector(0,n);
    rem=vector(0,n);
    d[2]=1.0;
    for (iter=1;iter<=ITMAX;iter++) {
        d[1]=(*b);
        d[0]=(*c);
        poldiv(p,n,d,2,q,rem);
        s=rem[0];                         First division r,s.
        r=rem[1];
        poldiv(q,(n-1),d,2,qq,rem);
        sb = -(*c)*(rc = -rem[1]);        Second division partial r,s with respect to
        rb = -(*b)*rc+(sc = -rem[0]);         c.
        dv=1.0/(sb*rc-sc*rb);             Solve 2x2 equation.
        delb=(r*sc-s*rc)*dv;
        delc=(-r*sb+s*rb)*dv;
        *b += (delb=(r*sc-s*rc)*dv);
        *c += (delc=(-r*sb+s*rb)*dv);
        if ((fabs(delb) <= eps*fabs(*b) || fabs(*b) < TINY)
            && (fabs(delc) <= eps*fabs(*c) || fabs(*c) < TINY)) {
            free_vector(rem,0,n);         Coefficients converged.
            free_vector(qq,0,n);
            free_vector(q,0,n);
            return;
        }
    }
    nrerror("Too many iterations in routine qroot");
}
```

We have already remarked on the annoyance of having two tentative roots collapse to one value under polishing. You are left not knowing whether your polishing procedure has lost a root, or whether there *is* actually a double root, which was split only by roundoff errors in your previous deflation. One solution is deflate-and-repolish; but deflation is what we are trying to avoid at the polishing stage. An alternative is *Maehly's procedure*. Maehly pointed out that the derivative of the reduced polynomial

$$P_j(x) \equiv \frac{P(x)}{(x-x_1)\cdots(x-x_j)} \tag{9.5.27}$$

can be written as

$$P_j'(x) = \frac{P'(x)}{(x-x_1)\cdots(x-x_j)} - \frac{P(x)}{(x-x_1)\cdots(x-x_j)}\sum_{i=1}^{j}(x-x_i)^{-1} \tag{9.5.28}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 58 of 181

Hence one step of Newton-Raphson, taking a guess x_k into a new guess x_{k+1}, can be written as

$$x_{k+1} = x_k - \frac{P(x_k)}{P'(x_k) - P(x_k)\sum_{i=1}^{j}(x_k - x_i)^{-1}} \tag{9.5.29}$$

This equation, if used with i ranging over the roots already polished, will prevent a tentative root from spuriously hopping to another one's true root. It is an example of so-called *zero suppression* as an alternative to true deflation.

Muller's method, which was described above, can also be useful at the polishing stage.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 7. [1]

Peters G., and Wilkinson, J.H. 1971, *Journal of the Institute of Mathematics and its Applications*, vol. 8, pp. 16–35. [2]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [3]

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.9–8.13. [4]

Adams, D.A. 1967, *Communications of the ACM*, vol. 10, pp. 655–658. [5]

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §4.4.3. [6]

Henrici, P. 1974, *Applied and Computational Complex Analysis*, vol. 1 (New York: Wiley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§5.5–5.9.

9.6 Newton-Raphson Method for Nonlinear Systems of Equations

We make an extreme, but wholly defensible, statement: There are *no* good, general methods for solving systems of more than one nonlinear equation. Furthermore, it is not hard to see why (very likely) there *never will be* any good, general methods: Consider the case of two dimensions, where we want to solve simultaneously

$$\begin{aligned} f(x, y) &= 0 \\ g(x, y) &= 0 \end{aligned} \tag{9.6.1}$$

The functions f and g are two arbitrary functions, each of which has zero contour lines that divide the (x, y) plane into regions where their respective function is positive or negative. These zero contour boundaries are of interest to us. The solutions that we seek are those points (if any) that are common to the zero contours of f and g (see Figure 9.6.1). Unfortunately, the functions f and g have, in general, no relation to each other at all! There is nothing special about a common point from either f's point of view, or from g's. In order to find all common points, which are

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 59 of 181

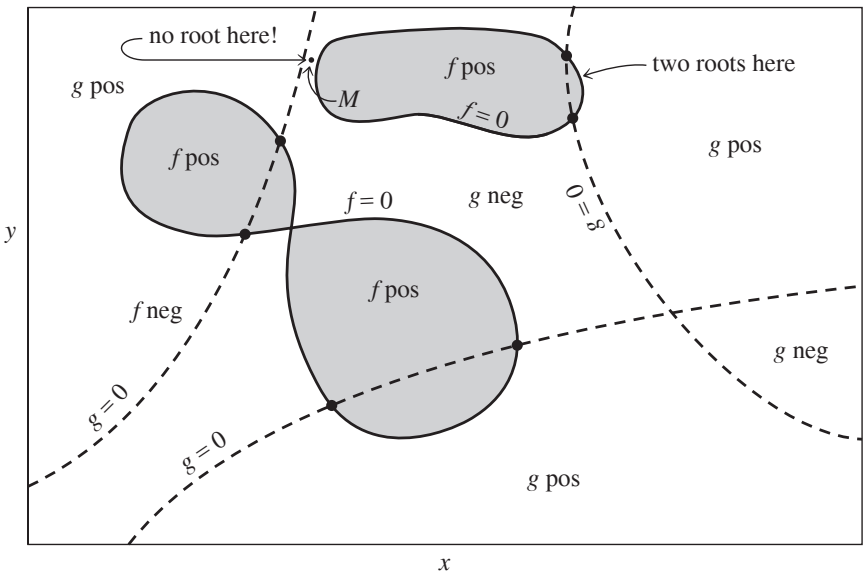

Figure 9.6.1. Solution of two nonlinear equations in two unknowns. Solid curves refer to $f(x, y)$, dashed curves to $g(x, y)$. Each equation divides the (x, y) plane into positive and negative regions, bounded by zero curves. The desired solutions are the intersections of these unrelated zero curves. The number of solutions is *a priori* unknown.

the solutions of our nonlinear equations, we will (in general) have to do neither more nor less than map out the full zero contours of both functions. Note further that the zero contours will (in general) consist of an unknown number of disjoint closed curves. How can we ever hope to know when we have found all such disjoint pieces?

For problems in more than two dimensions, we need to find points mutually common to N unrelated zero-contour hypersurfaces, each of dimension $N - 1$. You see that root finding becomes virtually impossible without insight! You will almost always have to use additional information, specific to your particular problem, to answer such basic questions as, "Do I expect a unique solution?" and "Approximately where?" Acton [1] has a good discussion of some of the particular strategies that can be tried.

In this section we will discuss the simplest multidimensional root finding method, Newton-Raphson. This method gives you a very efficient means of converging to a root, if you have a sufficiently good initial guess. It can also spectacularly fail to converge, indicating (though not proving) that your putative root does not exist nearby. In §9.7 we discuss more sophisticated implementations of the Newton-Raphson method, which try to improve on Newton-Raphson's poor global convergence. A multidimensional generalization of the secant method, called Broyden's method, is also discussed in §9.7.

A typical problem gives N functional relations to be zeroed, involving variables $x_i, i = 1, 2, \ldots, N$:

$$F_i(x_1, x_2, \ldots, x_N) = 0 \qquad i = 1, 2, \ldots, N. \tag{9.6.2}$$

We let $\mathbf{x}$ denote the entire vector of values x_i and $\mathbf{F}$ denote the entire vector of functions F_i. In the neighborhood of $\mathbf{x}$, each of the functions F_i can be expanded

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 60 of 181

in Taylor series

$$F_i(\mathbf{x}+\delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial F_i}{\partial x_j}\delta x_j + O(\delta\mathbf{x}^2). \tag{9.6.3}$$

The matrix of partial derivatives appearing in equation (9.6.3) is the *Jacobian* matrix $\mathbf{J}$:

$$J_{ij} \equiv \frac{\partial F_i}{\partial x_j}. \tag{9.6.4}$$

In matrix notation equation (9.6.3) is

$$\mathbf{F}(\mathbf{x}+\delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J}\cdot\delta\mathbf{x} + O(\delta\mathbf{x}^2). \tag{9.6.5}$$

By neglecting terms of order $\delta\mathbf{x}^2$ and higher and by setting $\mathbf{F}(\mathbf{x}+\delta\mathbf{x}) = 0$, we obtain a set of linear equations for the corrections $\delta\mathbf{x}$ that move each function closer to zero simultaneously, namely

$$\mathbf{J}\cdot\delta\mathbf{x} = -\mathbf{F}. \tag{9.6.6}$$

Matrix equation (9.6.6) can be solved by LU decomposition as described in §2.3. The corrections are then added to the solution vector,

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \delta\mathbf{x} \tag{9.6.7}$$

and the process is iterated to convergence. In general it is a good idea to check the degree to which both functions and variables have converged. Once either reaches machine accuracy, the other won't change.

The following routine `mnewt` performs `ntrial` iterations starting from an initial guess at the solution vector `x[1..n]`. Iteration stops if either the sum of the magnitudes of the functions F_i is less than some tolerance `tolf`, or the sum of the absolute values of the corrections to δx_i is less than some tolerance `tolx`. `mnewt` calls a user supplied function `usrfun` which must provide the function values $\mathbf{F}$ and the Jacobian matrix $\mathbf{J}$. If $\mathbf{J}$ is difficult to compute analytically, you can try having `usrfun` call the routine `fdjac` of §9.7 to compute the partial derivatives by finite differences. You should not make `ntrial` too big; rather inspect to see what is happening before continuing for some further iterations.

```
#include <math.h>
#include "nrutil.h"

void usrfun(float *x,int n,float *fvec,float **fjac);
#define FREERETURN {free_matrix(fjac,1,n,1,n);free_vector(fvec,1,n);\
    free_vector(p,1,n);free_ivector(indx,1,n);return;}

void mnewt(int ntrial, float x[], int n, float tolx, float tolf)
Given an initial guess x[1..n] for a root in n dimensions, take ntrial Newton-Raphson steps
to improve the root. Stop if the root converges in either summed absolute variable increments
tolx or summed absolute function values tolf.
{
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
```

```
    int k,i,*indx;
    float errx,errf,d,*fvec,**fjac,*p;

    indx=ivector(1,n);
    p=vector(1,n);
    fvec=vector(1,n);
    fjac=matrix(1,n,1,n);
    for (k=1;k<=ntrial;k++) {
        usrfun(x,n,fvec,fjac);          User function supplies function values at x in
        errf=0.0;                           fvec and Jacobian matrix in fjac.
        for (i=1;i<=n;i++) errf += fabs(fvec[i]);     Check function convergence.
        if (errf <= tolf) FREERETURN
        for (i=1;i<=n;i++) p[i] = -fvec[i];         Right-hand side of linear equations.
        ludcmp(fjac,n,indx,&d);         Solve linear equations using LU decomposition.
        lubksb(fjac,n,indx,p);
        errx=0.0;                       Check root convergence.
        for (i=1;i<=n;i++) {            Update solution.
            errx += fabs(p[i]);
            x[i] += p[i];
        }
        if (errx <= tolx) FREERETURN
    }
    FREERETURN
}
```

Newton's Method versus Minimization

In the next chapter, we will find that there *are* efficient general techniques for finding a minimum of a function of many variables. Why is that task (relatively) easy, while multidimensional root finding is often quite hard? Isn't minimization equivalent to finding a zero of an N-dimensional gradient vector, not so different from zeroing an N-dimensional function? No! The components of a gradient vector are not independent, arbitrary functions. Rather, they obey so-called integrability conditions that are highly restrictive. Put crudely, you can always find a minimum by sliding downhill on a single surface. The test of "downhillness" is thus one-dimensional. There is no analogous conceptual procedure for finding a multidimensional root, where "downhill" must mean simultaneously downhill in N separate function spaces, thus allowing a multitude of trade-offs, as to how much progress in one dimension is worth compared with progress in another.

It might occur to you to carry out multidimensional root finding by collapsing all these dimensions into one: Add up the sums of squares of the individual functions F_i to get a master function F which (i) is positive definite, and (ii) has a global minimum of zero exactly at all solutions of the original set of nonlinear equations. Unfortunately, as you will see in the next chapter, the efficient algorithms for finding minima come to rest on global and local minima indiscriminately. You will often find, to your great dissatisfaction, that your function F has a great number of local minima. In Figure 9.6.1, for example, there is likely to be a local minimum wherever the zero contours of f and g make a close approach to each other. The point labeled M is such a point, and one sees that there are no nearby roots.

However, we will now see that sophisticated strategies for multidimensional root finding can in fact make use of the idea of minimizing a master function F, by *combining* it with Newton's method applied to the full set of functions F_i. While such methods can still occasionally fail by coming to rest on a local minimum of

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 62 of 181

F, they often succeed where a direct attack via Newton's method alone fails. The next section deals with these methods.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 14. [1]

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press).

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press).

9.7 Globally Convergent Methods for Nonlinear Systems of Equations

We have seen that Newton's method for solving nonlinear equations has an unfortunate tendency to wander off into the wild blue yonder if the initial guess is not sufficiently close to the root. A *global* method is one that converges to a solution from almost any starting point. In this section we will develop an algorithm that combines the rapid local convergence of Newton's method with a globally convergent strategy that will guarantee some progress towards the solution at each iteration. The algorithm is closely related to the quasi-Newton method of minimization which we will describe in §10.7.

Recall our discussion of §9.6: the Newton step for the set of equations

$$\mathbf{F}(\mathbf{x}) = 0 \tag{9.7.1}$$

is

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \delta\mathbf{x} \tag{9.7.2}$$

where

$$\delta\mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F} \tag{9.7.3}$$

Here $\mathbf{J}$ is the Jacobian matrix. How do we decide whether to accept the Newton step $\delta\mathbf{x}$? A reasonable strategy is to require that the step decrease $|\mathbf{F}|^2 = \mathbf{F} \cdot \mathbf{F}$. This is the same requirement we would impose if we were trying to minimize

$$f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F} \tag{9.7.4}$$

(The $\frac{1}{2}$ is for later convenience.) Every solution to (9.7.1) minimizes (9.7.4), but there may be local minima of (9.7.4) that are not solutions to (9.7.1). Thus, as already mentioned, simply applying one of our minimum finding algorithms from Chapter 10 to (9.7.4) is *not* a good idea.

To develop a better strategy, note that the Newton step (9.7.3) is a *descent direction* for f:

$$\nabla f \cdot \delta\mathbf{x} = (\mathbf{F} \cdot \mathbf{J}) \cdot (-\mathbf{J}^{-1} \cdot \mathbf{F}) = -\mathbf{F} \cdot \mathbf{F} < 0 \tag{9.7.5}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 63 of 181

Thus our strategy is quite simple: We always first try the full Newton step, because once we are close enough to the solution we will get quadratic convergence. However, we check at each iteration that the proposed step reduces f. If not, we *backtrack* along the Newton direction until we have an acceptable step. Because the Newton step is a descent direction for f, we are guaranteed to find an acceptable step by backtracking. We will discuss the backtracking algorithm in more detail below.

Note that this method essentially minimizes f by taking Newton steps designed to bring $\mathbf{F}$ to zero. This is *not* equivalent to minimizing f directly by taking Newton steps designed to bring ∇f to zero. While the method can still occasionally fail by landing on a local minimum of f, this is quite rare in practice. The routine `newt` below will warn you if this happens. The remedy is to try a new starting point.

Line Searches and Backtracking

When we are not close enough to the minimum of f, taking the full Newton step $\mathbf{p} = \delta\mathbf{x}$ need not decrease the function; we may move too far for the quadratic approximation to be valid. All we are guaranteed is that *initially* f decreases as we move in the Newton direction. So the goal is to move to a new point $\mathbf{x}_{\rm new}$ along the *direction* of the Newton step $\mathbf{p}$, but not necessarily all the way:

$$\mathbf{x}_{\rm new} = \mathbf{x}_{\rm old} + \lambda\mathbf{p}, \qquad 0 < \lambda \le 1 \tag{9.7.6}$$

The aim is to find λ so that $f(\mathbf{x}_{\rm old} + \lambda\mathbf{p})$ has decreased sufficiently. Until the early 1970s, standard practice was to choose λ so that $\mathbf{x}_{\rm new}$ exactly minimizes f in the direction $\mathbf{p}$. However, we now know that it is extremely wasteful of function evaluations to do so. A better strategy is as follows: Since $\mathbf{p}$ is always the Newton direction in our algorithms, we first try $\lambda = 1$, the full Newton step. This will lead to quadratic convergence when $\mathbf{x}$ is sufficiently close to the solution. However, if $f(\mathbf{x}_{\rm new})$ does not meet our acceptance criteria, we *backtrack* along the Newton direction, trying a smaller value of λ, until we find a suitable point. Since the Newton direction is a descent direction, we are guaranteed to decrease f for sufficiently small λ.

What should the criterion for accepting a step be? It is *not* sufficient to require merely that $f(\mathbf{x}_{\rm new}) < f(\mathbf{x}_{\rm old})$. This criterion can fail to converge to a minimum of f in one of two ways. First, it is possible to construct a sequence of steps satisfying this criterion with f decreasing too slowly relative to the step lengths. Second, one can have a sequence where the step lengths are too small relative to the initial rate of decrease of f. (For examples of such sequences, see [1], p. 117.)

A simple way to fix the first problem is to require the *average* rate of decrease of f to be at least some fraction α of the *initial* rate of decrease $\nabla f \cdot \mathbf{p}$:

$$f(\mathbf{x}_{\rm new}) \le f(\mathbf{x}_{\rm old}) + \alpha\nabla f \cdot (\mathbf{x}_{\rm new} - \mathbf{x}_{\rm old}) \tag{9.7.7}$$

Here the parameter α satisfies $0 < \alpha < 1$. We can get away with quite small values of α; $\alpha = 10^{-4}$ is a good choice.

The second problem can be fixed by requiring the rate of decrease of f at $\mathbf{x}_{\rm new}$ to be greater than some fraction β of the rate of decrease of f at $\mathbf{x}_{\rm old}$. In practice, we will not need to impose this second constraint because our backtracking algorithm will have a built-in cutoff to avoid taking steps that are too small.

Here is the strategy for a practical backtracking routine: Define

$$g(\lambda) \equiv f(\mathbf{x}_{\rm old} + \lambda\mathbf{p}) \tag{9.7.8}$$

so that

$$g'(\lambda) = \nabla f \cdot \mathbf{p} \tag{9.7.9}$$

If we need to backtrack, then we model g with the most current information we have and choose λ to minimize the model. We start with $g(0)$ and $g'(0)$ available. The first step is

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 64 of 181

always the Newton step, $\lambda = 1$. If this step is not acceptable, we have available $g(1)$ as well. We can therefore model $g(\lambda)$ as a quadratic:

$$g(\lambda) \approx [g(1) - g(0) - g'(0)]\lambda^2 + g'(0)\lambda + g(0) \tag{9.7.10}$$

Taking the derivative of this quadratic, we find that it is a minimum when

$$\lambda = -\frac{g'(0)}{2[g(1) - g(0) - g'(0)]} \tag{9.7.11}$$

Since the Newton step failed, we can show that $\lambda \lesssim \frac{1}{2}$ for small α. We need to guard against too small a value of λ, however. We set $\lambda_{\min} = 0.1$.

On second and subsequent backtracks, we model g as a cubic in λ, using the previous value $g(\lambda_1)$ and the second most recent value $g(\lambda_2)$:

$$g(\lambda) = a\lambda^3 + b\lambda^2 + g'(0)\lambda + g(0) \tag{9.7.12}$$

Requiring this expression to give the correct values of g at λ_1 and λ_2 gives two equations that can be solved for the coefficients a and b:

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\lambda_1 - \lambda_2} \begin{bmatrix} 1/\lambda_1^2 & -1/\lambda_2^2 \\ -\lambda_2/\lambda_1^2 & \lambda_1/\lambda_2^2 \end{bmatrix} \cdot \begin{bmatrix} g(\lambda_1) - g'(0)\lambda_1 - g(0) \\ g(\lambda_2) - g'(0)\lambda_2 - g(0) \end{bmatrix} \tag{9.7.13}$$

The minimum of the cubic (9.7.12) is at

$$\lambda = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a} \tag{9.7.14}$$

We enforce that λ lie between $\lambda_{\max} = 0.5\lambda_1$ and $\lambda_{\min} = 0.1\lambda_1$.

The routine has two additional features, a minimum step length `alamin` and a maximum step length `stpmax`. `lnsrch` will also be used in the quasi-Newton minimization routine `dfpmin` in the next section.

```
#include <math.h>
#include "nrutil.h"
#define ALF 1.0e-4                    Ensures sufficient decrease in function value.
#define TOLX 1.0e-7                   Convergence criterion on Δx.

void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
    float *f, float stpmax, int *check, float (*func)(float []))
Given an n-dimensional point xold[1..n], the value of the function and gradient there, fold
and g[1..n], and a direction p[1..n], finds a new point x[1..n] along the direction p from
xold where the function func has decreased "sufficiently." The new function value is returned
in f. stpmax is an input quantity that limits the length of the steps so that you do not try to
evaluate the function in regions where it is undefined or subject to overflow. p is usually the
Newton direction. The output quantity check is false (0) on a normal exit. It is true (1) when
x is too close to xold. In a minimization algorithm, this usually signals convergence and can
be ignored. However, in a zero-finding algorithm the calling program should check whether the
convergence is spurious. Some "difficult" problems may require double precision in this routine.
{
    int i;
    float a,alam,alam2,alamin,b,disc,f2,rhs1,rhs2,slope,sum,temp,
        test,tmplam;

    *check=0;
    for (sum=0.0,i=1;i<=n;i++) sum += p[i]*p[i];
    sum=sqrt(sum);
    if (sum > stpmax)
        for (i=1;i<=n;i++) p[i] *= stpmax/sum;    Scale if attempted step is too big.
    for (slope=0.0,i=1;i<=n;i++)
        slope += g[i]*p[i];
    if (slope >= 0.0) nrerror("Roundoff problem in lnsrch.");
    test=0.0;                                     Compute λmin.
    for (i=1;i<=n;i++) {
```

```
        temp=fabs(p[i])/FMAX(fabs(xold[i]),1.0);
        if (temp > test) test=temp;
    }
    alamin=TOLX/test;
    alam=1.0;                                   Always try full Newton step first.
    for (;;) {                                  Start of iteration loop.
        for (i=1;i<=n;i++) x[i]=xold[i]+alam*p[i];
        *f=(*func)(x);
        if (alam < alamin) {                    Convergence on Δx. For zero find-
            for (i=1;i<=n;i++) x[i]=xold[i];        ing, the calling program should
            *check=1;                               verify the convergence.
            return;
        } else if (*f <= fold+ALF*alam*slope) return;      Sufficient function decrease.
        else {                                             Backtrack.
            if (alam == 1.0)
                tmplam = -slope/(2.0*(*f-fold-slope));     First time.
            else {                                         Subsequent backtracks.
                rhs1 = *f-fold-alam*slope;
                rhs2=f2-fold-alam2*slope;
                a=(rhs1/(alam*alam)-rhs2/(alam2*alam2))/(alam-alam2);
                b=(-alam2*rhs1/(alam*alam)+alam*rhs2/(alam2*alam2))/(alam-alam2);
                if (a == 0.0) tmplam = -slope/(2.0*b);
                else {
                    disc=b*b-3.0*a*slope;
                    if (disc < 0.0) tmplam=0.5*alam;
                    else if (b <= 0.0) tmplam=(-b+sqrt(disc))/(3.0*a);
                    else tmplam=-slope/(b+sqrt(disc));
                }
                if (tmplam > 0.5*alam)
                    tmplam=0.5*alam;            λ ≤ 0.5λ1.
            }
        }
        alam2=alam;
        f2 = *f;
        alam=FMAX(tmplam,0.1*alam);             λ ≥ 0.1λ1.
    }                                           Try again.
}
```

Here now is the globally convergent Newton routine `newt` that uses `lnsrch`. A feature of `newt` is that you need not supply the Jacobian matrix analytically; the routine will attempt to compute the necessary partial derivatives of **F** by finite differences in the routine `fdjac`. This routine uses some of the techniques described in §5.7 for computing numerical derivatives. Of course, you can always replace `fdjac` with a routine that calculates the Jacobian analytically if this is easy for you to do.

```
#include <math.h>
#include "nrutil.h"
#define MAXITS 200
#define TOLF 1.0e-4
#define TOLMIN 1.0e-6
#define TOLX 1.0e-7
#define STPMX 100.0
```

Here `MAXITS` is the maximum number of iterations; `TOLF` sets the convergence criterion on function values; `TOLMIN` sets the criterion for deciding whether spurious convergence to a minimum of `fmin` has occurred; `TOLX` is the convergence criterion on $\delta\mathbf{x}$; `STPMX` is the scaled maximum step length allowed in line searches.

```
int nn;                                         Global variables to communicate with fmin.
float *fvec;
void (*nrfuncv)(int n, float v[], float f[]);
#define FREERETURN {free_vector(fvec,1,n);free_vector(xold,1,n);\
    free_vector(p,1,n);free_vector(g,1,n);free_matrix(fjac,1,n,1,n);\
    free_ivector(indx,1,n);return;}
```

```
void newt(float x[], int n, int *check,
    void (*vecfunc)(int, float [], float []))
```

Given an initial guess `x[1..n]` for a root in `n` dimensions, find the root by a globally convergent Newton's method. The vector of functions to be zeroed, called `fvec[1..n]` in the routine below, is returned by the user-supplied routine `vecfunc(n,x,fvec)`. The output quantity `check` is false (0) on a normal return and true (1) if the routine has converged to a local minimum of the function `fmin` defined below. In this case try restarting from a different initial guess.

```
{
    void fdjac(int n, float x[], float fvec[], float **df,
        void (*vecfunc)(int, float [], float []));
    float fmin(float x[]);
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
         float *f, float stpmax, int *check, float (*func)(float []));
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,its,j,*indx;
    float d,den,f,fold,stpmax,sum,temp,test,**fjac,*g,*p,*xold;

    indx=ivector(1,n);
    fjac=matrix(1,n,1,n);
    g=vector(1,n);
    p=vector(1,n);
    xold=vector(1,n);
    fvec=vector(1,n);                          Define global variables.
    nn=n;
    nrfuncv=vecfunc;
    f=fmin(x);                                 fvec is also computed by this call.
    test=0.0;                                  Test for initial guess being a root. Use
    for (i=1;i<=n;i++)                             more stringent test than simply TOLF.
        if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
    if (test < 0.01*TOLF) {
        *check=0;
        FREERETURN
    }
    for (sum=0.0,i=1;i<=n;i++) sum += SQR(x[i]);      Calculate stpmax for line searches.
    stpmax=STPMX*FMAX(sqrt(sum),(float)n);
    for (its=1;its<=MAXITS;its++) {            Start of iteration loop.
        fdjac(n,x,fvec,fjac,vecfunc);
        If analytic Jacobian is available, you can replace the routine fdjac below with your
        own routine.
        for (i=1;i<=n;i++) {                   Compute ∇f for the line search.
            for (sum=0.0,j=1;j<=n;j++) sum += fjac[j][i]*fvec[j];
            g[i]=sum;
        }
        for (i=1;i<=n;i++) xold[i]=x[i];       Store x,
        fold=f;                                and f.
        for (i=1;i<=n;i++) p[i] = -fvec[i];    Right-hand side for linear equations.
        ludcmp(fjac,n,indx,&d);                Solve linear equations by LU decompo-
        lubksb(fjac,n,indx,p);                     sition.
        lnsrch(n,xold,fold,g,p,x,&f,stpmax,check,fmin);
        lnsrch returns new x and f. It also calculates fvec at the new x when it calls fmin.
        test=0.0;                              Test for convergence on function val-
        for (i=1;i<=n;i++)                         ues.
            if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
        if (test < TOLF) {
            *check=0;
            FREERETURN
        }
        if (*check) {                          Check for gradient of f zero, i.e., spuri-
            test=0.0;                              ous convergence.
            den=FMAX(f,0.5*n);
            for (i=1;i<=n;i++) {
                temp=fabs(g[i])*FMAX(fabs(x[i]),1.0)/den;
```

```
                if (temp > test) test=temp;
            }
            *check=(test < TOLMIN ? 1 : 0);
            FREERETURN
        }
        test=0.0;                                Test for convergence on δx.
        for (i=1;i<=n;i++) {
            temp=(fabs(x[i]-xold[i]))/FMAX(fabs(x[i]),1.0);
            if (temp > test) test=temp;
        }
        if (test < TOLX) FREERETURN
    }
    nrerror("MAXITS exceeded in newt");
}
```

```
#include <math.h>
#include "nrutil.h"
#define EPS 1.0e-4                Approximate square root of the machine precision.

void fdjac(int n, float x[], float fvec[], float **df,
    void (*vecfunc)(int, float [], float []))
```

Computes forward-difference approximation to Jacobian. On input, `x[1..n]` is the point at which the Jacobian is to be evaluated, `fvec[1..n]` is the vector of function values at the point, and `vecfunc(n,x,f)` is a user-supplied routine that returns the vector of functions at `x`. On output, `df[1..n][1..n]` is the Jacobian array.

```
{
    int i,j;
    float h,temp,*f;

    f=vector(1,n);
    for (j=1;j<=n;j++) {
        temp=x[j];
        h=EPS*fabs(temp);
        if (h == 0.0) h=EPS;
        x[j]=temp+h;              Trick to reduce finite precision error.
        h=x[j]-temp;
        (*vecfunc)(n,x,f);
        x[j]=temp;
        for (i=1;i<=n;i++) df[i][j]=(f[i]-fvec[i])/h;        Forward difference for-
    }                                                            mula.
    free_vector(f,1,n);
}
```

```
#include "nrutil.h"

extern int nn;
extern float *fvec;
extern void (*nrfuncv)(int n, float v[], float f[]);

float fmin(float x[])
```

Returns $f = \frac{1}{2}\mathbf{F}\cdot\mathbf{F}$ at `x`. The global pointer `*nrfuncv` points to a routine that returns the vector of functions at `x`. It is set to point to a user-supplied routine in the calling program. Global variables also communicate the function values back to the calling program.

```
{
    int i;
    float sum;

    (*nrfuncv)(nn,x,fvec);
    for (sum=0.0,i=1;i<=nn;i++) sum += SQR(fvec[i]);
    return 0.5*sum;
}
```

The routine `newt` assumes that typical values of all components of $\mathbf{x}$ and of $\mathbf{F}$ are of order unity, and it can fail if this assumption is badly violated. You should rescale the variables by their typical values before invoking `newt` if this problem occurs.

Multidimensional Secant Methods: Broyden's Method

Newton's method as implemented above is quite powerful, but it still has several disadvantages. One drawback is that the Jacobian matrix is needed. In many problems analytic derivatives are unavailable. If function evaluation is expensive, then the cost of finite-difference determination of the Jacobian can be prohibitive.

Just as the quasi-Newton methods to be discussed in §10.7 provide cheap approximations for the Hessian matrix in minimization algorithms, there are quasi-Newton methods that provide cheap approximations to the Jacobian for zero finding. These methods are often called *secant methods*, since they reduce to the secant method (§9.2) in one dimension (see, e.g., [1]). The best of these methods still seems to be the first one introduced, *Broyden's method* [2].

Let us denote the approximate Jacobian by $\mathbf{B}$. Then the ith quasi-Newton step $\delta\mathbf{x}_i$ is the solution of

$$\mathbf{B}_i \cdot \delta\mathbf{x}_i = -\mathbf{F}_i \tag{9.7.15}$$

where $\delta\mathbf{x}_i = \mathbf{x}_{i+1} - \mathbf{x}_i$ (cf. equation 9.7.3). The quasi-Newton or secant condition is that $\mathbf{B}_{i+1}$ satisfy

$$\mathbf{B}_{i+1} \cdot \delta\mathbf{x}_i = \delta\mathbf{F}_i \tag{9.7.16}$$

where $\delta\mathbf{F}_i = \mathbf{F}_{i+1} - \mathbf{F}_i$. This is the generalization of the one-dimensional secant approximation to the derivative, $\delta F/\delta x$. However, equation (9.7.16) does not determine $\mathbf{B}_{i+1}$ uniquely in more than one dimension.

Many different auxiliary conditions to pin down $\mathbf{B}_{i+1}$ have been explored, but the best-performing algorithm in practice results from Broyden's formula. This formula is based on the idea of getting $\mathbf{B}_{i+1}$ by making the least change to $\mathbf{B}_i$ consistent with the secant equation (9.7.16). Broyden showed that the resulting formula is

$$\mathbf{B}_{i+1} = \mathbf{B}_i + \frac{(\delta\mathbf{F}_i - \mathbf{B}_i \cdot \delta\mathbf{x}_i) \otimes \delta\mathbf{x}_i}{\delta\mathbf{x}_i \cdot \delta\mathbf{x}_i} \tag{9.7.17}$$

You can easily check that $\mathbf{B}_{i+1}$ satisfies (9.7.16).

Early implementations of Broyden's method used the Sherman-Morrison formula, equation (2.7.2), to invert equation (9.7.17) analytically,

$$\mathbf{B}_{i+1}^{-1} = \mathbf{B}_i^{-1} + \frac{(\delta\mathbf{x}_i - \mathbf{B}_i^{-1} \cdot \delta\mathbf{F}_i) \otimes \delta\mathbf{x}_i \cdot \mathbf{B}_i^{-1}}{\delta\mathbf{x}_i \cdot \mathbf{B}_i^{-1} \cdot \delta\mathbf{F}_i} \tag{9.7.18}$$

Then instead of solving equation (9.7.3) by e.g., LU decomposition, one determined

$$\delta\mathbf{x}_i = -\mathbf{B}_i^{-1} \cdot \mathbf{F}_i \tag{9.7.19}$$

by matrix multiplication in $O(N^2)$ operations. The disadvantage of this method is that it cannot easily be embedded in a globally convergent strategy, for which the gradient of equation (9.7.4) requires $\mathbf{B}$, not $\mathbf{B}^{-1}$,

$$\nabla(\tfrac{1}{2}\mathbf{F} \cdot \mathbf{F}) \simeq \mathbf{B}^T \cdot \mathbf{F} \tag{9.7.20}$$

Accordingly, we implement the update formula in the form (9.7.17).

However, we can still preserve the $O(N^2)$ solution of (9.7.3) by using QR decomposition (§2.10) instead of LU decomposition. The reason is that because of the special form of equation (9.7.17), the QR decomposition of $\mathbf{B}_i$ can be updated into the QR decomposition of $\mathbf{B}_{i+1}$ in $O(N^2)$ operations (§2.10). All we need is an initial approximation $\mathbf{B}_0$ to start the ball rolling. It is often acceptable to start simply with the identity matrix, and then allow $O(N)$ updates to produce a reasonable approximation to the Jacobian. We prefer to spend the first N function evaluations on a finite-difference approximation to initialize $\mathbf{B}$ via a call to `fdjac`.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 69 of 181

Since **B** is not the exact Jacobian, we are not guaranteed that $\delta\mathbf{x}$ is a descent direction for $f = \frac{1}{2}\mathbf{F}\cdot\mathbf{F}$ (cf. equation 9.7.5). Thus the line search algorithm can fail to return a suitable step if **B** wanders far from the true Jacobian. In this case, we reinitialize **B** by another call to `fdjac`.

Like the secant method in one dimension, Broyden's method converges superlinearly once you get close enough to the root. Embedded in a global strategy, it is almost as robust as Newton's method, and often needs far fewer function evaluations to determine a zero. Note that the final value of **B** is *not* always close to the true Jacobian at the root, even when the method converges.

The routine `broydn` given below is very similar to `newt` in organization. The principal differences are the use of QR decomposition instead of LU, and the updating formula instead of directly determining the Jacobian. The remarks at the end of `newt` about scaling the variables apply equally to `broydn`.

```
#include <math.h>
#include "nrutil.h"
#define MAXITS 200
#define EPS 1.0e-7
#define TOLF 1.0e-4
#define TOLX EPS
#define STPMX 100.0
#define TOLMIN 1.0e-6
```

Here `MAXITS` is the maximum number of iterations; `EPS` is a number close to the machine precision; `TOLF` is the convergence criterion on function values; `TOLX` is the convergence criterion on $\delta\mathbf{x}$; `STPMX` is the scaled maximum step length allowed in line searches; `TOLMIN` is used to decide whether spurious convergence to a minimum of `fmin` has occurred.

```
#define FREERETURN {free_vector(fvec,1,n);free_vector(xold,1,n);\
    free_vector(w,1,n);free_vector(t,1,n);free_vector(s,1,n);\
    free_matrix(r,1,n,1,n);free_matrix(qt,1,n,1,n);free_vector(p,1,n);\
    free_vector(g,1,n);free_vector(fvcold,1,n);free_vector(d,1,n);\
    free_vector(c,1,n);return;}

int nn;                                  Global variables to communicate with fmin.
float *fvec;
void (*nrfuncv)(int n, float v[], float f[]);

void broydn(float x[], int n, int *check,
    void (*vecfunc)(int, float [], float []))
```

Given an initial guess `x[1..n]` for a root in `n` dimensions, find the root by Broyden's method embedded in a globally convergent strategy. The vector of functions to be zeroed, called `fvec[1..n]` in the routine below, is returned by the user-supplied routine `vecfunc(n,x,fvec)`. The routine `fdjac` and the function `fmin` from `newt` are used. The output quantity `check` is false (0) on a normal return and true (1) if the routine has converged to a local minimum of the function `fmin` or if Broyden's method can make no further progress. In this case try restarting from a different initial guess.

```
{
    void fdjac(int n, float x[], float fvec[], float **df,
        void (*vecfunc)(int, float [], float []));
    float fmin(float x[]);
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
         float *f, float stpmax, int *check, float (*func)(float []));
    void qrdcmp(float **a, int n, float *c, float *d, int *sing);
    void qrupdt(float **r, float **qt, int n, float u[], float v[]);
    void rsolv(float **a, int n, float d[], float b[]);
    int i,its,j,k,restrt,sing,skip;
    float den,f,fold,stpmax,sum,temp,test,*c,*d,*fvcold;
    float *g,*p,**qt,**r,*s,*t,*w,*xold;

    c=vector(1,n);
    d=vector(1,n);
    fvcold=vector(1,n);
    g=vector(1,n);
    p=vector(1,n);
```

```
qt=matrix(1,n,1,n);
r=matrix(1,n,1,n);
s=vector(1,n);
t=vector(1,n);
w=vector(1,n);
xold=vector(1,n);
fvec=vector(1,n);                    Define global variables.
nn=n;
nrfuncv=vecfunc;
f=fmin(x);                           The vector fvec is also computed by this
test=0.0;                                call.
for (i=1;i<=n;i++)                   Test for initial guess being a root. Use more
    if (fabs(fvec[i]) > test)test=fabs(fvec[i]);      stringent test than sim-
if (test < 0.01*TOLF) {                                ply TOLF.
    *check=0;
    FREERETURN
}
for (sum=0.0,i=1;i<=n;i++) sum += SQR(x[i]);     Calculate stpmax for line searches.
stpmax=STPMX*FMAX(sqrt(sum),(float)n);
restrt=1;                            Ensure initial Jacobian gets computed.
for (its=1;its<=MAXITS;its++) {      Start of iteration loop.
    if (restrt) {
        fdjac(n,x,fvec,r,vecfunc);   Initialize or reinitialize Jacobian in r.
        qrdcmp(r,n,c,d,&sing);       QR decomposition of Jacobian.
        if (sing) nrerror("singular Jacobian in broydn");
        for (i=1;i<=n;i++) {                 Form Q^T explicitly.
            for (j=1;j<=n;j++) qt[i][j]=0.0;
            qt[i][i]=1.0;
        }
        for (k=1;k<n;k++) {
            if (c[k]) {
                for (j=1;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=1;i<=n;i++) {                 Form R explicitly.
            r[i][i]=d[i];
            for (j=1;j<i;j++) r[i][j]=0.0;
        }
    } else {                                 Carry out Broyden update.
        for (i=1;i<=n;i++) s[i]=x[i]-xold[i];    s = δx.
        for (i=1;i<=n;i++) {                     t = R · s.
            for (sum=0.0,j=i;j<=n;j++) sum += r[i][j]*s[j];
            t[i]=sum;
        }
        skip=1;
        for (i=1;i<=n;i++) {                     w = δF − B · s.
            for (sum=0.0,j=1;j<=n;j++) sum += qt[j][i]*t[j];
            w[i]=fvec[i]-fvcold[i]-sum;
            if (fabs(w[i]) >= EPS*(fabs(fvec[i])+fabs(fvcold[i]))) skip=0;
            Don't update with noisy components of w.
            else w[i]=0.0;
        }
        if (!skip) {
            for (i=1;i<=n;i++) {                 t = Q^T · w.
                for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*w[j];
                t[i]=sum;
            }
```

```
                for (den=0.0,i=1;i<=n;i++) den += SQR(s[i]);
                for (i=1;i<=n;i++) s[i] /= den;       Store s/(s · s) in s.
                qrupdt(r,qt,n,t,s);                   Update R and Q^T.
                for (i=1;i<=n;i++) {
                    if (r[i][i] == 0.0) nrerror("r singular in broydn");
                    d[i]=r[i][i];                     Diagonal of R stored in d.
                }
            }
        }
        for (i=1;i<=n;i++) {            Right-hand side for linear equations is −Q^T · F.
            for (sum=0.0,j=1;j<=n;j++) sum += qt[i][j]*fvec[j];
            p[i] = -sum;
        }
        for (i=n;i>=1;i--) {            Compute ∇f ≈ (Q · R)^T · F for the line search.
            for (sum=0.0,j=1;j<=i;j++) sum -= r[j][i]*p[j];
            g[i]=sum;
        }
        for (i=1;i<=n;i++) {            Store x and F.
            xold[i]=x[i];
            fvcold[i]=fvec[i];
        }
        fold=f;                         Store f.
        rsolv(r,n,d,p);                     Solve linear equations.
        lnsrch(n,xold,fold,g,p,x,&f,stpmax,check,fmin);
        lnsrch returns new x and f. It also calculates fvec at the new x when it calls fmin.
        test=0.0;                           Test for convergence on function values.
        for (i=1;i<=n;i++)
            if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
        if (test < TOLF) {
            *check=0;
            FREERETURN
        }
        if (*check) {                       True if line search failed to find a new x.
            if (restrt) FREERETURN          Failure; already tried reinitializing the Jaco-
            else {                              bian.
                test=0.0;                   Check for gradient of f zero, i.e., spurious
                den=FMAX(f,0.5*n);              convergence.
                for (i=1;i<=n;i++) {
                    temp=fabs(g[i])*FMAX(fabs(x[i]),1.0)/den;
                    if (temp > test) test=temp;
                }
                if (test < TOLMIN) FREERETURN
                else restrt=1;              Try reinitializing the Jacobian.
            }
        } else {                            Successful step; will use Broyden update for
            restrt=0;                           next step.
            test=0.0;                       Test for convergence on δx.
            for (i=1;i<=n;i++) {
                temp=(fabs(x[i]-xold[i]))/FMAX(fabs(x[i]),1.0);
                if (temp > test) test=temp;
            }
            if (test < TOLX) FREERETURN
        }
    }
    nrerror("MAXITS exceeded in broydn");
    FREERETURN
}
```

More Advanced Implementations

One of the principal ways that the methods described so far can fail is if **J** (in Newton's method) or **B** in (Broyden's method) becomes singular or nearly singular, so that $\delta\mathbf{x}$ cannot be determined. If you are lucky, this situation will not occur very often in practice. Methods developed so far to deal with this problem involve monitoring the condition number of **J** and perturbing **J** if singularity or near singularity is detected. This is most easily implemented if the QR decomposition is used instead of LU in Newton's method (see [1] for details). Our personal experience is that, while such an algorithm can solve problems where **J** is exactly singular and the standard Newton's method fails, it is occasionally less robust on other problems where LU decomposition succeeds. Clearly implementation details involving roundoff, underflow, etc., are important here and the last word is yet to be written.

Our global strategies both for minimization and zero finding have been based on line searches. Other global algorithms, such as the *hook step* and *dogleg step* methods, are based instead on the *model-trust region* approach, which is related to the Levenberg-Marquardt algorithm for nonlinear least-squares (§15.5). While somewhat more complicated than line searches, these methods have a reputation for robustness even when starting far from the desired zero or minimum [1].

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Broyden, C.G. 1965, *Mathematics of Computation*, vol. 19, pp. 577–593. [2]

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 73 of 181

Chapter 10. Minimization or Maximization of Functions

10.0 Introduction

In a nutshell: You are given a single function f that depends on one or more independent variables. You want to find the value of those variables where f takes on a maximum or a minimum value. You can then calculate what value of f is achieved at the maximum or minimum. The tasks of maximization and minimization are trivially related to each other, since one person's function f could just as well be another's $-f$. The computational desiderata are the usual ones: Do it quickly, cheaply, and in small memory. Often the computational effort is dominated by the cost of evaluating f (and also perhaps its partial derivatives with respect to all variables, if the chosen algorithm requires them). In such cases the desiderata are sometimes replaced by the simple surrogate: Evaluate f as few times as possible.

An extremum (maximum or minimum point) can be either *global* (truly the highest or lowest function value) or *local* (the highest or lowest in a finite neighborhood and not on the boundary of that neighborhood). (See Figure 10.0.1.) Finding a global extremum is, in general, a very difficult problem. Two standard heuristics are widely used: (i) find local extrema starting from widely varying starting values of the independent variables (perhaps chosen quasi-randomly, as in §7.7), and then pick the most extreme of these (if they are not all the same); or (ii) perturb a local extremum by taking a finite amplitude step away from it, and then see if your routine returns you to a better point, or "always" to the same one. Relatively recently, so-called "simulated annealing methods" (§10.9) have demonstrated important successes on a variety of global extremization problems.

Our chapter title could just as well be *optimization*, which is the usual name for this very large field of numerical research. The importance ascribed to the various tasks in this field depends strongly on the particular interests of whom you talk to. Economists, and some engineers, are particularly concerned with *constrained optimization*, where there are *a priori* limitations on the allowed values of independent variables. For example, the production of wheat in the U.S. must be a nonnegative number. One particularly well-developed area of constrained optimization is *linear programming*, where both the function to be optimized and the constraints happen to be linear functions of the independent variables. Section 10.8, which is otherwise somewhat disconnected from the rest of the material that we have chosen to include in this chapter, implements the so-called "simplex algorithm" for linear programming problems.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 74 of 181

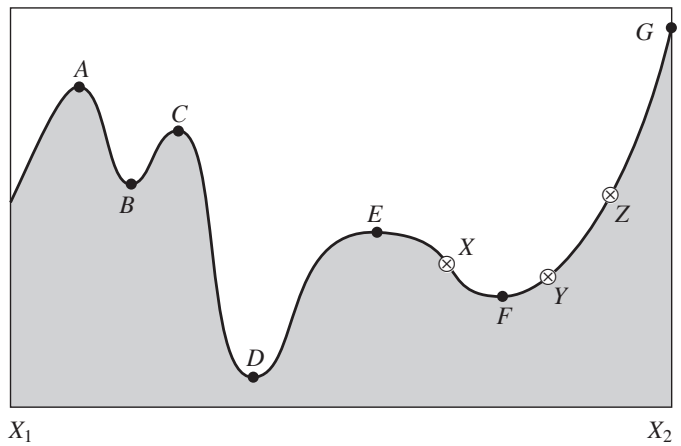

Figure 10.0.1. Extrema of a function in an interval. Points A, C, and E are local, but not global maxima. Points B and F are local, but not global minima. The global maximum occurs at G, which is on the boundary of the interval so that the derivative of the function need not vanish there. The global minimum is at D. At point E, derivatives higher than the first vanish, a situation which can cause difficulty for some algorithms. The points X, Y, and Z are said to "bracket" the minimum F, since Y is less than both X and Z.

One other section, §10.9, also lies outside of our main thrust, but for a different reason: so-called "annealing methods" are relatively new, so we do not yet know where they will ultimately fit into the scheme of things. However, these methods have solved some problems previously thought to be practically insoluble; they address directly the problem of finding global extrema in the presence of large numbers of undesired local extrema.

The other sections in this chapter constitute a selection of the best established algorithms in unconstrained minimization. (For definiteness, we will henceforth regard the optimization problem as that of minimization.) These sections are connected, with later ones depending on earlier ones. If you are just looking for the one "perfect" algorithm to solve your particular application, you may feel that we are telling you more than you want to know. Unfortunately, there is *no* perfect optimization algorithm. This is a case where we strongly urge you to try more than one method in comparative fashion. Your initial choice of method can be based on the following considerations:

- You must choose between methods that need only evaluations of the function to be minimized and methods that also require evaluations of the derivative of that function. In the multidimensional case, this derivative is the gradient, a vector quantity. Algorithms using the derivative are somewhat more powerful than those using only the function, but not always enough so as to compensate for the additional calculations of derivatives. We can easily construct examples favoring one approach or favoring the other. However, if you *can* compute derivatives, be prepared to try using them.
- For one-dimensional minimization (minimize a function of one variable) *without* calculation of the derivative, bracket the minimum as described in §10.1, and then use *Brent's method* as described in §10.2. If your function has a discontinuous second (or lower) derivative, then the parabolic

interpolations of Brent's method are of no advantage, and you might wish to use the simplest form of *golden section search*, as described in §10.1.

- For one-dimensional minimization *with* calculation of the derivative, §10.3 supplies a variant of Brent's method which makes limited use of the first derivative information. We shy away from the alternative of using derivative information to construct high-order interpolating polynomials. In our experience the improvement in convergence very near a smooth, analytic minimum does not make up for the tendency of polynomials sometimes to give wildly wrong interpolations at early stages, especially for functions that may have sharp, "exponential" features.

We now turn to the multidimensional case, both with and without computation of first derivatives.

- You must choose between methods that require storage of order N^2 and those that require only of order N, where N is the number of dimensions. For moderate values of N and reasonable memory sizes this is not a serious constraint. There will be, however, the occasional application where storage may be critical.
- We give in §10.4 a sometimes overlooked *downhill simplex method* due to Nelder and Mead. (This use of the word "simplex" is not to be confused with the simplex method of linear programming.) This method just crawls downhill in a straightforward fashion that makes almost no special assumptions about your function. This can be extremely slow, but it can also, in some cases, be extremely robust. Not to be overlooked is the fact that the code is concise and completely self-contained: a general N-dimensional minimization program in under 100 program lines! This method is most useful when the minimization calculation is only an incidental part of your overall problem. The storage requirement is of order N^2, and derivative calculations are not required.
- Section 10.5 deals with *direction-set methods*, of which *Powell's method* is the prototype. These are the methods of choice when you cannot easily calculate derivatives, and are not necessarily to be sneered at even if you can. Although derivatives are not needed, the method does require a one-dimensional minimization sub-algorithm such as Brent's method (see above). Storage is of order N^2.

There are two major families of algorithms for multidimensional minimization *with* calculation of first derivatives. Both families require a one-dimensional minimization sub-algorithm, which can itself either use, or not use, the derivative information, as you see fit (depending on the relative effort of computing the function and of its gradient vector). We do not think that either family dominates the other in all applications; you should think of them as available alternatives:

- The first family goes under the name *conjugate gradient methods*, as typified by the *Fletcher-Reeves algorithm* and the closely related and probably superior *Polak-Ribiere algorithm*. Conjugate gradient methods require only of order a few times N storage, require derivative calculations and

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 76 of 181

one-dimensional sub-minimization. Turn to §10.6 for detailed discussion and implementation.

- The second family goes under the names *quasi-Newton* or *variable metric* methods, as typified by the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to just as *Fletcher-Powell*) or the closely related *Broyden-Fletcher-Goldfarb-Shanno (BFGS)* algorithm. These methods require of order N^2 storage, require derivative calculations and one-dimensional sub-minimization. Details are in §10.7.

You are now ready to proceed with scaling the peaks (and/or plumbing the depths) of practical optimization.

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall).

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press).

Gill, P.E., Murray, W., and Wright, M.H. 1981, *Practical Optimization* (New York: Academic Press).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 17.

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall).

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

10.1 Golden Section Search in One Dimension

Recall how the bisection method finds roots of functions in one dimension (§9.1): The root is supposed to have been bracketed in an interval (a, b). One then evaluates the function at an intermediate point x and obtains a new, smaller bracketing interval, either (a, x) or (x, b). The process continues until the bracketing interval is acceptably small. It is optimal to choose x to be the midpoint of (a, b) so that the decrease in the interval length is maximized when the function is as uncooperative as it can be, i.e., when the luck of the draw forces you to take the bigger bisected segment.

There is a precise, though slightly subtle, translation of these considerations to the minimization problem: What does it mean to *bracket* a minimum? A root of a function is known to be bracketed by a pair of points, a and b, when the function has opposite sign at those two points. A minimum, by contrast, is known to be bracketed only when there is a *triplet* of points, $a < b < c$ (or $c < b < a$), such that $f(b)$ is less than both $f(a)$ and $f(c)$. In this case we know that the function (if it is nonsingular) has a minimum in the interval (a, c).

The analog of bisection is to choose a new point x, either between a and b or between b and c. Suppose, to be specific, that we make the latter choice. Then we evaluate $f(x)$. If $f(b) < f(x)$, then the new bracketing triplet of points is (a, b, x);

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 77 of 181

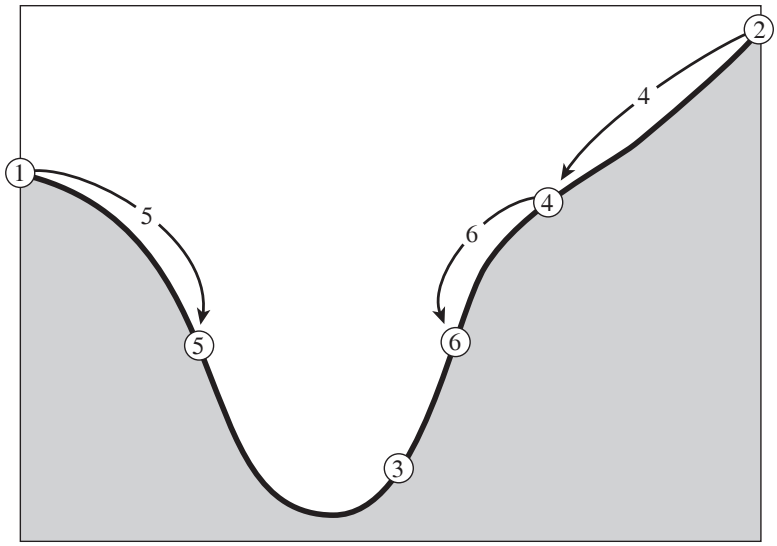

Figure 10.1.1. Successive bracketing of a minimum. The minimum is originally bracketed by points 1,3,2. The function is evaluated at 4, which replaces 2; then at 5, which replaces 1; then at 6, which replaces 4. The rule at each stage is to keep a center point that is lower than the two outside points. After the steps shown, the minimum is bracketed by points 5,3,6.

contrariwise, if $f(b) > f(x)$, then the new bracketing triplet is (b, x, c). In all cases the middle point of the new triplet is the abscissa whose ordinate is the best minimum achieved so far; see Figure 10.1.1. We continue the process of bracketing until the distance between the two outer points of the triplet is tolerably small.

How small is "tolerably" small? For a minimum located at a value b, you might naively think that you will be able to bracket it in as small a range as $(1-\epsilon)b < b < (1+\epsilon)b$, where ϵ is your computer's floating-point precision, a number like 3×10^{-8} (for `float`) or 10^{-15} (for `double`). Not so! In general, the shape of your function $f(x)$ near b will be given by Taylor's theorem

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x-b)^2 \tag{10.1.1}$$

The second term will be negligible compared to the first (that is, will be a factor ϵ smaller and will act just like zero when added to it) whenever

$$|x-b| < \sqrt{\epsilon}|b| \sqrt{\frac{2\,|f(b)|}{b^2 f''(b)}} \tag{10.1.2}$$

The reason for writing the right-hand side in this way is that, for most functions, the final square root is a number of order unity. Therefore, as a rule of thumb, it is hopeless to ask for a bracketing interval of width less than $\sqrt{\epsilon}$ times its central value, a fractional width of only about 10^{-4} (single precision) or 3×10^{-8} (double precision). Knowing this inescapable fact will save you a lot of useless bisections!

The minimum-finding routines of this chapter will often call for a user-supplied argument `tol`, and return with an abscissa whose fractional precision is about $\pm$`tol` (bracketing interval of fractional size about $2\times$`tol`). Unless you have a better

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 78 of 181

estimate for the right-hand side of equation (10.1.2), you should set `tol` equal to (not much less than) the square root of your machine's floating-point precision, since smaller values will gain you nothing.

It remains to decide on a strategy for choosing the new point x, given (a, b, c). Suppose that b is a fraction w of the way between a and c, i.e.

$$\frac{b-a}{c-a} = w \qquad \frac{c-b}{c-a} = 1 - w \tag{10.1.3}$$

Also suppose that our next trial point x is an additional fraction z beyond b,

$$\frac{x-b}{c-a} = z \tag{10.1.4}$$

Then the next bracketing segment will either be of length $w + z$ relative to the current one, or else of length $1 - w$. If we want to minimize the worst case possibility, then we will choose z to make these equal, namely

$$z = 1 - 2w \tag{10.1.5}$$

We see at once that the new point is the symmetric point to b in the original interval, namely with $|b - a|$ equal to $|x - c|$. This implies that the point x lies in the larger of the two segments (z is positive only if $w < 1/2$).

But where in the larger segment? Where did the value of w itself come from? Presumably from the previous stage of applying our same strategy. Therefore, if z is chosen to be optimal, then so was w before it. This *scale similarity* implies that x should be the same fraction of the way from b to c (if that is the bigger segment) as was b from a to c, in other words,

$$\frac{z}{1-w} = w \tag{10.1.6}$$

Equations (10.1.5) and (10.1.6) give the quadratic equation

$$w^2 - 3w + 1 = 0 \qquad \text{yielding} \qquad w = \frac{3-\sqrt{5}}{2} \approx 0.38197 \tag{10.1.7}$$

In other words, the optimal bracketing interval (a, b, c) has its middle point b a fractional distance 0.38197 from one end (say, a), and 0.61803 from the other end (say, b). These fractions are those of the so-called *golden mean* or *golden section*, whose supposedly aesthetic properties hark back to the ancient Pythagoreans. This optimal method of function minimization, the analog of the bisection method for finding zeros, is thus called the *golden section search*, summarized as follows:

Given, at each stage, a bracketing triplet of points, the next point to be tried is that which is a fraction 0.38197 into the larger of the two intervals (measuring from the central point of the triplet). If you start out with a bracketing triplet whose segments are not in the golden ratios, the procedure of choosing successive points at the golden mean point of the larger segment will quickly converge you to the proper, self-replicating ratios.

The golden section search guarantees that each new function evaluation will (after self-replicating ratios have been achieved) bracket the minimum to an interval

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 79 of 181

just 0.61803 times the size of the preceding interval. This is comparable to, but not quite as good as, the 0.50000 that holds when finding roots by bisection. Note that the convergence is *linear* (in the language of Chapter 9), meaning that successive significant figures are won linearly with additional function evaluations. In the next section we will give a superlinear method, where the rate at which successive significant figures are liberated increases with each successive function evaluation.

Routine for Initially Bracketing a Minimum

The preceding discussion has assumed that you are able to bracket the minimum in the first place. We consider this initial bracketing to be an essential part of any one-dimensional minimization. There are some one-dimensional algorithms that do not require a rigorous initial bracketing. However, we would *never* trade the secure feeling of *knowing* that a minimum is "in there somewhere" for the dubious reduction of function evaluations that these nonbracketing routines may promise. Please bracket your minima (or, for that matter, your zeros) before isolating them!

There is not much theory as to how to do this bracketing. Obviously you want to step downhill. But how far? We like to take larger and larger steps, starting with some (wild?) initial guess and then increasing the stepsize at each step either by a constant factor, or else by the result of a parabolic extrapolation of the preceding points that is designed to take us to the extrapolated turning point. It doesn't much matter if the steps get big. After all, we are stepping downhill, so we already have the left and middle points of the bracketing triplet. We just need to take a big enough step to stop the downhill trend and get a high third point.

Our standard routine is this:

```
#include <math.h>
#include "nrutil.h"
#define GOLD 1.618034
#define GLIMIT 100.0
#define TINY 1.0e-20
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);
```

Here GOLD is the default ratio by which successive intervals are magnified; GLIMIT is the maximum magnification allowed for a parabolic-fit step.

```
void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb, float *fc,
    float (*func)(float))
```

Given a function func, and given distinct initial points ax and bx, this routine searches in the downhill direction (defined by the function as evaluated at the initial points) and returns new points ax, bx, cx that bracket a minimum of the function. Also returned are the function values at the three points, fa, fb, and fc.

```
{
    float ulim,u,r,q,fu,dum;

    *fa=(*func)(*ax);
    *fb=(*func)(*bx);
    if (*fb > *fa) {                        Switch roles of a and b so that we can go
        SHFT(dum,*ax,*bx,dum)                   downhill in the direction from a to b.
        SHFT(dum,*fb,*fa,dum)
    }
    *cx=(*bx)+GOLD*(*bx-*ax);               First guess for c.
    *fc=(*func)(*cx);
    while (*fb > *fc) {                     Keep returning here until we bracket.
        r=(*bx-*ax)*(*fb-*fc);              Compute u by parabolic extrapolation from
        q=(*bx-*cx)*(*fb-*fa);                  a, b, c. TINY is used to prevent any pos-
        u=(*bx)-((*bx-*cx)*q-(*bx-*ax)*r)/          sible division by zero.
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 80 of 181

```
            (2.0*SIGN(FMAX(fabs(q-r),TINY),q-r));
        ulim=(*bx)+GLIMIT*(*cx-*bx);
        We won't go farther than this. Test various possibilities:
        if ((*bx-u)*(u-*cx) > 0.0) {        Parabolic u is between b and c: try it.
            fu=(*func)(u);
            if (fu < *fc) {                 Got a minimum between b and c.
                *ax=(*bx);
                *bx=u;
                *fa=(*fb);
                *fb=fu;
                return;
            } else if (fu > *fb) {          Got a minimum between between a and u.
                *cx=u;
                *fc=fu;
                return;
            }
            u=(*cx)+GOLD*(*cx-*bx);         Parabolic fit was no use. Use default mag-
            fu=(*func)(u);                      nification.
        } else if ((*cx-u)*(u-ulim) > 0.0) {        Parabolic fit is between c and its
            fu=(*func)(u);                              allowed limit.
            if (fu < *fc) {
                SHFT(*bx,*cx,u,*cx+GOLD*(*cx-*bx))
                SHFT(*fb,*fc,fu,(*func)(u))
            }
        } else if ((u-ulim)*(ulim-*cx) >= 0.0) {    Limit parabolic u to maximum
            u=ulim;                                     allowed value.
            fu=(*func)(u);
        } else {                            Reject parabolic u, use default magnifica-
            u=(*cx)+GOLD*(*cx-*bx);             tion.
            fu=(*func)(u);
        }
        SHFT(*ax,*bx,*cx,u)                 Eliminate oldest point and continue.
        SHFT(*fa,*fb,*fc,fu)
    }
}
```

(Because of the housekeeping involved in moving around three or four points and their function values, the above program ends up looking deceptively formidable. That is true of several other programs in this chapter as well. The underlying ideas, however, are quite simple.)

Routine for Golden Section Search

```
#include <math.h>
#define R 0.61803399                        The golden ratios.
#define C (1.0-R)
#define SHFT2(a,b,c) (a)=(b);(b)=(c);
#define SHFT3(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

float golden(float ax, float bx, float cx, float (*f)(float), float tol,
    float *xmin)
Given a function f, and given a bracketing triplet of abscissas ax, bx, cx (such that bx is
between ax and cx, and f(bx) is less than both f(ax) and f(cx)), this routine performs a
golden section search for the minimum, isolating it to a fractional precision of about tol. The
abscissa of the minimum is returned as xmin, and the minimum function value is returned as
golden, the returned function value.
{
    float f1,f2,x0,x1,x2,x3;
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 81 of 181

```
    x0=ax;                                  At any given time we will keep track of four
    x3=cx;                                      points, x0,x1,x2,x3.
    if (fabs(cx-bx) > fabs(bx-ax)) {        Make x0 to x1 the smaller segment,
        x1=bx;
        x2=bx+C*(cx-bx);                    and fill in the new point to be tried.
    } else {
        x2=bx;
        x1=bx-C*(bx-ax);
    }
    f1=(*f)(x1);                            The initial function evaluations. Note that
    f2=(*f)(x2);                                we never need to evaluate the function
    while (fabs(x3-x0) > tol*(fabs(x1)+fabs(x2))) {      at the original endpoints.
        if (f2 < f1) {                      One possible outcome,
            SHFT3(x0,x1,x2,R*x1+C*x3)       its housekeeping,
            SHFT2(f1,f2,(*f)(x2))           and a new function evaluation.
        } else {                            The other outcome,
            SHFT3(x3,x2,x1,R*x2+C*x0)
            SHFT2(f2,f1,(*f)(x1))           and its new function evaluation.
        }
    }                                       Back to see if we are done.
    if (f1 < f2) {                          We are done. Output the best of the two
        *xmin=x1;                               current values.
        return f1;
    } else {
        *xmin=x2;
        return f2;
    }
}
```

10.2 Parabolic Interpolation and Brent's Method in One Dimension

We already tipped our hand about the desirability of parabolic interpolation in the previous section's `mnbrak` routine, but it is now time to be more explicit. A golden section search is designed to handle, in effect, the worst possible case of function minimization, with the uncooperative minimum hunted down and cornered like a scared rabbit. But why assume the worst? If the function is nicely parabolic near to the minimum — surely the generic case for sufficiently smooth functions — then the parabola fitted through any three points ought to take us in a single leap to the minimum, or at least very near to it (see Figure 10.2.1). Since we want to find an abscissa rather than an ordinate, the procedure is technically called *inverse parabolic interpolation*.

The formula for the abscissa x that is the minimum of a parabola through three points $f(a)$, $f(b)$, and $f(c)$ is

$$x = b - \frac{1}{2}\,\frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]} \tag{10.2.1}$$

as you can easily derive. This formula fails only if the three points are collinear, in which case the denominator is zero (minimum of the parabola is infinitely far

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 82 of 181

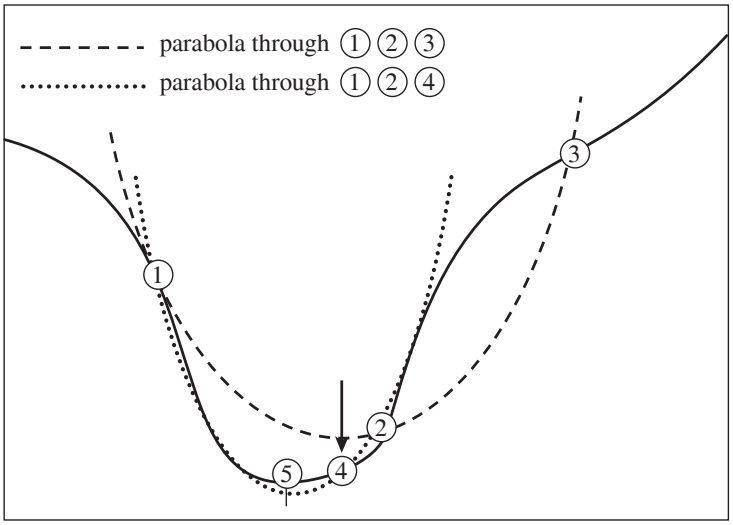

Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn through points 1,4,2. The minimum of this parabola is at 5, which is close to the minimum of the function.

away). Note, however, that (10.2.1) is as happy jumping to a parabolic maximum as to a minimum. No minimization scheme that depends solely on (10.2.1) is likely to succeed in practice.

The exacting task is to invent a scheme that relies on a sure-but-slow technique, like golden section search, when the function is not cooperative, but that switches over to (10.2.1) when the function allows. The task is nontrivial for several reasons, including these: (i) The housekeeping needed to avoid unnecessary function evaluations in switching between the two methods can be complicated. (ii) Careful attention must be given to the "endgame," where the function is being evaluated very near to the roundoff limit of equation (10.1.2). (iii) The scheme for detecting a cooperative versus noncooperative function must be very robust.

Brent's method [1] is up to the task in all particulars. At any particular stage, it is keeping track of six function points (not necessarily all distinct), a, b, u, v, w and x, defined as follows: the minimum is bracketed between a and b; x is the point with the very least function value found so far (or the most recent one in case of a tie); w is the point with the second least function value; v is the previous value of w; u is the point at which the function was evaluated most recently. Also appearing in the algorithm is the point x_m, the midpoint between a and b; however, the function is not evaluated there.

You can read the code below to understand the method's logical organization. Mention of a few general principles here may, however, be helpful: Parabolic interpolation is attempted, fitting through the points x, v, and w. To be acceptable, the parabolic step must (i) fall within the bounding interval (a, b), and (ii) imply a movement from the best current value x that is *less* than half the movement of the *step before last*. This second criterion insures that the parabolic steps are actually converging to something, rather than, say, bouncing around in some nonconvergent limit cycle. In the worst possible case, where the parabolic steps are acceptable but

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 83 of 181

useless, the method will approximately alternate between parabolic steps and golden sections, converging in due course by virtue of the latter. The reason for comparing to the step *before* last seems essentially heuristic: Experience shows that it is better not to "punish" the algorithm for a single bad step if it can make it up on the next one.

Another principle exemplified in the code is never to evaluate the function less than a distance `tol` from a point already evaluated (or from a known bracketing point). The reason is that, as we saw in equation (10.1.2), there is simply no information content in doing so: the function will differ from the value already evaluated only by an amount of order the roundoff error. Therefore in the code below you will find several tests and modifications of a potential new point, imposing this restriction. This restriction also interacts subtly with the test for "doneness," which the method takes into account.

A typical ending configuration for Brent's method is that a and b are $2 \times x \times \texttt{tol}$ apart, with x (the best abscissa) at the midpoint of a and b, and therefore fractionally accurate to $\pm\texttt{tol}$.

Indulge us a final reminder that `tol` should generally be no smaller than the square root of your machine's floating-point precision.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100
#define CGOLD 0.3819660
#define ZEPS 1.0e-10
```

Here `ITMAX` is the maximum allowed number of iterations; `CGOLD` is the golden ratio; `ZEPS` is a small number that protects against trying to achieve fractional accuracy for a minimum that happens to be exactly zero.

```
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

float brent(float ax, float bx, float cx, float (*f)(float), float tol,
    float *xmin)
```

Given a function `f`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` (such that `bx` is between `ax` and `cx`, and `f(bx)` is less than both `f(ax)` and `f(cx)`), this routine isolates the minimum to a fractional precision of about `tol` using Brent's method. The abscissa of the minimum is returned as `xmin`, and the minimum function value is returned as `brent`, the returned function value.

```
{
    int iter;
    float a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
    float e=0.0;                                    This will be the distance moved on
                                                        the step before last.
    a=(ax < cx ? ax : cx);                          a and b must be in ascending order,
    b=(ax > cx ? ax : cx);                              but input abscissas need not be.
    x=w=v=bx;                                       Initializations...
    fw=fv=fx=(*f)(x);
    for (iter=1;iter<=ITMAX;iter++) {               Main program loop.
        xm=0.5*(a+b);
        tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
        if (fabs(x-xm) <= (tol2-0.5*(b-a))) {       Test for done here.
            *xmin=x;
            return fx;
        }
        if (fabs(e) > tol1) {                       Construct a trial parabolic fit.
            r=(x-w)*(fx-fv);
            q=(x-v)*(fx-fw);
            p=(x-v)*q-(x-w)*r;
            q=2.0*(q-r);
            if (q > 0.0) p = -p;
            q=fabs(q);
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 84 of 181

```
            etemp=e;
            e=d;
            if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))
                d=CGOLD*(e=(x >= xm ? a-x : b-x));
            The above conditions determine the acceptability of the parabolic fit. Here we
            take the golden section step into the larger of the two segments.
            else {
                d=p/q;                              Take the parabolic step.
                u=x+d;
                if (u-a < tol2 || b-u < tol2)
                    d=SIGN(tol1,xm-x);
            }
        } else {
            d=CGOLD*(e=(x >= xm ? a-x : b-x));
        }
        u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
        fu=(*f)(u);
        This is the one function evaluation per iteration.
        if (fu <= fx) {                             Now decide what to do with our func-
            if (u >= x) a=x; else b=x;                  tion evaluation.
            SHFT(v,w,x,u)                           Housekeeping follows:
            SHFT(fv,fw,fx,fu)
        } else {
            if (u < x) a=u; else b=u;
            if (fu <= fw || w == x) {
                v=w;
                w=u;
                fv=fw;
                fw=fu;
            } else if (fu <= fv || v == x || v == w) {
                v=u;
                fv=fu;
            }
        }                                           Done with housekeeping. Back for
    }                                                   another iteration.
    nrerror("Too many iterations in brent");
    *xmin=x;                                        Never get here.
    return fx;
}
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 5. [1]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §8.2.

10.3 One-Dimensional Search with First Derivatives

Here we want to accomplish precisely the same goal as in the previous section, namely to isolate a functional minimum that is bracketed by the triplet of abscissas (a, b, c), but utilizing an additional capability to compute the function's first derivative as well as its value.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 85 of 181

In principle, we might simply search for a zero of the derivative, ignoring the function value information, using a root finder like `rtflsp` or `zbrent` (§§9.2–9.3). It doesn't take long to reject *that* idea: How do we distinguish maxima from minima? Where do we go from initial conditions where the derivatives on one or both of the outer bracketing points indicate that "downhill" is in the direction *out* of the bracketed interval?

We don't want to give up our strategy of maintaining a rigorous bracket on the minimum at all times. The only way to keep such a bracket is to update it using function (not derivative) information, with the central point in the bracketing triplet always that with the lowest function value. Therefore the role of the derivatives can only be to help us choose new trial points within the bracket.

One school of thought is to "use everything you've got": Compute a polynomial of relatively high order (cubic or above) that agrees with some number of previous function and derivative evaluations. For example, there is a unique cubic that agrees with function and derivative at two points, and one can jump to the interpolated minimum of that cubic (if there is a minimum within the bracket). Suggested by Davidon and others, formulas for this tactic are given in [1].

We like to be more conservative than this. Once superlinear convergence sets in, it hardly matters whether its order is moderately lower or higher. In practical problems that we have met, most function evaluations are spent in getting globally close enough to the minimum for superlinear convergence to commence. So we are more worried about all the funny "stiff" things that high-order polynomials can do (cf. Figure 3.0.1b), and about their sensitivities to roundoff error.

This leads us to use derivative information only as follows: The sign of the derivative at the central point of the bracketing triplet (a, b, c) indicates uniquely whether the next test point should be taken in the interval (a, b) or in the interval (b, c). The value of this derivative and of the derivative at the second-best-so-far point are extrapolated to zero by the secant method (inverse linear interpolation), which by itself is superlinear of order 1.618. (The golden mean again: see [1], p. 57.) We impose the same sort of restrictions on this new trial point as in Brent's method. If the trial point must be rejected, we *bisect* the interval under scrutiny.

Yes, we are fuddy-duddies when it comes to making flamboyant use of derivative information in one-dimensional minimization. But we have met too many functions whose computed "derivatives" *don't* integrate up to the function value and *don't* accurately point the way to the minimum, usually because of roundoff errors, sometimes because of truncation error in the method of derivative evaluation.

You will see that the following routine is closely modeled on `brent` in the previous section.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100
#define ZEPS 1.0e-10
#define MOV3(a,b,c, d,e,f) (a)=(d);(b)=(e);(c)=(f);

float dbrent(float ax, float bx, float cx, float (*f)(float),
    float (*df)(float), float tol, float *xmin)
```

Given a function `f` and its derivative function `df`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` [such that `bx` is between `ax` and `cx`, and `f(bx)` is less than both `f(ax)` and `f(cx)`], this routine isolates the minimum to a fractional precision of about `tol` using a modification of Brent's method that uses derivatives. The abscissa of the minimum is returned as `xmin`, and

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 86 of 181

the minimum function value is returned as dbrent, the returned function value.

```
{
    int iter,ok1,ok2;                          Will be used as flags for whether pro-
    float a,b,d,d1,d2,du,dv,dw,dx,e=0.0;           posed steps are acceptable or not.
    float fu,fv,fw,fx,olde,tol1,tol2,u,u1,u2,v,w,x,xm;

    Comments following will point out only differences from the routine brent. Read that
    routine first.
    a=(ax < cx ? ax : cx);
    b=(ax > cx ? ax : cx);
    x=w=v=bx;
    fw=fv=fx=(*f)(x);
    dw=dv=dx=(*df)(x);                            All our housekeeping chores are dou-
    for (iter=1;iter<=ITMAX;iter++) {                bled by the necessity of moving
        xm=0.5*(a+b);                                derivative values around as well
        tol1=tol*fabs(x)+ZEPS;                       as function values.
        tol2=2.0*tol1;
        if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
            *xmin=x;
            return fx;
        }
        if (fabs(e) > tol1) {
            d1=2.0*(b-a);                         Initialize these d's to an out-of-bracket
            d2=d1;                                   value.
            if (dw != dx) d1=(w-x)*dx/(dx-dw);    Secant method with one point.
            if (dv != dx) d2=(v-x)*dx/(dx-dv);    And the other.
            Which of these two estimates of d shall we take? We will insist that they be within
            the bracket, and on the side pointed to by the derivative at x:
            u1=x+d1;
            u2=x+d2;
            ok1 = (a-u1)*(u1-b) > 0.0 && dx*d1 <= 0.0;
            ok2 = (a-u2)*(u2-b) > 0.0 && dx*d2 <= 0.0;
            olde=e;                               Movement on the step before last.
            e=d;
            if (ok1 || ok2) {                     Take only an acceptable d, and if
                if (ok1 && ok2)                          both are acceptable, then take
                    d=(fabs(d1) < fabs(d2) ? d1 : d2);   the smallest one.
                else if (ok1)
                    d=d1;
                else
                    d=d2;
                if (fabs(d) <= fabs(0.5*olde)) {
                    u=x+d;
                    if (u-a < tol2 || b-u < tol2)
                        d=SIGN(tol1,xm-x);
                } else {                          Bisect, not golden section.
                    d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
                    Decide which segment by the sign of the derivative.
                }
            } else {
                d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
            }
        } else {
            d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
        }
        if (fabs(d) >= tol1) {
            u=x+d;
            fu=(*f)(u);
        } else {
            u=x+SIGN(tol1,d);
            fu=(*f)(u);
            if (fu > fx) {                        If the minimum step in the downhill
                *xmin=x;                             direction takes us uphill, then
                return fx;                           we are done.
```

```
            }
        }
        du=(*df)(u);                                Now all the housekeeping, sigh.
        if (fu <= fx) {
            if (u >= x) a=x; else b=x;
            MOV3(v,fv,dv, w,fw,dw)
            MOV3(w,fw,dw, x,fx,dx)
            MOV3(x,fx,dx, u,fu,du)
        } else {
            if (u < x) a=u; else b=u;
            if (fu <= fw || w == x) {
                MOV3(v,fv,dv, w,fw,dw)
                MOV3(w,fw,dw, u,fu,du)
            } else if (fu < fv || v == x || v == w) {
                MOV3(v,fv,dv, u,fu,du)
            }
        }
    }
    nrerror("Too many iterations in routine dbrent");
    return 0.0;                                     Never get here.
}
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 55; 454–458. [1]

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), p. 78.

10.4 Downhill Simplex Method in Multidimensions

With this section we begin consideration of multidimensional minimization, that is, finding the minimum of a function of more than one independent variable. This section stands apart from those which follow, however: All of the algorithms after this section will make explicit use of a one-dimensional minimization algorithm as a part of their computational strategy. This section implements an entirely self-contained strategy, in which one-dimensional minimization does not figure.

The *downhill simplex method* is due to Nelder and Mead [1]. The method requires only function evaluations, not derivatives. It is not very efficient in terms of the number of function evaluations that it requires. Powell's method (§10.5) is almost surely faster in all likely applications. However, the downhill simplex method may frequently be the *best* method to use if the figure of merit is "get something working quickly" for a problem whose computational burden is small.

The method has a geometrical naturalness about it which makes it delightful to describe or work through:

A *simplex* is the geometrical figure consisting, in N dimensions, of $N + 1$ points (or vertices) and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron, not necessarily the regular tetrahedron. (The *simplex method* of linear programming,

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 88 of 181

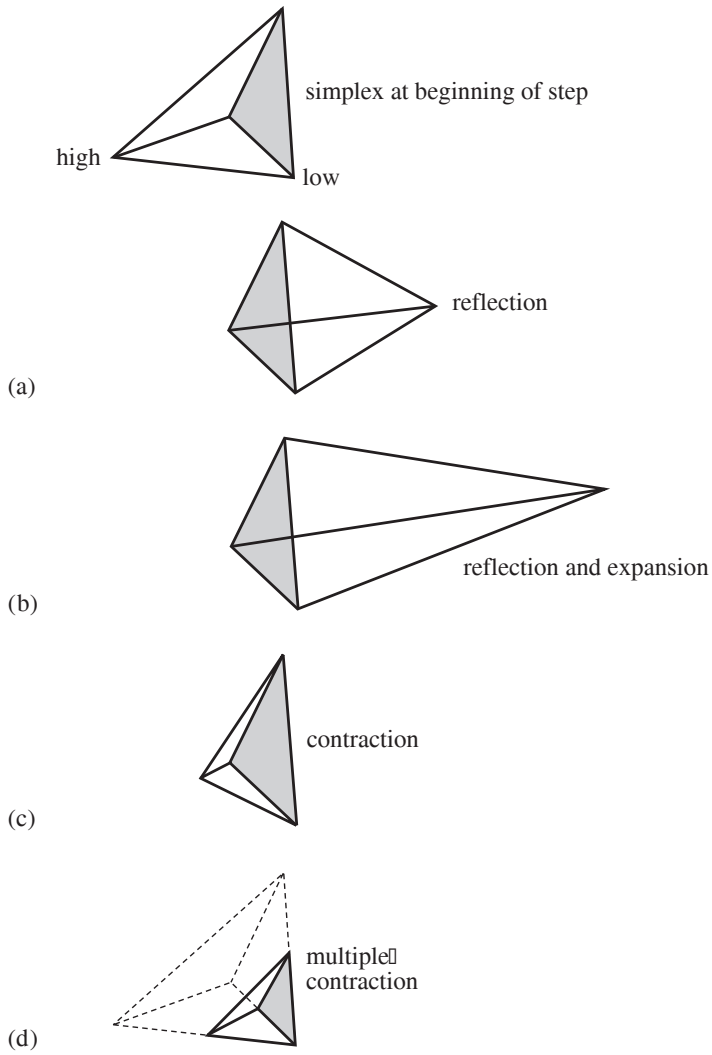

Figure 10.4.1. Possible outcomes for a step in the downhill simplex method. The simplex at the beginning of the step, here a tetrahedron, is shown, top. The simplex at the end of the step can be any one of (a) a reflection away from the high point, (b) a reflection and expansion away from the high point, (c) a contraction along one dimension from the high point, or (d) a contraction along all dimensions towards the low point. An appropriate sequence of such steps will always converge to a minimum of the function.

described in §10.8, also makes use of the geometrical concept of a simplex. Otherwise it is completely unrelated to the algorithm that we are describing in this section.) In general we are only interested in simplexes that are nondegenerate, i.e., that enclose a finite inner N-dimensional volume. If any point of a nondegenerate simplex is taken as the origin, then the N other points define vector directions that span the N-dimensional vector space.

In one-dimensional minimization, it was possible to bracket a minimum, so that the success of a subsequent isolation was guaranteed. Alas! There is no analogous procedure in multidimensional space. For multidimensional minimization, the best we can do is give our algorithm a starting guess, that is, an N-vector of independent variables as the first point to try. The algorithm is then supposed to make its own way

downhill through the unimaginable complexity of an N-dimensional topography, until it encounters a (local, at least) minimum.

The downhill simplex method must be started not just with a single point, but with $N + 1$ points, defining an initial simplex. If you think of one of these points (it matters not which) as being your initial starting point $\mathbf{P}_0$, then you can take the other N points to be

$$\mathbf{P}_i = \mathbf{P}_0 + \lambda \mathbf{e}_i \tag{10.4.1}$$

where the $\mathbf{e}_i$'s are N unit vectors, and where λ is a constant which is your guess of the problem's characteristic length scale. (Or, you could have different λ_i's for each vector direction.)

The downhill simplex method now takes a series of steps, most steps just moving the point of the simplex where the function is largest ("highest point") through the opposite face of the simplex to a lower point. These steps are called reflections, and they are constructed to conserve the volume of the simplex (hence maintain its nondegeneracy). When it can do so, the method expands the simplex in one or another direction to take larger steps. When it reaches a "valley floor," the method contracts itself in the transverse direction and tries to ooze down the valley. If there is a situation where the simplex is trying to "pass through the eye of a needle," it contracts itself in all directions, pulling itself in around its lowest (best) point. The routine name `amoeba` is intended to be descriptive of this kind of behavior; the basic moves are summarized in Figure 10.4.1.

Termination criteria can be delicate in any multidimensional minimization routine. Without bracketing, and with more than one independent variable, we no longer have the option of requiring a certain tolerance for a single independent variable. We typically can identify one "cycle" or "step" of our multidimensional algorithm. It is then possible to terminate when the vector distance moved in that step is fractionally smaller in magnitude than some tolerance `tol`. Alternatively, we could require that the decrease in the function value in the terminating step be fractionally smaller than some tolerance `ftol`. Note that while `tol` should not usually be smaller than the square root of the machine precision, it is perfectly appropriate to let `ftol` be of order the machine precision (or perhaps slightly larger so as not to be diddled by roundoff).

Note well that either of the above criteria might be fooled by a single anomalous step that, for one reason or another, failed to get anywhere. Therefore, it is frequently a good idea to *restart* a multidimensional minimization routine at a point where it claims to have found a minimum. For this restart, you should reinitialize any ancillary input quantities. In the downhill simplex method, for example, you should reinitialize N of the $N + 1$ vertices of the simplex again by equation (10.4.1), with $\mathbf{P}_0$ being one of the vertices of the claimed minimum.

Restarts should never be very expensive; your algorithm did, after all, converge to the restart point once, and now you are starting the algorithm already there.

Consider, then, our N-dimensional amoeba:

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 90 of 181

```
#include <math.h>
#include "nrutil.h"
#define TINY 1.0e-10                    A small number.
#define NMAX 5000                       Maximum allowed number of function evalua-
#define GET_PSUM \                         tions.
                  for (j=1;j<=ndim;j++) {\
                  for (sum=0.0,i=1;i<=mpts;i++) sum += p[i][j];\
                  psum[j]=sum;}
#define SWAP(a,b) {swap=(a);(a)=(b);(b)=swap;}

void amoeba(float **p, float y[], int ndim, float ftol,
    float (*funk)(float []), int *nfunk)
```

Multidimensional minimization of the function `funk(x)` where `x[1..ndim]` is a vector in `ndim` dimensions, by the downhill simplex method of Nelder and Mead. The matrix `p[1..ndim+1][1..ndim]` is input. Its `ndim+1` rows are `ndim`-dimensional vectors which are the vertices of the starting simplex. Also input is the vector `y[1..ndim+1]`, whose components must be pre-initialized to the values of `funk` evaluated at the `ndim+1` vertices (rows) of `p`; and `ftol` the fractional convergence tolerance to be achieved in the function value (n.b.!). On output, `p` and `y` will have been reset to `ndim+1` new points all within `ftol` of a minimum function value, and `nfunk` gives the number of function evaluations taken.

```
{
    float amotry(float **p, float y[], float psum[], int ndim,
        float (*funk)(float []), int ihi, float fac);
    int i,ihi,ilo,inhi,j,mpts=ndim+1;
    float rtol,sum,swap,ysave,ytry,*psum;

    psum=vector(1,ndim);
    *nfunk=0;
    GET_PSUM
    for (;;) {
        ilo=1;
        First we must determine which point is the highest (worst), next-highest, and lowest
        (best), by looping over the points in the simplex.
        ihi = y[1]>y[2] ? (inhi=2,1) : (inhi=1,2);
        for (i=1;i<=mpts;i++) {
            if (y[i] <= y[ilo]) ilo=i;
            if (y[i] > y[ihi]) {
                inhi=ihi;
                ihi=i;
            } else if (y[i] > y[inhi] && i != ihi) inhi=i;
        }
        rtol=2.0*fabs(y[ihi]-y[ilo])/(fabs(y[ihi])+fabs(y[ilo])+TINY);
        Compute the fractional range from highest to lowest and return if satisfactory.
        if (rtol < ftol) {              If returning, put best point and value in slot 1.
            SWAP(y[1],y[ilo])
            for (i=1;i<=ndim;i++) SWAP(p[1][i],p[ilo][i])
            break;
        }
        if (*nfunk >= NMAX) nrerror("NMAX exceeded");
        *nfunk += 2;
        Begin a new iteration. First extrapolate by a factor −1 through the face of the simplex
        across from the high point, i.e., reflect the simplex from the high point.
        ytry=amotry(p,y,psum,ndim,funk,ihi,-1.0);
        if (ytry <= y[ilo])
            Gives a result better than the best point, so try an additional extrapolation by a
            factor 2.
            ytry=amotry(p,y,psum,ndim,funk,ihi,2.0);
        else if (ytry >= y[inhi]) {
            The reflected point is worse than the second-highest, so look for an intermediate
            lower point, i.e., do a one-dimensional contraction.
            ysave=y[ihi];
            ytry=amotry(p,y,psum,ndim,funk,ihi,0.5);
            if (ytry >= ysave) {        Can't seem to get rid of that high point. Better
                for (i=1;i<=mpts;i++) {         contract around the lowest (best) point.
```

```
                    if (i != ilo) {
                        for (j=1;j<=ndim;j++)
                            p[i][j]=psum[j]=0.5*(p[i][j]+p[ilo][j]);
                        y[i]=(*funk)(psum);
                    }
                }
                *nfunk += ndim;          Keep track of function evaluations.
                GET_PSUM                 Recompute psum.
            }
        } else --(*nfunk);               Correct the evaluation count.
    }                                    Go back for the test of doneness and the next
    free_vector(psum,1,ndim);                iteration.
}
```

```
#include "nrutil.h"

float amotry(float **p, float y[], float psum[], int ndim,
    float (*funk)(float []), int ihi, float fac)
```

Extrapolates by a factor fac through the face of the simplex across from the high point, tries it, and replaces the high point if the new point is better.

```
{
    int j;
    float fac1,fac2,ytry,*ptry;

    ptry=vector(1,ndim);
    fac1=(1.0-fac)/ndim;
    fac2=fac1-fac;
    for (j=1;j<=ndim;j++) ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
    ytry=(*funk)(ptry);          Evaluate the function at the trial point.
    if (ytry < y[ihi]) {         If it's better than the highest, then replace the highest.
        y[ihi]=ytry;
        for (j=1;j<=ndim;j++) {
            psum[j] += ptry[j]-p[ihi][j];
            p[ihi][j]=ptry[j];
        }
    }
    free_vector(ptry,1,ndim);
    return ytry;
}
```

CITED REFERENCES AND FURTHER READING:

Nelder, J.A., and Mead, R. 1965, *Computer Journal*, vol. 7, pp. 308–313. [1]

Yarbro, L.A., and Deming, S.N. 1974, *Analytica Chimica Acta*, vol. 73, pp. 391–398.

Jacoby, S.L.S, Kowalik, J.S., and Pizzo, J.T. 1972, *Iterative Methods for Nonlinear Optimization Problems* (Englewood Cliffs, NJ: Prentice-Hall).

10.5 Direction Set (Powell's) Methods in Multidimensions

We know (§10.1–§10.3) how to minimize a function of one variable. If we start at a point **P** in N-dimensional space, and proceed from there in some vector

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 92 of 181

direction $\mathbf{n}$, then any function of N variables $f(\mathbf{P})$ can be minimized along the line $\mathbf{n}$ by our one-dimensional methods. One can dream up various multidimensional minimization methods that consist of sequences of such line minimizations. Different methods will differ only by how, at each stage, they choose the next direction $\mathbf{n}$ to try. All such methods presume the existence of a "black-box" sub-algorithm, which we might call `linmin` (given as an explicit routine at the end of this section), whose definition can be taken for now as

> `linmin`: Given as input the vectors $\mathbf{P}$ and $\mathbf{n}$, and the function f, find the scalar λ that minimizes $f(\mathbf{P}+\lambda\mathbf{n})$. Replace $\mathbf{P}$ by $\mathbf{P}+\lambda\mathbf{n}$. Replace $\mathbf{n}$ by $\lambda\mathbf{n}$. Done.

All the minimization methods in this section and in the two sections following fall under this general schema of successive line minimizations. (The algorithm in §10.7 does not need very accurate line minimizations. Accordingly, it has its own approximate line minimization routine, `lnsrch`.) In this section we consider a class of methods whose choice of successive directions does not involve explicit computation of the function's gradient; the next two sections do require such gradient calculations. You will note that we need not specify whether `linmin` uses gradient information or not. That choice is up to you, and its optimization depends on your particular function. You would be crazy, however, to use gradients in `linmin` and *not* use them in the choice of directions, since in this latter role they can drastically reduce the total computational burden.

But what if, in your application, calculation of the gradient is out of the question. You might first think of this simple method: Take the unit vectors $\mathbf{e}_1, \mathbf{e}_2, \ldots \mathbf{e}_N$ as a *set of directions*. Using `linmin`, move along the first direction to its minimum, then *from there* along the second direction to *its* minimum, and so on, cycling through the whole set of directions as many times as necessary, until the function stops decreasing.

This simple method is actually not too bad for many functions. Even more interesting is why it *is* bad, i.e. very inefficient, for some other functions. Consider a function of two dimensions whose contour map (level lines) happens to define a long, narrow valley at some angle to the coordinate basis vectors (see Figure 10.5.1). Then the only way "down the length of the valley" going along the basis vectors at each stage is by a series of many tiny steps. More generally, in N dimensions, if the function's second derivatives are much larger in magnitude in some directions than in others, then many cycles through all N basis vectors will be required in order to get anywhere. This condition is not all that unusual; according to Murphy's Law, you should count on it.

Obviously what we need is a better set of directions than the $\mathbf{e}_i$'s. All *direction set methods* consist of prescriptions for updating the set of directions as the method proceeds, attempting to come up with a set which either (i) includes some very good directions that will take us far along narrow valleys, or else (more subtly) (ii) includes some number of "non-interfering" directions with the special property that minimization along one is not "spoiled" by subsequent minimization along another, so that interminable cycling through the set of directions can be avoided.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 93 of 181

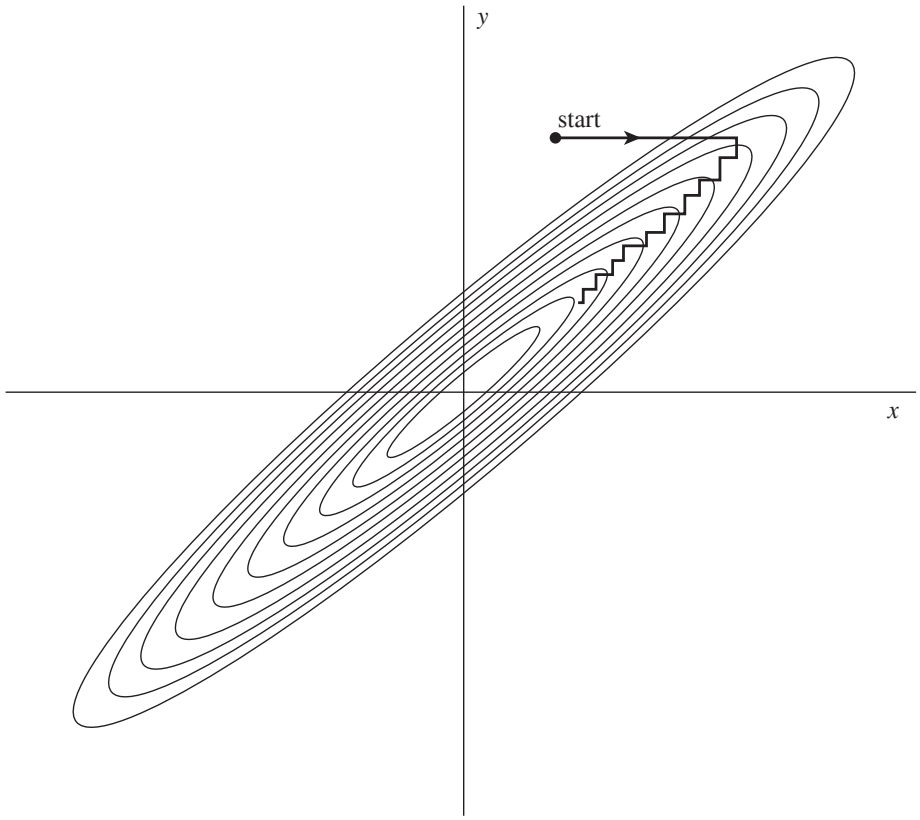

Figure 10.5.1. Successive minimizations along coordinate directions in a long, narrow "valley" (shown as contour lines). Unless the valley is optimally oriented, this method is extremely inefficient, taking many tiny steps to get to the minimum, crossing and re-crossing the principal axis.

Conjugate Directions

This concept of "non-interfering" directions, more conventionally called *conjugate directions*, is worth making mathematically explicit.

First, note that if we minimize a function along some direction **u**, then the gradient of the function must be perpendicular to **u** at the line minimum; if not, then there would still be a nonzero directional derivative along **u**.

Next take some particular point **P** as the origin of the coordinate system with coordinates **x**. Then any function f can be approximated by its Taylor series

$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{P}) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \cdots \\ &\approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \end{aligned} \tag{10.5.1}$$

where

$$c \equiv f(\mathbf{P}) \qquad \mathbf{b} \equiv -\nabla f|_{\mathbf{P}} \qquad [\mathbf{A}]_{ij} \equiv \left. \frac{\partial^2 f}{\partial x_i \partial x_j} \right|_{\mathbf{P}} \tag{10.5.2}$$

The matrix **A** whose components are the second partial derivative matrix of the function is called the *Hessian matrix* of the function at **P**.

In the approximation of (10.5.1), the gradient of f is easily calculated as

$$\nabla f = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{10.5.3}$$

(This implies that the gradient will vanish — the function will be at an extremum — at a value of $\mathbf{x}$ obtained by solving $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. This idea we will return to in §10.7!)

How does the gradient ∇f *change* as we move along some direction? Evidently

$$\delta(\nabla f) = \mathbf{A} \cdot (\delta \mathbf{x}) \tag{10.5.4}$$

Suppose that we have moved along some direction $\mathbf{u}$ to a minimum and now propose to move along some new direction $\mathbf{v}$. The condition that motion along $\mathbf{v}$ not *spoil* our minimization along $\mathbf{u}$ is just that the gradient stay perpendicular to $\mathbf{u}$, i.e., that the change in the gradient be perpendicular to $\mathbf{u}$. By equation (10.5.4) this is just

$$0 = \mathbf{u} \cdot \delta(\nabla f) = \mathbf{u} \cdot \mathbf{A} \cdot \mathbf{v} \tag{10.5.5}$$

When (10.5.5) holds for two vectors $\mathbf{u}$ and $\mathbf{v}$, they are said to be *conjugate*. When the relation holds pairwise for all members of a set of vectors, they are said to be a conjugate set. If you do successive line minimization of a function along a conjugate set of directions, then you don't need to redo any of those directions (unless, of course, you spoil things by minimizing along a direction that they are *not* conjugate to).

A triumph for a direction set method is to come up with a set of N linearly independent, mutually conjugate directions. Then, one pass of N line minimizations will put it exactly at the minimum of a quadratic form like (10.5.1). For functions f that are not exactly quadratic forms, it won't be exactly at the minimum; but repeated cycles of N line minimizations will in due course converge *quadratically* to the minimum.

Powell's Quadratically Convergent Method

Powell first discovered a direction set method that does produce N mutually conjugate directions. Here is how it goes: Initialize the set of directions $\mathbf{u}_i$ to the basis vectors,

$$\mathbf{u}_i = \mathbf{e}_i \qquad i = 1, \ldots, N \tag{10.5.6}$$

Now repeat the following sequence of steps ("basic procedure") until your function stops decreasing:

- Save your starting position as $\mathbf{P}_0$.
- For $i = 1, \ldots, N$, move $\mathbf{P}_{i-1}$ to the minimum along direction $\mathbf{u}_i$ and call this point $\mathbf{P}_i$.
- For $i = 1, \ldots, N-1$, set $\mathbf{u}_i \leftarrow \mathbf{u}_{i+1}$.
- Set $\mathbf{u}_N \leftarrow \mathbf{P}_N - \mathbf{P}_0$.
- Move $\mathbf{P}_N$ to the minimum along direction $\mathbf{u}_N$ and call this point $\mathbf{P}_0$.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 95 of 181

Powell, in 1964, showed that, for a quadratic form like (10.5.1), k iterations of the above basic procedure produce a set of directions $\mathbf{u}_i$ whose last k members are mutually conjugate. Therefore, N iterations of the basic procedure, amounting to $N(N+1)$ line minimizations in all, will exactly minimize a quadratic form. Brent [1] gives proofs of these statements in accessible form.

Unfortunately, there is a problem with Powell's quadratically convergent algorithm. The procedure of throwing away, at each stage, $\mathbf{u}_1$ in favor of $\mathbf{P}_N - \mathbf{P}_0$ tends to produce sets of directions that "fold up on each other" and become linearly dependent. Once this happens, then the procedure finds the minimum of the function f only over a subspace of the full N-dimensional case; in other words, it gives the wrong answer. Therefore, the algorithm must not be used in the form given above.

There are a number of ways to fix up the problem of linear dependence in Powell's algorithm, among them:

1. You can reinitialize the set of directions $\mathbf{u}_i$ to the basis vectors $\mathbf{e}_i$ after every N or $N+1$ iterations of the basic procedure. This produces a serviceable method, which we commend to you if quadratic convergence is important for your application (i.e., if your functions are close to quadratic forms and if you desire high accuracy).

2. Brent points out that the set of directions can equally well be reset to the columns of any orthogonal matrix. Rather than throw away the information on conjugate directions already built up, he resets the direction set to calculated principal directions of the matrix $\mathbf{A}$ (which he gives a procedure for determining). The calculation is essentially a singular value decomposition algorithm (see §2.6). Brent has a number of other cute tricks up his sleeve, and his modification of Powell's method is probably the best presently known. Consult [1] for a detailed description and listing of the program. Unfortunately it is rather too elaborate for us to include here.

3. You can give up the property of quadratic convergence in favor of a more heuristic scheme (due to Powell) which tries to find a few good directions along narrow valleys instead of N necessarily conjugate directions. This is the method that we now implement. (It is also the version of Powell's method given in Acton [2], from which parts of the following discussion are drawn.)

Discarding the Direction of Largest Decrease

The fox and the grapes: Now that we are going to give up the property of quadratic convergence, was it so important after all? That depends on the function that you are minimizing. Some applications produce functions with long, twisty valleys. Quadratic convergence is of no particular advantage to a program which must slalom down the length of a valley floor that twists one way and another (and another, and another, ... – there are N dimensions!). Along the long direction, a quadratically convergent method is trying to extrapolate to the minimum of a parabola which just isn't (yet) there; while the conjugacy of the $N-1$ transverse directions keeps getting spoiled by the twists.

Sooner or later, however, we do arrive at an approximately ellipsoidal minimum (cf. equation 10.5.1 when $\mathbf{b}$, the gradient, is zero). Then, depending on how much accuracy we require, a method with quadratic convergence can save us several times N^2 extra line minimizations, since quadratic convergence *doubles* the number of significant figures at each iteration.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 96 of 181

The basic idea of our now-modified Powell's method is still to take $\mathbf{P}_N - \mathbf{P}_0$ as a new direction; it is, after all, the average direction moved after trying all N possible directions. For a valley whose long direction is twisting slowly, this direction is likely to give us a good run along the new long direction. The change is to discard the old direction along which the function f made its *largest decrease*. This seems paradoxical, since that direction was the *best* of the previous iteration. However, it is also likely to be a major component of the new direction that we are adding, so dropping it gives us the best chance of avoiding a buildup of linear dependence.

There are a couple of exceptions to this basic idea. Sometimes it is better *not* to add a new direction at all. Define

$$f_0 \equiv f(\mathbf{P}_0) \qquad f_N \equiv f(\mathbf{P}_N) \qquad f_E \equiv f(2\mathbf{P}_N - \mathbf{P}_0) \tag{10.5.7}$$

Here f_E is the function value at an "extrapolated" point somewhat further along the proposed new direction. Also define Δf to be the magnitude of the largest decrease along one particular direction of the present basic procedure iteration. (Δf is a positive number.) Then:

1. If $f_E \geq f_0$, then keep the old set of directions for the next basic procedure, because the average direction $\mathbf{P}_N - \mathbf{P}_0$ is all played out.

2. If $2\,(f_0 - 2f_N + f_E)\,[(f_0 - f_N) - \Delta f]^2 \geq (f_0 - f_E)^2 \Delta f$, then keep the old set of directions for the next basic procedure, because either (i) the decrease along the average direction was not primarily due to any single direction's decrease, or (ii) there is a substantial second derivative along the average direction and we seem to be near to the bottom of its minimum.

The following routine implements Powell's method in the version just described. In the routine, `xi` is the matrix whose columns are the set of directions $\mathbf{n}_i$; otherwise the correspondence of notation should be self-evident.

```
#include <math.h>
#include "nrutil.h"
#define TINY 1.0e-25                A small number.
#define ITMAX 200                   Maximum allowed iterations.

void powell(float p[], float **xi, int n, float ftol, int *iter, float *fret,
    float (*func)(float []))
Minimization of a function func of n variables. Input consists of an initial starting point
p[1..n]; an initial matrix xi[1..n][1..n], whose columns contain the initial set of di-
rections (usually the n unit vectors); and ftol, the fractional tolerance in the function value
such that failure to decrease by more than this amount on one iteration signals doneness. On
output, p is set to the best point found, xi is the then-current direction set, fret is the returned
function value at p, and iter is the number of iterations taken. The routine linmin is used.
{
    void linmin(float p[], float xi[], int n, float *fret,
        float (*func)(float []));
    int i,ibig,j;
    float del,fp,fptt,t,*pt,*ptt,*xit;

    pt=vector(1,n);
    ptt=vector(1,n);
    xit=vector(1,n);
    *fret=(*func)(p);
    for (j=1;j<=n;j++) pt[j]=p[j];          Save the initial point.
    for (*iter=1;;++(*iter)) {
        fp=(*fret);
        ibig=0;
```

```
        del=0.0;                        Will be the biggest function decrease.
        for (i=1;i<=n;i++) {            In each iteration, loop over all directions in the set.
            for (j=1;j<=n;j++) xit[j]=xi[j][i];        Copy the direction,
            fptt=(*fret);
            linmin(p,xit,n,fret,func);          minimize along it,
            if (fptt-(*fret) > del) {           and record it if it is the largest decrease
                del=fptt-(*fret);                   so far.
                ibig=i;
            }
        }
        if (2.0*(fp-(*fret)) <= ftol*(fabs(fp)+fabs(*fret))+TINY) {
            free_vector(xit,1,n);               Termination criterion.
            free_vector(ptt,1,n);
            free_vector(pt,1,n);
            return;
        }
        if (*iter == ITMAX) nrerror("powell exceeding maximum iterations.");
        for (j=1;j<=n;j++) {                    Construct the extrapolated point and the
            ptt[j]=2.0*p[j]-pt[j];                  average direction moved. Save the
            xit[j]=p[j]-pt[j];                      old starting point.
            pt[j]=p[j];
        }
        fptt=(*func)(ptt);                      Function value at extrapolated point.
        if (fptt < fp) {
            t=2.0*(fp-2.0*(*fret)+fptt)*SQR(fp-(*fret)-del)-del*SQR(fp-fptt);
            if (t < 0.0) {
                linmin(p,xit,n,fret,func);      Move to the minimum of the new direc-
                for (j=1;j<=n;j++) {                tion, and save the new direction.
                    xi[j][ibig]=xi[j][n];
                    xi[j][n]=xit[j];
                }
            }
        }
    }                                           Back for another iteration.
}
```

Implementation of Line Minimization

Make no mistake, there is a *right* way to implement `linmin`: It is to use the *methods* of one-dimensional minimization described in §10.1–§10.3, but to rewrite the programs of those sections so that their bookkeeping is done on vector-valued points **P** (all lying along a given direction **n**) rather than scalar-valued abscissas x. That straightforward task produces long routines densely populated with "`for(k=1;k<=n;k++)`" loops.

We do not have space to include such routines in this book. Our `linmin`, which works just fine, is instead a kind of bookkeeping swindle. It constructs an "artificial" function of one variable called `f1dim`, which is the value of your function, say, `func`, along the line going through the point `p` in the direction `xi`. `linmin` calls our familiar one-dimensional routines `mnbrak` (§10.1) and `brent` (§10.3) and instructs them to minimize `f1dim`. `linmin` communicates with `f1dim` "over the head" of `mnbrak` and `brent`, through global (external) variables. That is also how it passes to `f1dim` a pointer to your user-supplied function.

The only thing inefficient about `linmin` is this: Its use as an interface between a multidimensional minimization strategy and a one-dimensional minimization routine results in some unnecessary copying of vectors hither and yon. That should not

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 98 of 181

normally be a significant addition to the overall computational burden, but we cannot disguise its inelegance.

```
#include "nrutil.h"
#define TOL 2.0e-4                Tolerance passed to brent.

int ncom;                         Global variables communicate with f1dim.
float *pcom,*xicom,(*nrfunc)(float []);

void linmin(float p[], float xi[], int n, float *fret, float (*func)(float []))
Given an n-dimensional point p[1..n] and an n-dimensional direction xi[1..n], moves and
resets p to where the function func(p) takes on a minimum along the direction xi from p,
and replaces xi by the actual vector displacement that p was moved. Also returns as fret
the value of func at the returned location p. This is actually all accomplished by calling the
routines mnbrak and brent.
{
    float brent(float ax, float bx, float cx,
        float (*f)(float), float tol, float *xmin);
    float f1dim(float x);
    void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb,
        float *fc, float (*func)(float));
    int j;
    float xx,xmin,fx,fb,fa,bx,ax;

    ncom=n;                       Define the global variables.
    pcom=vector(1,n);
    xicom=vector(1,n);
    nrfunc=func;
    for (j=1;j<=n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;                       Initial guess for brackets.
    xx=1.0;
    mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
    *fret=brent(ax,xx,bx,f1dim,TOL,&xmin);
    for (j=1;j<=n;j++) {          Construct the vector results to return.
        xi[j] *= xmin;
        p[j] += xi[j];
    }
    free_vector(xicom,1,n);
    free_vector(pcom,1,n);
}
```

```
#include "nrutil.h"

extern int ncom;                        Defined in linmin.
extern float *pcom,*xicom,(*nrfunc)(float []);

float f1dim(float x)
Must accompany linmin.
{
    int j;
    float f,*xt;

    xt=vector(1,ncom);
    for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
    f=(*nrfunc)(xt);
    free_vector(xt,1,ncom);
    return f;
}
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 7. [1]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 464–467. [2]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), pp. 259–262.

10.6 Conjugate Gradient Methods in Multidimensions

We consider now the case where you are able to calculate, at a given N-dimensional point $\mathbf{P}$, not just the value of a function $f(\mathbf{P})$ but also the gradient (vector of first partial derivatives) $\nabla f(\mathbf{P})$.

A rough counting argument will show how advantageous it is to use the gradient information: Suppose that the function f is roughly approximated as a quadratic form, as above in equation (10.5.1),

$$f(\mathbf{x}) \approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \tag{10.6.1}$$

Then the number of unknown parameters in f is equal to the number of free parameters in $\mathbf{A}$ and $\mathbf{b}$, which is $\frac{1}{2}N(N+1)$, which we see to be of order N^2. Changing any one of these parameters can move the location of the minimum. Therefore, we should not expect to be able to *find* the minimum until we have collected an equivalent information content, of order N^2 numbers.

In the direction set methods of §10.5, we collected the necessary information by making on the order of N^2 separate line minimizations, each requiring "a few" (but sometimes a *big* few!) function evaluations. Now, each evaluation of the gradient will bring us N new components of information. If we use them wisely, we should need to make only of order N separate line minimizations. That is in fact the case for the algorithms in this section and the next.

A factor of N improvement in computational speed is not necessarily implied. As a rough estimate, we might imagine that the calculation of *each component* of the gradient takes about as long as evaluating the function itself. In that case there will be of order N^2 equivalent function evaluations both with and without gradient information. Even if the advantage is not of order N, however, it is nevertheless quite substantial: (i) Each calculated component of the gradient will typically save not just one function evaluation, but a number of them, equivalent to, say, a whole line minimization. (ii) There is often a high degree of redundancy in the formulas for the various components of a function's gradient; when this is so, especially when there is also redundancy with the calculation of the function, then the calculation of the gradient may cost significantly less than N function evaluations.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 100 of 181

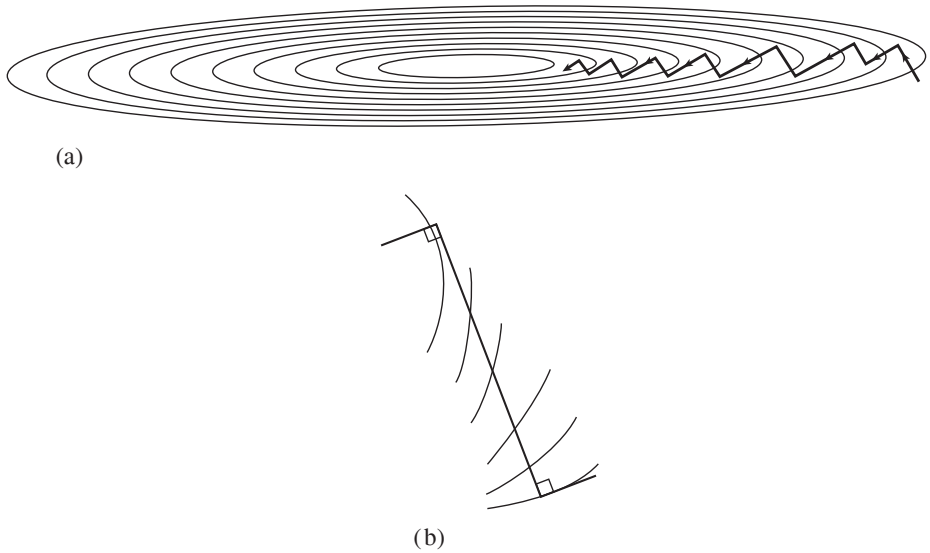

Figure 10.6.1. (a) Steepest descent method in a long, narrow "valley." While more efficient than the strategy of Figure 10.5.1, steepest descent is nonetheless an inefficient strategy, taking many steps to reach the valley floor. (b) Magnified view of one step: A step starts off in the local gradient direction, perpendicular to the contour lines, and traverses a straight line until a local minimum is reached, where the traverse is parallel to the local contour lines.

A common beginner's error is to assume that any reasonable way of incorporating gradient information should be about as good as any other. This line of thought leads to the following *not very good* algorithm, the *steepest descent method*:

> Steepest Descent: Start at a point $\mathbf{P}_0$. As many times as needed, move from point $\mathbf{P}_i$ to the point $\mathbf{P}_{i+1}$ by minimizing along the line from $\mathbf{P}_i$ in the direction of the local downhill gradient $-\nabla f(\mathbf{P}_i)$.

The problem with the steepest descent method (which, incidentally, goes back to Cauchy), is similar to the problem that was shown in Figure 10.5.1. The method will perform many small steps in going down a long, narrow valley, even if the valley is a perfect quadratic form. You might have hoped that, say in two dimensions, your first step would take you to the valley floor, the second step directly down the long axis; but remember that the new gradient at the minimum point of any line minimization is perpendicular to the direction just traversed. Therefore, with the steepest descent method, you *must* make a right angle turn, which does *not*, in general, take you to the minimum. (See Figure 10.6.1.)

Just as in the discussion that led up to equation (10.5.5), we really want a way of proceeding not down the new gradient, but rather in a direction that is somehow constructed to be *conjugate* to the old gradient, and, insofar as possible, to all previous directions traversed. Methods that accomplish this construction are called *conjugate gradient* methods.

In §2.7 we discussed the conjugate gradient method as a technique for solving linear algebraic equations by minimizing a quadratic form. That formalism can also be applied to the problem of minimizing a function *approximated* by the quadratic

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 101 of 181

form (10.6.1). Recall that, starting with an arbitrary initial vector $\mathbf{g}_0$ and letting $\mathbf{h}_0 = \mathbf{g}_0$, the conjugate gradient method constructs two sequences of vectors from the recurrence

$$\mathbf{g}_{i+1} = \mathbf{g}_i - \lambda_i \mathbf{A} \cdot \mathbf{h}_i \qquad \mathbf{h}_{i+1} = \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i \qquad i = 0, 1, 2, \ldots \tag{10.6.2}$$

The vectors satisfy the orthogonality and conjugacy conditions

$$\mathbf{g}_i \cdot \mathbf{g}_j = 0 \qquad \mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_j = 0 \qquad \mathbf{g}_i \cdot \mathbf{h}_j = 0 \qquad j < i \tag{10.6.3}$$

The scalars λ_i and γ_i are given by

$$\lambda_i = \frac{\mathbf{g}_i \cdot \mathbf{g}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} = \frac{\mathbf{g}_i \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \tag{10.6.4}$$

$$\gamma_i = \frac{\mathbf{g}_{i+1} \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{10.6.5}$$

Equations (10.6.2)–(10.6.5) are simply equations (2.7.32)–(2.7.35) for a symmetric $\mathbf{A}$ in a new notation. (A self-contained derivation of these results in the context of function minimization is given by Polak [1].)

Now suppose that we knew the Hessian matrix $\mathbf{A}$ in equation (10.6.1). Then we could use the construction (10.6.2) to find successively conjugate directions $\mathbf{h}_i$ along which to line-minimize. After N such, we would efficiently have arrived at the minimum of the quadratic form. But we don't know $\mathbf{A}$.

Here is a remarkable theorem to save the day: Suppose we happen to have $\mathbf{g}_i = -\nabla f(\mathbf{P}_i)$, for some point $\mathbf{P}_i$, where f is of the form (10.6.1). Suppose that we proceed from $\mathbf{P}_i$ along the direction $\mathbf{h}_i$ to the local minimum of f located at some point $\mathbf{P}_{i+1}$ and then set $\mathbf{g}_{i+1} = -\nabla f(\mathbf{P}_{i+1})$. Then, this $\mathbf{g}_{i+1}$ is the same vector as would have been constructed by equation (10.6.2). (And we have constructed it without knowledge of $\mathbf{A}$!)

Proof: By equation (10.5.3), $\mathbf{g}_i = -\mathbf{A} \cdot \mathbf{P}_i + \mathbf{b}$, and

$$\mathbf{g}_{i+1} = -\mathbf{A} \cdot (\mathbf{P}_i + \lambda \mathbf{h}_i) + \mathbf{b} = \mathbf{g}_i - \lambda \mathbf{A} \cdot \mathbf{h}_i \tag{10.6.6}$$

with λ chosen to take us to the line minimum. But at the line minimum $\mathbf{h}_i \cdot \nabla f = -\mathbf{h}_i \cdot \mathbf{g}_{i+1} = 0$. This latter condition is easily combined with (10.6.6) to solve for λ. The result is exactly the expression (10.6.4). But with this value of λ, (10.6.6) is the same as (10.6.2), q.e.d.

We have, then, the basis of an algorithm that requires neither knowledge of the Hessian matrix $\mathbf{A}$, nor even the storage necessary to store such a matrix. A sequence of directions $\mathbf{h}_i$ is constructed, using only line minimizations, evaluations of the gradient vector, and an auxiliary vector to store the latest in the sequence of $\mathbf{g}$'s.

The algorithm described so far is the original Fletcher-Reeves version of the conjugate gradient algorithm. Later, Polak and Ribiere introduced one tiny, but sometimes significant, change. They proposed using the form

$$\gamma_i = \frac{(\mathbf{g}_{i+1} - \mathbf{g}_i) \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{10.6.7}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 102 of 181

instead of equation (10.6.5). "Wait," you say, "aren't they equal by the orthogonality conditions (10.6.3)?" They are equal for exact quadratic forms. In the real world, however, your function is not exactly a quadratic form. Arriving at the supposed minimum of the quadratic form, you may still need to proceed for another set of iterations. There is some evidence [2] that the Polak-Ribiere formula accomplishes the transition to further iterations more gracefully: When it runs out of steam, it tends to reset **h** to be down the local gradient, which is equivalent to beginning the conjugate-gradient procedure anew.

The following routine implements the Polak-Ribiere variant, which we recommend; but changing one program line, as shown, will give you Fletcher-Reeves. The routine presumes the existence of a function `func(p)`, where `p[1..n]` is a vector of length `n`, and also presumes the existence of a function `dfunc(p,df)` that sets the vector gradient `df[1..n]` evaluated at the input point `p`.

The routine calls `linmin` to do the line minimizations. As already discussed, you may wish to use a modified version of `linmin` that uses `dbrent` instead of `brent`, i.e., that uses the gradient in doing the line minimizations. See note below.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 200
#define EPS 1.0e-10
```

Here `ITMAX` is the maximum allowed number of iterations, while `EPS` is a small number to rectify the special case of converging to exactly zero function value.

```
#define FREEALL free_vector(xi,1,n);free_vector(h,1,n);free_vector(g,1,n);

void frprmn(float p[], int n, float ftol, int *iter, float *fret,
    float (*func)(float []), void (*dfunc)(float [], float []))
```

Given a starting point `p[1..n]`, Fletcher-Reeves-Polak-Ribiere minimization is performed on a function `func`, using its gradient as calculated by a routine `dfunc`. The convergence tolerance on the function value is input as `ftol`. Returned quantities are `p` (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function). The routine `linmin` is called to perform line minimizations.

```
{
    void linmin(float p[], float xi[], int n, float *fret,
        float (*func)(float []));
    int j,its;
    float gg,gam,fp,dgg;
    float *g,*h,*xi;

    g=vector(1,n);
    h=vector(1,n);
    xi=vector(1,n);
    fp=(*func)(p);                          Initializations.
    (*dfunc)(p,xi);
    for (j=1;j<=n;j++) {
        g[j] = -xi[j];
        xi[j]=h[j]=g[j];
    }
    for (its=1;its<=ITMAX;its++) {          Loop over iterations.
        *iter=its;
        linmin(p,xi,n,fret,func);           Next statement is the normal return:
        if (2.0*fabs(*fret-fp) <= ftol*(fabs(*fret)+fabs(fp)+EPS)) {
            FREEALL
            return;
        }
        fp= *fret;
        (*dfunc)(p,xi);
        dgg=gg=0.0;
        for (j=1;j<=n;j++) {
```

```
            gg += g[j]*g[j];
/*         dgg += xi[j]*xi[j];    */          This statement for Fletcher-Reeves.
            dgg += (xi[j]+g[j])*xi[j];        This statement for Polak-Ribiere.
        }
        if (gg == 0.0) {                      Unlikely. If gradient is exactly zero then
            FREEALL                               we are already done.
            return;
        }
        gam=dgg/gg;
        for (j=1;j<=n;j++) {
            g[j] = -xi[j];
            xi[j]=h[j]=g[j]+gam*h[j];
        }
    }
    nrerror("Too many iterations in frprmn");
}
```

Note on Line Minimization Using Derivatives

Kindly reread the last part of §10.5. We here want to do the same thing, but using derivative information in performing the line minimization.

The modified version of `linmin`, called `dlinmin`, and its required companion routine `df1dim` follow:

```
#include "nrutil.h"
#define TOL 2.0e-4                Tolerance passed to dbrent.

int ncom;                         Global variables communicate with df1dim.
float *pcom,*xicom,(*nrfunc)(float []);
void (*nrdfun)(float [], float []);

void dlinmin(float p[], float xi[], int n, float *fret, float (*func)(float []),
    void (*dfunc)(float [], float []))
Given an n-dimensional point p[1..n] and an n-dimensional direction xi[1..n], moves and
resets p to where the function func(p) takes on a minimum along the direction xi from p,
and replaces xi by the actual vector displacement that p was moved. Also returns as fret
the value of func at the returned location p. This is actually all accomplished by calling the
routines mnbrak and dbrent.
{
    float dbrent(float ax, float bx, float cx,
        float (*f)(float), float (*df)(float), float tol, float *xmin);
    float f1dim(float x);
    float df1dim(float x);
    void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb,
        float *fc, float (*func)(float));
    int j;
    float xx,xmin,fx,fb,fa,bx,ax;

    ncom=n;                       Define the global variables.
    pcom=vector(1,n);
    xicom=vector(1,n);
    nrfunc=func;
    nrdfun=dfunc;
    for (j=1;j<=n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;                       Initial guess for brackets.
    xx=1.0;
    mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
```

```
    *fret=dbrent(ax,xx,bx,f1dim,df1dim,TOL,&xmin);
    for (j=1;j<=n;j++) {        Construct the vector results to return.
        xi[j] *= xmin;
        p[j] += xi[j];
    }
    free_vector(xicom,1,n);
    free_vector(pcom,1,n);
}
```

```
#include "nrutil.h"

extern int ncom;                        Defined in dlinmin.
extern float *pcom,*xicom,(*nrfunc)(float []);
extern void (*nrdfun)(float [], float []);

float df1dim(float x)
{
    int j;
    float df1=0.0;
    float *xt,*df;

    xt=vector(1,ncom);
    df=vector(1,ncom);
    for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
    (*nrdfun)(xt,df);
    for (j=1;j<=ncom;j++) df1 += df[j]*xicom[j];
    free_vector(df,1,ncom);
    free_vector(xt,1,ncom);
    return df1;
}
```

CITED REFERENCES AND FURTHER READING:

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), §2.3. [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.7 (by K.W. Brodlie). [2]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §8.7.

10.7 Variable Metric Methods in Multidimensions

The goal of *variable metric* methods, which are sometimes called *quasi-Newton* methods, is not different from the goal of conjugate gradient methods: to accumulate information from successive line minimizations so that N such line minimizations lead to the exact minimum of a quadratic form in N dimensions. In that case, the method will also be quadratically convergent for more general smooth functions.

Both variable metric and conjugate gradient methods require that you are able to compute your function's gradient, or first partial derivatives, at arbitrary points. The variable metric approach differs from the conjugate gradient in the way that it stores

and updates the information that is accumulated. Instead of requiring intermediate storage on the order of N, the number of dimensions, it requires a matrix of size $N \times N$. Generally, for any moderate N, this is an entirely trivial disadvantage.

On the other hand, there is not, as far as we know, any overwhelming advantage that the variable metric methods hold over the conjugate gradient techniques, except perhaps a historical one. Developed somewhat earlier, and more widely propagated, the variable metric methods have by now developed a wider constituency of satisfied users. Likewise, some fancier implementations of variable metric methods (going beyond the scope of this book, see below) have been developed to a greater level of sophistication on issues like the minimization of roundoff error, handling of special conditions, and so on. *We* tend to use variable metric rather than conjugate gradient, but we have no reason to urge this habit on you.

Variable metric methods come in two main flavors. One is the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to as simply *Fletcher-Powell*). The other goes by the name *Broyden-Fletcher-Goldfarb-Shanno (BFGS)*. The BFGS and DFP schemes differ only in details of their roundoff error, convergence tolerances, and similar "dirty" issues which are outside of our scope [1,2]. However, it has become generally recognized that, empirically, the BFGS scheme is superior in these details. We will implement BFGS in this section.

As before, we imagine that our arbitrary function $f(\mathbf{x})$ can be locally approximated by the quadratic form of equation (10.6.1). We don't, however, have any information about the values of the quadratic form's parameters $\mathbf{A}$ and $\mathbf{b}$, except insofar as we can glean such information from our function evaluations and line minimizations.

The basic idea of the variable metric method is to build up, iteratively, a good approximation to the inverse Hessian matrix $\mathbf{A}^{-1}$, that is, to construct a sequence of matrices $\mathbf{H}_i$ with the property,

$$\lim_{i\to\infty} \mathbf{H}_i = \mathbf{A}^{-1} \tag{10.7.1}$$

Even better if the limit is achieved after N iterations instead of ∞.

The reason that variable metric methods are sometimes called quasi-Newton methods can now be explained. Consider finding a minimum by using Newton's method to search for a zero of the gradient of the function. Near the current point $\mathbf{x}_i$, we have to second order

$$f(\mathbf{x}) = f(\mathbf{x}_i) + (\mathbf{x} - \mathbf{x}_i) \cdot \nabla f(\mathbf{x}_i) + \tfrac{1}{2}(\mathbf{x} - \mathbf{x}_i) \cdot \mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) \tag{10.7.2}$$

so

$$\nabla f(\mathbf{x}) = \nabla f(\mathbf{x}_i) + \mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) \tag{10.7.3}$$

In Newton's method we set $\nabla f(\mathbf{x}) = 0$ to determine the next iteration point:

$$\mathbf{x} - \mathbf{x}_i = -\mathbf{A}^{-1} \cdot \nabla f(\mathbf{x}_i) \tag{10.7.4}$$

The left-hand side is the finite step we need take to get to the exact minimum; the right-hand side is known once we have accumulated an accurate $\mathbf{H} \approx \mathbf{A}^{-1}$.

The "quasi" in quasi-Newton is because we don't use the actual Hessian matrix of f, but instead use our current approximation of it. This is often *better* than

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 106 of 181

using the true Hessian. We can understand this paradoxical result by considering the *descent directions* of f at $\mathbf{x}_i$. These are the directions $\mathbf{p}$ along which f decreases: $\nabla f \cdot \mathbf{p} < 0$. For the Newton direction (10.7.4) to be a descent direction, we must have

$$\nabla f(\mathbf{x}_i) \cdot (\mathbf{x} - \mathbf{x}_i) = -(\mathbf{x} - \mathbf{x}_i) \cdot \mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) < 0 \tag{10.7.5}$$

which is true if $\mathbf{A}$ is positive definite. In general, far from a minimum, we have no guarantee that the Hessian is positive definite. Taking the actual Newton step with the real Hessian can move us to points where the function is *increasing* in value. The idea behind quasi-Newton methods is to start with a positive definite, symmetric approximation to $\mathbf{A}$ (usually the unit matrix) and build up the approximating $\mathbf{H}_i$'s in such a way that the matrix $\mathbf{H}_i$ remains positive definite and symmetric. Far from the minimum, this guarantees that we always move in a downhill direction. Close to the minimum, the updating formula approaches the true Hessian and we enjoy the quadratic convergence of Newton's method.

When we are not close enough to the minimum, taking the full Newton step $\mathbf{p}$ even with a positive definite $\mathbf{A}$ need not decrease the function; we may move too far for the quadratic approximation to be valid. All we are guaranteed is that *initially* f decreases as we move in the Newton direction. Once again we can use the backtracking strategy described in §9.7 to choose a step along the *direction* of the Newton step $\mathbf{p}$, but not necessarily all the way.

We won't rigorously derive the DFP algorithm for taking $\mathbf{H}_i$ into $\mathbf{H}_{i+1}$; you can consult [3] for clear derivations. Following Brodlie (in [2]), we will give the following heuristic motivation of the procedure.

Subtracting equation (10.7.4) at $\mathbf{x}_{i+1}$ from that same equation at $\mathbf{x}_i$ gives

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{A}^{-1} \cdot (\nabla f_{i+1} - \nabla f_i) \tag{10.7.6}$$

where $\nabla f_j \equiv \nabla f(\mathbf{x}_j)$. Having made the step from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$, we might reasonably want to require that the new approximation $\mathbf{H}_{i+1}$ satisfy (10.7.6) as if it were actually $\mathbf{A}^{-1}$, that is,

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{H}_{i+1} \cdot (\nabla f_{i+1} - \nabla f_i) \tag{10.7.7}$$

We might also imagine that the updating formula should be of the form $\mathbf{H}_{i+1} = \mathbf{H}_i +$ correction.

What "objects" are around out of which to construct a correction term? Most notable are the two vectors $\mathbf{x}_{i+1} - \mathbf{x}_i$ and $\nabla f_{i+1} - \nabla f_i$; and there is also $\mathbf{H}_i$. There are not infinitely many natural ways of making a matrix out of these objects, especially if (10.7.7) must hold! One such way, the *DFP updating formula*, is

$$\begin{aligned}\mathbf{H}_{i+1} = \mathbf{H}_i &+ \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i) \otimes (\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} \\ &- \frac{[\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)] \otimes [\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)}\end{aligned} \tag{10.7.8}$$

where $\otimes$ denotes the "outer" or "direct" product of two vectors, a matrix: The ij component of $\mathbf{u} \otimes \mathbf{v}$ is $u_i v_j$. (You might want to verify that 10.7.8 does satisfy 10.7.7.)

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 107 of 181

The *BFGS updating formula* is exactly the same, but with one additional term,

$$\cdots + [(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]\, \mathbf{u} \otimes \mathbf{u} \tag{10.7.9}$$

where $\mathbf{u}$ is defined as the vector

$$\begin{aligned} \mathbf{u} \equiv & \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} \\ & - \frac{\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \end{aligned} \tag{10.7.10}$$

(You might also verify that this satisfies 10.7.7.)

You will have to take on faith — or else consult [3] for details of — the "deep" result that equation (10.7.8), with or without (10.7.9), does in fact converge to $\mathbf{A}^{-1}$ in N steps, if f is a quadratic form.

Here now is the routine `dfpmin` that implements the quasi-Newton method, and uses `lnsrch` from §9.7. As mentioned at the end of `newt` in §9.7, this algorithm can fail if your variables are badly scaled.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 200                          Maximum allowed number of iterations.
#define EPS 3.0e-8                         Machine precision.
#define TOLX (4*EPS)                       Convergence criterion on x values.
#define STPMX 100.0                        Scaled maximum step length allowed in
                                               line searches.
#define FREEALL free_vector(xi,1,n);free_vector(pnew,1,n); \
free_matrix(hessin,1,n,1,n);free_vector(hdg,1,n);free_vector(g,1,n); \
free_vector(dg,1,n);

void dfpmin(float p[], int n, float gtol, int *iter, float *fret,
    float(*func)(float []), void (*dfunc)(float [], float []))
Given a starting point p[1..n] that is a vector of length n, the Broyden-Fletcher-Goldfarb-
Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function func, using
its gradient as calculated by a routine dfunc. The convergence requirement on zeroing the
gradient is input as gtol. Returned quantities are p[1..n] (the location of the minimum),
iter (the number of iterations that were performed), and fret (the minimum value of the
function). The routine lnsrch is called to perform approximate line minimizations.
{
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
         float *f, float stpmax, int *check, float (*func)(float []));
    int check,i,its,j;
    float den,fac,fad,fae,fp,stpmax,sum=0.0,sumdg,sumxi,temp,test;
    float *dg,*g,*hdg,**hessin,*pnew,*xi;

    dg=vector(1,n);
    g=vector(1,n);
    hdg=vector(1,n);
    hessin=matrix(1,n,1,n);
    pnew=vector(1,n);
    xi=vector(1,n);
    fp=(*func)(p);                         Calculate starting function value and gra-
    (*dfunc)(p,g);                             dient,
    for (i=1;i<=n;i++) {                   and initialize the inverse Hessian to the
        for (j=1;j<=n;j++) hessin[i][j]=0.0;   unit matrix.
        hessin[i][i]=1.0;
        xi[i] = -g[i];                     Initial line direction.
        sum += p[i]*p[i];
    }
    stpmax=STPMX*FMAX(sqrt(sum),(float)n);
```

```
    for (its=1;its<=ITMAX;its++) {              Main loop over the iterations.
        *iter=its;
        lnsrch(n,p,fp,g,xi,pnew,fret,stpmax,&check,func);
        The new function evaluation occurs in lnsrch; save the function value in fp for the
        next line search. It is usually safe to ignore the value of check.
        fp = *fret;
        for (i=1;i<=n;i++) {
            xi[i]=pnew[i]-p[i];                 Update the line direction,
            p[i]=pnew[i];                       and the current point.
        }
        test=0.0;                               Test for convergence on Δx.
        for (i=1;i<=n;i++) {
            temp=fabs(xi[i])/FMAX(fabs(p[i]),1.0);
            if (temp > test) test=temp;
        }
        if (test < TOLX) {
            FREEALL
            return;
        }
        for (i=1;i<=n;i++) dg[i]=g[i];          Save the old gradient,
        (*dfunc)(p,g);                          and get the new gradient.
        test=0.0;                               Test for convergence on zero gradient.
        den=FMAX(*fret,1.0);
        for (i=1;i<=n;i++) {
            temp=fabs(g[i])*FMAX(fabs(p[i]),1.0)/den;
            if (temp > test) test=temp;
        }
        if (test < gtol) {
            FREEALL
            return;
        }
        for (i=1;i<=n;i++) dg[i]=g[i]-dg[i];    Compute difference of gradients,
        for (i=1;i<=n;i++) {                    and difference times current matrix.
            hdg[i]=0.0;
            for (j=1;j<=n;j++) hdg[i] += hessin[i][j]*dg[j];
        }
        fac=fae=sumdg=sumxi=0.0;                Calculate dot products for the denomi-
        for (i=1;i<=n;i++) {                        nators.
            fac += dg[i]*xi[i];
            fae += dg[i]*hdg[i];
            sumdg += SQR(dg[i]);
            sumxi += SQR(xi[i]);
        }
        if (fac > sqrt(EPS*sumdg*sumxi)) {      Skip update if fac not sufficiently posi-
            fac=1.0/fac;                            tive.
            fad=1.0/fae;
            The vector that makes BFGS different from DFP:
            for (i=1;i<=n;i++) dg[i]=fac*xi[i]-fad*hdg[i];
            for (i=1;i<=n;i++) {                The BFGS updating formula:
                for (j=i;j<=n;j++) {
                    hessin[i][j] += fac*xi[i]*xi[j]
                    -fad*hdg[i]*hdg[j]+fae*dg[i]*dg[j];
                    hessin[j][i]=hessin[i][j];
                }
            }
        }
        for (i=1;i<=n;i++) {                    Now calculate the next direction to go,
            xi[i]=0.0;
            for (j=1;j<=n;j++) xi[i] -= hessin[i][j]*g[j];
        }
    }                                           and go back for another iteration.
    nrerror("too many iterations in dfpmin");
    FREEALL
}
```

Quasi-Newton methods like `dfpmin` work well with the approximate line minimization done by `lnsrch`. The routines `powell` (§10.5) and `frprmn` (§10.6), however, need more accurate line minimization, which is carried out by the routine `linmin`.

Advanced Implementations of Variable Metric Methods

Although rare, it can conceivably happen that roundoff errors cause the matrix $\mathbf{H}_i$ to become nearly singular or non-positive-definite. This can be serious, because the supposed search directions might then not lead downhill, and because nearly singular $\mathbf{H}_i$'s tend to give subsequent $\mathbf{H}_i$'s that are also nearly singular.

There is a simple fix for this rare problem, the same as was mentioned in §10.4: In case of any doubt, you should *restart* the algorithm at the claimed minimum point, and see if it goes anywhere. Simple, but not very elegant. Modern implementations of variable metric methods deal with the problem in a more sophisticated way.

Instead of building up an approximation to $\mathbf{A}^{-1}$, it is possible to build up an approximation of $\mathbf{A}$ itself. Then, instead of calculating the left-hand side of (10.7.4) directly, one solves the set of linear equations

$$\mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) = -\nabla f(\mathbf{x}_i) \tag{10.7.11}$$

At first glance this seems like a bad idea, since solving (10.7.11) is a process of order N^3 — and anyway, how does this help the roundoff problem? The trick is not to store $\mathbf{A}$ but rather a triangular decomposition of $\mathbf{A}$, its *Cholesky decomposition* (cf. §2.9). The updating formula used for the Cholesky decomposition of $\mathbf{A}$ is of order N^2 and can be arranged to guarantee that the matrix remains positive definite and nonsingular, even in the presence of finite roundoff. This method is due to Gill and Murray [1,2].

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1, §§3–6 (by K. W. Brodlie). [2]

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), pp. 56ff. [3]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 467–468.

10.8 Linear Programming and the Simplex Method

The subject of *linear programming*, sometimes called *linear optimization*, concerns itself with the following problem: For N independent variables $x_1, \ldots, x_N$, *maximize* the function

$$z = a_{01}x_1 + a_{02}x_2 + \cdots + a_{0N}x_N \tag{10.8.1}$$

subject to the primary constraints

$$x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots \quad x_N \geq 0 \tag{10.8.2}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 110 of 181

and simultaneously subject to $M = m_1 + m_2 + m_3$ additional constraints, m_1 of them of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{iN}x_N \le b_i \qquad (b_i \ge 0) \qquad i = 1, \ldots, m_1 \tag{10.8.3}$$

m_2 of them of the form

$$a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jN}x_N \ge b_j \ge 0 \qquad j = m_1 + 1, \ldots, m_1 + m_2 \tag{10.8.4}$$

and m_3 of them of the form

$$\begin{aligned} a_{k1}x_1 + a_{k2}x_2 + \cdots + a_{kN}x_N = b_k \ge 0 \\ k = m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 \end{aligned} \tag{10.8.5}$$

The various a_{ij}'s can have either sign, or be zero. The fact that the b's must all be nonnegative (as indicated by the final inequality in the above three equations) is a matter of convention only, since you can multiply any contrary inequality by -1. There is no particular significance in the number of constraints M being less than, equal to, or greater than the number of unknowns N.

A set of values $x_1 \ldots x_N$ that satisfies the constraints (10.8.2)–(10.8.5) is called a *feasible vector*. The function that we are trying to maximize is called the *objective function*. The feasible vector that maximizes the objective function is called the *optimal feasible vector*. An optimal feasible vector can fail to exist for two distinct reasons: (i) there are *no* feasible vectors, i.e., the given constraints are incompatible, or (ii) there is no maximum, i.e., there is a direction in N space where one or more of the variables can be taken to infinity while still satisfying the constraints, giving an unbounded value for the objective function.

As you see, the subject of linear programming is surrounded by notational and terminological thickets. Both of these thorny defenses are lovingly cultivated by a coterie of stern acolytes who have devoted themselves to the field. Actually, the basic ideas of linear programming are quite simple. Avoiding the shrubbery, we want to teach you the basics by means of a couple of specific examples; it should then be quite obvious how to generalize.

Why is linear programming so important? (i) Because "nonnegativity" is the usual constraint on any variable x_i that represents the tangible amount of some physical commodity, like guns, butter, dollars, units of vitamin E, food calories, kilowatt hours, mass, etc. Hence equation (10.8.2). (ii) Because one is often interested in additive (linear) limitations or bounds imposed by man or nature: minimum nutritional requirement, maximum affordable cost, maximum on available labor or capital, minimum tolerable level of voter approval, etc. Hence equations (10.8.3)–(10.8.5). (iii) Because the function that one wants to optimize may be linear, or else may at least be approximated by a linear function — since that is the problem that linear programming *can* solve. Hence equation (10.8.1). For a short, semipopular survey of linear programming applications, see Bland [1].

Here is a specific example of a problem in linear programming, which has $N = 4$, $m_1 = 2$, $m_2 = m_3 = 1$, hence $M = 4$:

$$\text{Maximize} \quad z = x_1 + x_2 + 3x_3 - \tfrac{1}{2}x_4 \tag{10.8.6}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 111 of 181

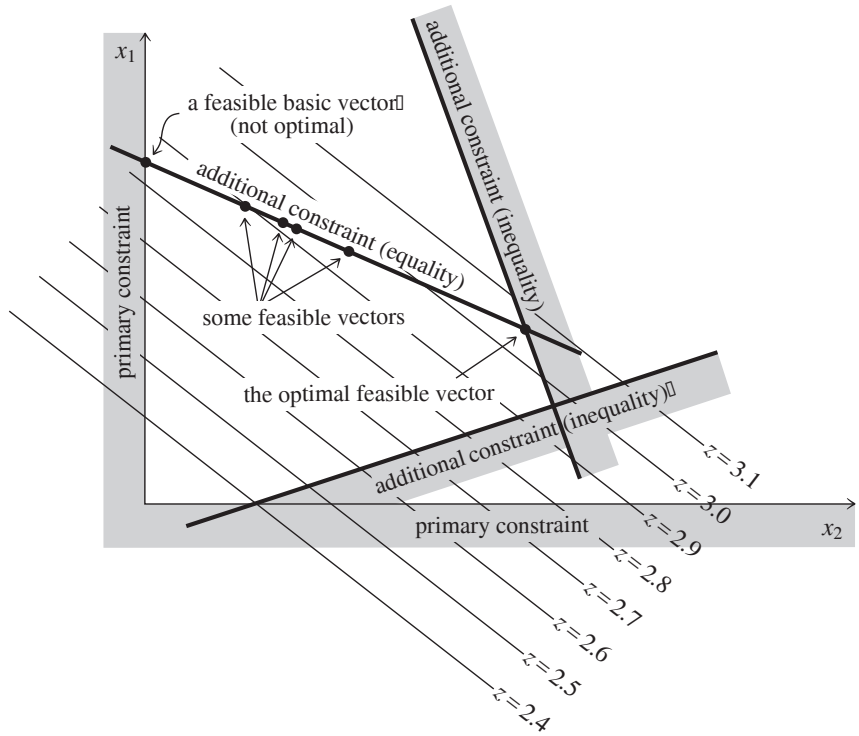

Figure 10.8.1. Basic concepts of linear programming. The case of only two independent variables, x_1, x_2, is shown. The linear function z, to be maximized, is represented by its contour lines. Primary constraints require x_1 and x_2 to be positive. Additional constraints may restrict the solution to regions (inequality constraints) or to surfaces of lower dimensionality (equality constraints). Feasible vectors satisfy all constraints. Feasible basic vectors also lie on the boundary of the allowed region. The simplex method steps among feasible basic vectors until the optimal feasible vector is found.

with all the x's nonnegative and also with

$$\begin{aligned} x_1 + 2x_3 &\le 740 \\ 2x_2 - 7x_4 &\le 0 \\ x_2 - x_3 + 2x_4 &\ge \tfrac{1}{2} \\ x_1 + x_2 + x_3 + x_4 &= 9 \end{aligned} \tag{10.8.7}$$

The answer turns out to be (to 2 decimals) $x_1 = 0$, $x_2 = 3.33$, $x_3 = 4.73$, $x_4 = 0.95$. In the rest of this section we will learn how this answer is obtained. Figure 10.8.1 summarizes some of the terminology thus far.

Fundamental Theorem of Linear Optimization

Imagine that we start with a full N-dimensional space of candidate vectors. Then (in mind's eye, at least) we carve away the regions that are eliminated in turn by each imposed constraint. Since the constraints are linear, every boundary introduced by this process is a plane, or rather hyperplane. Equality constraints of the form (10.8.5)

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 112 of 181

force the feasible region onto hyperplanes of smaller dimension, while inequalities simply divide the then-feasible region into allowed and nonallowed pieces.

When all the constraints are imposed, either we are left with some feasible region or else there are no feasible vectors. Since the feasible region is bounded by hyperplanes, it is geometrically a kind of convex polyhedron or simplex (cf. §10.4). If there is a feasible region, can the optimal feasible vector be somewhere in its interior, away from the boundaries? No, because the objective function is linear. This means that it always has a nonzero vector gradient. This, in turn, means that we could always increase the objective function by running up the gradient until we hit a boundary wall.

The boundary of any geometrical region has one less dimension than its interior. Therefore, we can now run up the gradient projected into the boundary wall until we reach an edge of that wall. We can then run up that edge, and so on, down through whatever number of dimensions, until we finally arrive at a point, a *vertex* of the original simplex. Since this point has all N of its coordinates defined, it must be the solution of N simultaneous *equalities* drawn from the original set of equalities and inequalities (10.8.2)–(10.8.5).

Points that are feasible vectors and that satisfy N of the original constraints as equalities, are termed *feasible basic vectors*. If $N > M$, then a feasible basic vector has *at least* $N - M$ of its components equal to zero, since at least that many of the constraints (10.8.2) will be needed to make up the total of N. Put the other way, *at most* M components of a feasible basic vector are nonzero. In the example (10.8.6)–(10.8.7), you can check that the solution as given satisfies as equalities the last three constraints of (10.8.7) and the constraint $x_1 \geq 0$, for the required total of 4.

Put together the two preceding paragraphs and you have the *Fundamental Theorem of Linear Optimization*: If an optimal feasible vector exists, then there is a feasible basic vector that is optimal. (Didn't we warn you about the terminological thicket?)

The importance of the fundamental theorem is that it reduces the optimization problem to a "combinatorial" problem, that of determining which N constraints (out of the $M + N$ constraints in 10.8.2–10.8.5) should be satisfied by the optimal feasible vector. We have only to keep trying different combinations, and computing the objective function for each trial, until we find the best.

Doing this blindly would take halfway to forever. The *simplex method*, first published by Dantzig in 1948 (see [2]), is a way of organizing the procedure so that (i) a series of combinations is tried for which the objective function increases at each step, and (ii) the optimal feasible vector is reached after a number of iterations that is almost always no larger than of order M or N, whichever is larger. An interesting mathematical sidelight is that this second property, although known empirically ever since the simplex method was devised, was not proved to be true until the 1982 work of Stephen Smale. (For a contemporary account, see [3].)

Simplex Method for a Restricted Normal Form

A linear programming problem is said to be in *normal form* if it has no constraints in the form (10.8.3) or (10.8.4), but rather only equality constraints of the form (10.8.5) and nonnegativity constraints of the form (10.8.2).

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 113 of 181

For our purposes it will be useful to consider an even more restricted set of cases, with this additional property: Each equality constraint of the form (10.8.5) must have at least one variable that has a positive coefficient and *that appears uniquely in that one constraint only*. We can then choose one such variable in each constraint equation, and solve that constraint equation for it. The variables thus chosen are called *left-hand variables* or *basic variables*, and there are exactly M $(= m_3)$ of them. The remaining $N - M$ variables are called *right-hand variables* or *nonbasic variables*. Obviously this *restricted normal form* can be achieved only in the case $M \leq N$, so that is the case that we will consider.

You may be thinking that our restricted normal form is so specialized that it is unlikely to include the linear programming problem that you wish to solve. Not at all! We will presently show how *any* linear programming problem can be transformed into restricted normal form. Therefore bear with us and learn how to apply the simplex method to a restricted normal form.

Here is an example of a problem in restricted normal form:

$$\text{Maximize} \quad z = 2x_2 - 4x_3 \tag{10.8.8}$$

with x_1, x_2, x_3, and x_4 all nonnegative and also with

$$\begin{aligned} x_1 &= 2 - 6x_2 + x_3 \\ x_4 &= 8 + 3x_2 - 4x_3 \end{aligned} \tag{10.8.9}$$

This example has $N = 4$, $M = 2$; the left-hand variables are x_1 and x_4; the right-hand variables are x_2 and x_3. The objective function (10.8.8) is written so as to depend only on right-hand variables; note, however, that this is not an actual restriction on objective functions in restricted normal form, since any left-hand variables appearing in the objective function could be eliminated algebraically by use of (10.8.9) or its analogs.

For any problem in restricted normal form, we can instantly read off a feasible basic vector (although not necessarily the *optimal* feasible basic vector). Simply set all right-hand variables equal to zero, and equation (10.8.9) then gives the values of the left-hand variables for which the constraints are satisfied. The idea of the simplex method is to proceed by a series of exchanges. In each exchange, a right-hand variable and a left-hand variable change places. At each stage we maintain a problem in restricted normal form that is equivalent to the original problem.

It is notationally convenient to record the information content of equations (10.8.8) and (10.8.9) in a so-called *tableau*, as follows:

		x_2	x_3
z	0	2	-4
x_1	2	-6	1
x_4	8	3	-4

(10.8.10)

You should study (10.8.10) to be sure that you understand where each entry comes from, and how to translate back and forth between the tableau and equation formats of a problem in restricted normal form.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 114 of 181

The first step in the simplex method is to examine the top row of the tableau, which we will call the "z-row." Look at the entries in columns labeled by right-hand variables (we will call these "right-columns"). We want to imagine in turn the effect of increasing each right-hand variable from its present value of zero, while leaving all the other right-hand variables at zero. Will the objective function increase or decrease? The answer is given by the sign of the entry in the z-row. Since we want to increase the objective function, only right columns having positive z-row entries are of interest. In (10.8.10) there is only one such column, whose z-row entry is 2.

The second step is to examine the column entries below each z-row entry that was selected by step one. We want to ask how much we can increase the right-hand variable before one of the left-hand variables is driven negative, which is not allowed. If the tableau element at the intersection of the right-hand column and the left-hand variable's row is positive, then it poses no restriction: the corresponding left-hand variable will just be driven more and more positive. If *all* the entries in any right-hand column are positive, then there is no bound on the objective function and (having said so) we are done with the problem.

If one or more entries below a positive z-row entry are negative, then we have to figure out which such entry first limits the increase of that column's right-hand variable. Evidently the limiting increase is given by dividing the element in the right-hand column (which is called the *pivot element*) into the element in the "constant column" (leftmost column) of the pivot element's row. A value that is small in magnitude is most restrictive. The increase in the objective function for this choice of pivot element is then that value multiplied by the z-row entry of that column. We repeat this procedure on all possible right-hand columns to find the pivot element with the largest such increase. That completes our "choice of a pivot element."

In the above example, the only positive z-row entry is 2. There is only one negative entry below it, namely -6, so this is the pivot element. Its constant-column entry is 2. This pivot will therefore allow x_2 to be increased by $2 \div |6|$, which results in an increase of the objective function by an amount $(2 \times 2) \div |6|$.

The third step is to *do* the increase of the selected right-hand variable, thus making it a left-hand variable; and simultaneously to modify the left-hand variables, reducing the pivot-row element to zero and thus making it a right-hand variable. For our above example let's do this first by hand: We begin by solving the pivot-row equation for the new left-hand variable x_2 in favor of the old one x_1, namely

$$x_1 = 2 - 6x_2 + x_3 \qquad \rightarrow \qquad x_2 = \tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3 \tag{10.8.11}$$

We then substitute this into the old z-row,

$$z = 2x_2 - 4x_3 = 2\left[\tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3\right] - 4x_3 = \tfrac{2}{3} - \tfrac{1}{3}x_1 - \tfrac{11}{3}x_3 \tag{10.8.12}$$

and into all other left-variable rows, in this case only x_4,

$$x_4 = 8 + 3\left[\tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3\right] - 4x_3 = 9 - \tfrac{1}{2}x_1 - \tfrac{7}{2}x_3 \tag{10.8.13}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 115 of 181

Equations (10.8.11)–(10.8.13) form the new tableau

		x_1	x_3
z	$\frac{2}{3}$	$-\frac{1}{3}$	$-\frac{11}{3}$
x_2	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$
x_4	9	$-\frac{1}{2}$	$-\frac{7}{2}$

(10.8.14)

The fourth step is to go back and repeat the first step, looking for another possible increase of the objective function. We do this as many times as possible, that is, until all the right-hand entries in the z-row are negative, signaling that no further increase is possible. In the present example, this already occurs in (10.8.14), so we are done.

The answer can now be read from the constant column of the final tableau. In (10.8.14) we see that the objective function is maximized to a value of $2/3$ for the solution vector $x_2 = 1/3$, $x_4 = 9$, $x_1 = x_3 = 0$.

Now look back over the procedure that led from (10.8.10) to (10.8.14). You will find that it could be summarized entirely in tableau format as a series of prescribed elementary matrix operations:

- Locate the pivot element and save it.
- Save the whole pivot column.
- Replace each row, except the pivot row, by that linear combination of itself and the pivot row which makes its pivot-column entry zero.
- Divide the pivot row by the negative of the pivot.
- Replace the pivot element by the reciprocal of its saved value.
- Replace the rest of the pivot column by its saved values divided by the saved pivot element.

This is the sequence of operations actually performed by a linear programming routine, such as the one that we will presently give.

You should now be able to solve almost any linear programming problem that starts in restricted normal form. The only special case that might stump you is if an entry in the constant column turns out to be zero at some stage, so that a left-hand variable is zero at the same time as all the right-hand variables are zero. This is called a *degenerate feasible vector*. To proceed, you may need to exchange the degenerate left-hand variable for one of the right-hand variables, perhaps even making several such exchanges.

Writing the General Problem in Restricted Normal Form

Here is a pleasant surprise. There exist a couple of clever tricks that render trivial the task of translating a general linear programming problem into restricted normal form!

First, we need to get rid of the inequalities of the form (10.8.3) or (10.8.4), for example, the first three constraints in (10.8.7). We do this by adding to the problem so-called *slack variables* which, when their nonnegativity is required, convert the inequalities to equalities. We will denote slack variables as y_i. There will be $m_1 + m_2$ of them. Once they are introduced, you treat them on an equal footing with the original variables x_i; then, at the very end, you simply ignore them.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 116 of 181

For example, introducing slack variables leaves (10.8.6) unchanged but turns (10.8.7) into

$$\begin{aligned} x_1 + 2x_3 + y_1 &= 740 \\ 2x_2 - 7x_4 + y_2 &= 0 \\ x_2 - x_3 + 2x_4 - y_3 &= \tfrac{1}{2} \\ x_1 + x_2 + x_3 + x_4 &= 9 \end{aligned} \tag{10.8.15}$$

(Notice how the sign of the coefficient of the slack variable is determined by which sense of inequality it is replacing.)

Second, we need to insure that there is a set of M left-hand vectors, so that we can set up a starting tableau in restricted normal form. (In other words, we need to find a "feasible basic starting vector.") The trick is again to invent new variables! There are M of these, and they are called *artificial variables*; we denote them by z_i. You put exactly one artificial variable into each constraint equation on the following model for the example (10.8.15):

$$\begin{aligned} z_1 &= 740 - x_1 - 2x_3 - y_1 \\ z_2 &= -2x_2 + 7x_4 - y_2 \\ z_3 &= \tfrac{1}{2} - x_2 + x_3 - 2x_4 + y_3 \\ z_4 &= 9 - x_1 - x_2 - x_3 - x_4 \end{aligned} \tag{10.8.16}$$

Our example is now in restricted normal form.

Now you may object that (10.8.16) is not the same problem as (10.8.15) or (10.8.7) *unless all the z_i's are zero*. Right you are! There is some subtlety here! We must proceed to solve our problem in two phases. First phase: We replace our objective function (10.8.6) by a so-called *auxiliary objective function*

$$z' \equiv -z_1 - z_2 - z_3 - z_4 = -(749\tfrac{1}{2} - 2x_1 - 4x_2 - 2x_3 + 4x_4 - y_1 - y_2 + y_3) \tag{10.8.17}$$

(where the last equality follows from using 10.8.16). We now perform the simplex method on the auxiliary objective function (10.8.17) with the constraints (10.8.16). Obviously the auxiliary objective function will be maximized for nonnegative z_i's if all the z_i's are zero. We therefore expect the simplex method in this first phase to produce a set of left-hand variables drawn from the x_i's and y_i's only, with all the z_i's being right-hand variables. Aha! We then cross out the z_i's, leaving a problem involving only x_i's and y_i's in restricted normal form. In other words, the first phase produces an initial feasible basic vector. Second phase: Solve the problem produced by the first phase, using the original objective function, not the auxiliary.

And what if the first phase *doesn't* produce zero values for all the z_i's? That signals that there is *no* initial feasible basic vector, i.e., that the constraints given to us are inconsistent among themselves. Report that fact, and you are done.

Here is how to translate into tableau format the information needed for both the first and second phases of the overall method. As before, the underlying problem

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 117 of 181

to be solved is as posed in equations (10.8.6)–(10.8.7).

		x_1	x_2	x_3	x_4	y_1	y_2	y_3
z	0	1	1	3	$-\frac{1}{2}$	0	0	0
z_1	740	-1	0	-2	0	-1	0	0
z_2	0	0	-2	0	7	0	-1	0
z_3	$\frac{1}{2}$	0	-1	1	-2	0	0	1
z_4	9	-1	-1	-1	-1	0	0	0
z'	$-749\frac{1}{2}$	2	4	2	-4	1	1	-1

(10.8.18)

This is not as daunting as it may, at first sight, appear. The table entries inside the box of double lines are no more than the coefficients of the original problem (10.8.6)–(10.8.7) organized into a tabular form. In fact, these entries, along with the values of N, M, m_1, m_2, and m_3, are the only input that is needed by the simplex method routine below. The columns under the slack variables y_i simply record whether each of the M constraints is of the form $\leq$, $\geq$, or $=$; this is redundant information with the values m_1, m_2, m_3, as long as we are sure to enter the rows of the tableau in the correct respective order. The coefficients of the auxiliary objective function (bottom row) are just the negatives of the column sums of the rows above, so these are easily calculated automatically.

The output from a simplex routine will be (i) a flag telling whether a finite solution, no solution, or an unbounded solution was found, and (ii) an updated tableau. The output tableau that derives from (10.8.18), given to two significant figures, is

		x_1	y_2	y_3	$\cdots$
z	17.03	$-.95$	$-.05$	-1.05	$\cdots$
x_2	3.33	$-.35$	$-.15$	$.35$	$\cdots$
x_3	4.73	$-.55$	$.05$	$-.45$	$\cdots$
x_4	$.95$	$-.10$	$.10$	$.10$	$\cdots$
y_1	730.55	$.10$	$-.10$	$.90$	$\cdots$

(10.8.19)

A little counting of the x_i's and y_i's will convince you that there are $M + 1$ rows (including the z-row) in both the input and the output tableaux, but that only $N + 1 - m_3$ columns of the output tableau (including the constant column) contain any useful information, the other columns belonging to now-discarded artificial variables. In the output, the first numerical column contains the solution vector, along with the maximum value of the objective function. Where a slack variable (y_i) appears on the left, the corresponding value is the amount by which its inequality is safely satisfied. Variables that are not left-hand variables in the output tableau have zero values. Slack variables with zero values represent constraints that are satisfied as equalities.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 118 of 181

Routine Implementing the Simplex Method

The following routine is based algorithmically on the implementation of Kuenzi, Tzschach, and Zehnder [4]. Aside from input values of M, N, m_1, m_2, m_3, the principal input to the routine is a two-dimensional array a containing the portion of the tableau (10.8.18) that is contained between the double lines. This input occupies the $M+1$ rows and $N+1$ columns of a[1..m+1][1..n+1]. Note, however, that reference is made internally to row $M+2$ of a (used for the auxiliary objective function, just as in 10.8.18). Therefore the variable declared as float **a, must point to allocated memory allowing references in the subrange

$$\texttt{a}[i][k], \quad i = 1 \ldots \texttt{m+2},\ k = 1 \ldots \texttt{n+1} \tag{10.8.20}$$

You will suffer endless agonies if you fail to understand this simple point. Also do not neglect to order the rows of a in the same order as equations (10.8.1), (10.8.3), (10.8.4), and (10.8.5), that is, objective function, $\leq$-constraints, $\geq$-constraints, $=$-constraints.

On output, the tableau a is indexed by two returned arrays of integers. iposv[j] contains, for j$= 1 \ldots M$, the number i whose original variable x_i is now represented by row j+1 of a. These are thus the left-hand variables in the solution. (The first row of a is of course the z-row.) A value $i > N$ indicates that the variable is a y_i rather than an x_i, $x_{N+j} \equiv y_j$. Likewise, izrov[j] contains, for j$= 1 \ldots N$, the number i whose original variable x_i is now a right-hand variable, represented by column j+1 of a. These variables are all zero in the solution. The meaning of $i > N$ is the same as above, except that $i > N + m_1 + m_2$ denotes an artificial or slack variable which was used only internally and should now be entirely ignored.

The flag icase is set to zero if a finite solution is found, $+1$ if the objective function is unbounded, -1 if no solution satisfies the given constraints.

The routine treats the case of degenerate feasible vectors, so don't worry about them. You may also wish to admire the fact that the routine does not require storage for the columns of the tableau (10.8.18) that are to the right of the double line; it keeps track of slack variables by more efficient bookkeeping.

Please note that, as given, the routine is only "semi-sophisticated" in its tests for convergence. While the routine properly implements tests for inequality with zero as tests against some small parameter EPS, it does not adjust this parameter to reflect the scale of the input data. This is adequate for many problems, where the input data do not differ from unity by too many orders of magnitude. If, however, you encounter endless cycling, then you should modify EPS in the routines simplx and simp2. Permuting your variables can also help. Finally, consult [5].

```
#include "nrutil.h"
#define EPS 1.0e-6
Here EPS is the absolute precision, which should be adjusted to the scale of your variables.
#define FREEALL free_ivector(l3,1,m);free_ivector(l1,1,n+1);

void simplx(float **a, int m, int n, int m1, int m2, int m3, int *icase,
    int izrov[], int iposv[])
Simplex method for linear programming. Input parameters a, m, n, mp, np, m1, m2, and m3,
and output parameters a, icase, izrov, and iposv are described above.
{
    void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
```

```
    float *bmax);
void simp2(float **a, int m, int n, int *ip, int kp);
void simp3(float **a, int i1, int k1, int ip, int kp);
int i,ip,is,k,kh,kp,nl1;
int *l1,*l3;
float q1,bmax;

if (m != (m1+m2+m3)) nrerror("Bad input constraint counts in simplx");
l1=ivector(1,n+1);
l3=ivector(1,m);
nl1=n;
for (k=1;k<=n;k++) l1[k]=izrov[k]=k;
```

Initialize index list of columns admissible for exchange, and make all variables initially right-hand.

```
for (i=1;i<=m;i++) {
    if (a[i+1][1] < 0.0) nrerror("Bad input tableau in simplx");
```

Constants b_i must be nonnegative.

```
    iposv[i]=n+i;
```

Initial left-hand variables. m1 type constraints are represented by having their slack variable initially left-hand, with no artificial variable. m2 type constraints have their slack variable initially left-hand, with a minus sign, and their artificial variable handled implicitly during their first exchange. m3 type constraints have their artificial variable initially left-hand.

```
}
if (m2+m3) {                                    Origin is not a feasible starting so-
    for (i=1;i<=m2;i++) l3[i]=1;                    lution: we must do phase one.
```

Initialize list of m2 constraints whose slack variables have never been exchanged out of the initial basis.

```
    for (k=1;k<=(n+1);k++) {                    Compute the auxiliary objective func-
        q1=0.0;                                     tion.
        for (i=m1+1;i<=m;i++) q1 += a[i+1][k];
        a[m+2][k] = -q1;
    }
    for (;;) {
        simp1(a,m+1,l1,nl1,0,&kp,&bmax);        Find max. coeff. of auxiliary objec-
        if (bmax <= EPS && a[m+2][1] < -EPS) {      tive fn.
            *icase = -1;
```

Auxiliary objective function is still negative and can't be improved, hence no feasible solution exists.

```
            FREEALL return;
        } else if (bmax <= EPS && a[m+2][1] <= EPS) {
```

Auxiliary objective function is zero and can't be improved; we have a feasible starting vector. Clean out the artificial variables corresponding to any remaining equality constraints by goto one and then move on to phase two.

```
            for (ip=m1+m2+1;ip<=m;ip++) {
                if (iposv[ip] == (ip+n)) {      Found an artificial variable for an
                    simp1(a,ip,l1,nl1,1,&kp,&bmax);     equality constraint.
                    if (bmax > EPS)             Exchange with column correspond-
                        goto one;                   ing to maximum pivot element
                }                                   in row.
            }
            for (i=m1+1;i<=m1+m2;i++)           Change sign of row for any m2 con-
                if (l3[i-m1] == 1)                  straints still present from the ini-
                    for (k=1;k<=n+1;k++)            tial basis.
                        a[i+1][k] = -a[i+1][k];
            break;                              Go to phase two.
        }
        simp2(a,m,n,&ip,kp);                    Locate a pivot element (phase one).
        if (ip == 0) {                          Maximum of auxiliary objective func-
            *icase = -1;                            tion is unbounded, so no feasi-
            FREEALL return;                         ble solution exists.
        }
one:    simp3(a,m+1,n,ip,kp);
```

Exchange a left- and a right-hand variable (phase one), then update lists.

```
            if (iposv[ip] >= (n+m1+m2+1)) {        Exchanged out an artificial variable
                for (k=1;k<=nl1;k++)                   for an equality constraint. Make
                    if (l1[k] == kp) break;            sure it stays out by removing it
                --nl1;                                 from the l1 list.
                for (is=k;is<=nl1;is++) l1[is]=l1[is+1];
            } else {
                kh=iposv[ip]-m1-n;
                if (kh >= 1 && l3[kh]) {           Exchanged out an m2 type constraint
                    l3[kh]=0;                          for the first time. Correct the
                    ++a[m+2][kp+1];                    pivot column for the minus sign
                    for (i=1;i<=m+2;i++)               and the implicit artificial vari-
                        a[i][kp+1] = -a[i][kp+1];      able.
                }
            }
            is=izrov[kp];                          Update lists of left- and right-hand
            izrov[kp]=iposv[ip];                       variables.
            iposv[ip]=is;
        }                                          Still in phase one, go back to the
    }                                                  for(;;).
    End of phase one code for finding an initial feasible solution. Now, in phase two, optimize
    it.
    for (;;) {
        simp1(a,0,l1,nl1,0,&kp,&bmax);             Test the z-row for doneness.
        if (bmax <= EPS) {                         Done. Solution found. Return with
            *icase=0;                                  the good news.
            FREEALL return;
        }
        simp2(a,m,n,&ip,kp);                       Locate a pivot element (phase two).
        if (ip == 0) {                             Objective function is unbounded. Re-
            *icase=1;                                  port and return.
            FREEALL return;
        }
        simp3(a,m,n,ip,kp);                        Exchange a left- and a right-hand
        is=izrov[kp];                                  variable (phase two),
        izrov[kp]=iposv[ip];
        iposv[ip]=is;
    }                                              and return for another iteration.
}
```

The preceding routine makes use of the following utility functions.

```
#include <math.h>

void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
    float *bmax)
Determines the maximum of those elements whose index is contained in the supplied list ll,
either with or without taking the absolute value, as flagged by iabf.
{
    int k;
    float test;

    if (nll <= 0)                No eligible columns.
        *bmax=0.0;
    else {
        *kp=ll[1];
        *bmax=a[mm+1][*kp+1];
        for (k=2;k<=nll;k++) {
            if (iabf == 0)
                test=a[mm+1][ll[k]+1]-(*bmax);
```

```
            else
                test=fabs(a[mm+1][ll[k]+1])-fabs(*bmax);
            if (test > 0.0) {
                *bmax=a[mm+1][ll[k]+1];
                *kp=ll[k];
            }
        }
    }
}
```

```
#define EPS 1.0e-6

void simp2(float **a, int m, int n, int *ip, int kp)
Locate a pivot element, taking degeneracy into account.
{
    int k,i;
    float qp,q0,q,q1;

    *ip=0;
    for (i=1;i<=m;i++)
        if (a[i+1][kp+1] < -EPS) break;        Any possible pivots?
    if (i>m) return;
    q1 = -a[i+1][1]/a[i+1][kp+1];
    *ip=i;
    for (i=*ip+1;i<=m;i++) {
        if (a[i+1][kp+1] < -EPS) {
            q = -a[i+1][1]/a[i+1][kp+1];
            if (q < q1) {
                *ip=i;
                q1=q;
            } else if (q == q1) {              We have a degeneracy.
                for (k=1;k<=n;k++) {
                    qp = -a[*ip+1][k+1]/a[*ip+1][kp+1];
                    q0 = -a[i+1][k+1]/a[i+1][kp+1];
                    if (q0 != qp) break;
                }
                if (q0 < qp) *ip=i;
            }
        }
    }
}
```

```
void simp3(float **a, int i1, int k1, int ip, int kp)
Matrix operations to exchange a left-hand and right-hand variable (see text).
{
    int kk,ii;
    float piv;

    piv=1.0/a[ip+1][kp+1];
    for (ii=1;ii<=i1+1;ii++)
        if (ii-1 != ip) {
            a[ii][kp+1] *= piv;
            for (kk=1;kk<=k1+1;kk++)
                if (kk-1 != kp)
                    a[ii][kk] -= a[ip+1][kk]*a[ii][kp+1];
        }
```

```
    for (kk=1;kk<=k1+1;kk++)
        if (kk-1 != kp) a[ip+1][kk] *= -piv;
    a[ip+1][kp+1]=piv;
}
```

Other Topics Briefly Mentioned

Every linear programming problem in normal form with N variables and M constraints has a corresponding *dual* problem with M variables and N constraints. The tableau of the dual problem is, in essence, the transpose of the tableau of the original (sometimes called *primal*) problem. It is possible to go from a solution of the dual to a solution of the primal. This can occasionally be computationally useful, but generally it is no big deal.

The *revised simplex method* is exactly equivalent to the simplex method in its choice of which left-hand and right-hand variables are exchanged. Its computational effort is not significantly less than that of the simplex method. It does differ in the organization of its storage, requiring only a matrix of size $M \times M$, rather than $M \times N$, in its intermediate stages. If you have a lot of constraints, and memory size is one of them, then you should look into it.

The *primal-dual algorithm* and the *composite simplex algorithm* are two different methods for avoiding the two phases of the usual simplex method: Progress is made simultaneously towards finding a feasible solution and finding an optimal solution. There seems to be no clearcut evidence that these methods are superior to the usual method by any factor substantially larger than the "tender-loving-care factor" (which reflects the programming effort of the proponents).

Problems where the objective function and/or one or more of the constraints are replaced by expressions nonlinear in the variables are called *nonlinear programming problems*. The literature on such problems is vast, but outside our scope. The special case of quadratic expressions is called *quadratic programming*. Optimization problems where the variables take on only integer values are called *integer programming* problems, a special case of *discrete optimization* generally. The next section looks at a particular kind of discrete optimization problem.

CITED REFERENCES AND FURTHER READING:

Bland, R.G. 1981, *Scientific American*, vol. 244 (June), pp. 126–144. [1]

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press). [2]

Kolata, G. 1982, *Science*, vol. 217, p. 39. [3]

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), Chapters 7–8.

Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders).

Gass, S.T. 1969, *Linear Programming*, 3rd ed. (New York: McGraw-Hill).

Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley).

Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience).

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press). [4]

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 123 of 181

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.10.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [5]

10.9 Simulated Annealing Methods

The *method of simulated annealing* [1,2] is a technique that has attracted significant attention as suitable for optimization problems of large scale, especially ones where a desired global extremum is hidden among many, poorer, local extrema. For practical purposes, simulated annealing has effectively "solved" the famous *traveling salesman problem* of finding the shortest cyclical itinerary for a traveling salesman who must visit each of N cities in turn. (Other practical methods have also been found.) The method has also been used successfully for designing complex integrated circuits: The arrangement of several hundred thousand circuit elements on a tiny silicon substrate is optimized so as to minimize interference among their connecting wires [3,4]. Surprisingly, the implementation of the algorithm is relatively simple.

Notice that the two applications cited are both examples of *combinatorial minimization*. There is an objective function to be minimized, as usual; but the space over which that function is defined is not simply the N-dimensional space of N continuously variable parameters. Rather, it is a discrete, but very large, configuration space, like the set of possible orders of cities, or the set of possible allocations of silicon "real estate" blocks to circuit elements. The number of elements in the configuration space is factorially large, so that they cannot be explored exhaustively. Furthermore, since the set is discrete, we are deprived of any notion of "continuing downhill in a favorable direction." The concept of "direction" may not have any meaning in the configuration space.

Below, we will also discuss how to use simulated annealing methods for spaces with continuous control parameters, like those of §§10.4–10.7. This application is actually more complicated than the combinatorial one, since the familiar problem of "long, narrow valleys" again asserts itself. Simulated annealing, as we will see, tries "random" steps; but in a long, narrow valley, almost all random steps are uphill! Some additional finesse is therefore required.

At the heart of the method of simulated annealing is an analogy with thermodynamics, specifically with the way that liquids freeze and crystallize, or metals cool and anneal. At high temperatures, the molecules of a liquid move freely with respect to one another. If the liquid is cooled slowly, thermal mobility is lost. The atoms are often able to line themselves up and form a pure crystal that is completely ordered over a distance up to billions of times the size of an individual atom in all directions. This crystal is the state of minimum energy for this system. The amazing fact is that, for slowly cooled systems, nature is able to find this minimum energy state. In fact, if a liquid metal is cooled quickly or "quenched," it does not reach this state but rather ends up in a polycrystalline or amorphous state having somewhat higher energy.

So the essence of the process is *slow* cooling, allowing ample time for redistribution of the atoms as they lose mobility. This is the technical definition of *annealing*, and it is essential for ensuring that a low energy state will be achieved.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 124 of 181

Although the analogy is not perfect, there is a sense in which all of the minimization algorithms thus far in this chapter correspond to rapid cooling or quenching. In all cases, we have gone greedily for the quick, nearby solution: From the starting point, go immediately downhill as far as you can go. This, as often remarked above, leads to a local, but not necessarily a global, minimum. Nature's own minimization algorithm is based on quite a different procedure. The so-called Boltzmann probability distribution,

$$\text{Prob}\,(E) \sim \exp(-E/kT) \tag{10.9.1}$$

expresses the idea that a system in thermal equilibrium at temperature T has its energy probabilistically distributed among all different energy states E. Even at low temperature, there is a chance, albeit very small, of a system being in a high energy state. Therefore, there is a corresponding chance for the system to get out of a local energy minimum in favor of finding a better, more global, one. The quantity k (Boltzmann's constant) is a constant of nature that relates temperature to energy. In other words, the system sometimes goes *uphill* as well as downhill; but the lower the temperature, the less likely is any significant uphill excursion.

In 1953, Metropolis and coworkers [5] first incorporated these kinds of principles into numerical calculations. Offered a succession of options, a simulated thermodynamic system was assumed to change its configuration from energy E_1 to energy E_2 with probability $p = \exp[-(E_2 - E_1)/kT]$. Notice that if $E_2 < E_1$, this probability is greater than unity; in such cases the change is arbitrarily assigned a probability $p = 1$, i.e., the system *always* took such an option. This general scheme, of always taking a downhill step while *sometimes* taking an uphill step, has come to be known as the Metropolis algorithm.

To make use of the Metropolis algorithm for other than thermodynamic systems, one must provide the following elements:

1. A description of possible system configurations.

2. A generator of random changes in the configuration; these changes are the "options" presented to the system.

3. An objective function E (analog of energy) whose minimization is the goal of the procedure.

4. A control parameter T (analog of temperature) and an *annealing schedule* which tells how it is lowered from high to low values, e.g., after how many random changes in configuration is each downward step in T taken, and how large is that step. The meaning of "high" and "low" in this context, and the assignment of a schedule, may require physical insight and/or trial-and-error experiments.

Combinatorial Minimization: The Traveling Salesman

A concrete illustration is provided by the traveling salesman problem. The proverbial seller visits N cities with given positions (x_i, y_i), returning finally to his or her city of origin. Each city is to be visited only once, and the route is to be made as short as possible. This problem belongs to a class known as *NP-complete* problems, whose computation time for an *exact* solution increases with N as $\exp(\text{const.} \times N)$, becoming rapidly prohibitive in cost as N increases. The traveling salesman problem also belongs to a class of minimization problems for which the objective function E

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 125 of 181

has many local minima. In practical cases, it is often enough to be able to choose from these a minimum which, even if not absolute, cannot be significantly improved upon. The annealing method manages to achieve this, while limiting its calculations to scale as a small power of N.

As a problem in simulated annealing, the traveling salesman problem is handled as follows:

1. *Configuration.* The cities are numbered $i = 1 \ldots N$ and each has coordinates (x_i, y_i). A configuration is a permutation of the number $1 \ldots N$, interpreted as the order in which the cities are visited.

2. *Rearrangements.* An efficient set of moves has been suggested by Lin [6]. The moves consist of two types: (a) A section of path is removed and then replaced with the same cities running in the opposite order; or (b) a section of path is removed and then replaced in between two cities on another, randomly chosen, part of the path.

3. *Objective Function.* In the simplest form of the problem, E is taken just as the total length of journey,

$$E = L \equiv \sum_{i=1}^{N} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \tag{10.9.2}$$

with the convention that point $N + 1$ is identified with point 1. To illustrate the flexibility of the method, however, we can add the following additional wrinkle: Suppose that the salesman has an irrational fear of flying over the Mississippi River. In that case, we would assign each city a parameter μ_i, equal to $+1$ if it is east of the Mississippi, -1 if it is west, and take the objective function to be

$$E = \sum_{i=1}^{N} \left[\sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} + \lambda(\mu_i - \mu_{i+1})^2 \right] \tag{10.9.3}$$

A penalty 4λ is thereby assigned to any river crossing. The algorithm now finds the shortest path that avoids crossings. The relative importance that it assigns to length of path versus river crossings is determined by our choice of λ. Figure 10.9.1 shows the results obtained. Clearly, this technique can be generalized to include many conflicting goals in the minimization.

4. *Annealing schedule.* This requires experimentation. We first generate some random rearrangements, and use them to determine the range of values of ΔE that will be encountered from move to move. Choosing a starting value for the parameter T which is considerably larger than the largest ΔE normally encountered, we proceed downward in multiplicative steps each amounting to a 10 percent decrease in T. We hold each new value of T constant for, say, $100N$ reconfigurations, or for $10N$ successful reconfigurations, whichever comes first. When efforts to reduce E further become sufficiently discouraging, we stop.

The following traveling salesman program, using the Metropolis algorithm, illustrates the main aspects of the simulated annealing technique for combinatorial problems.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 126 of 181

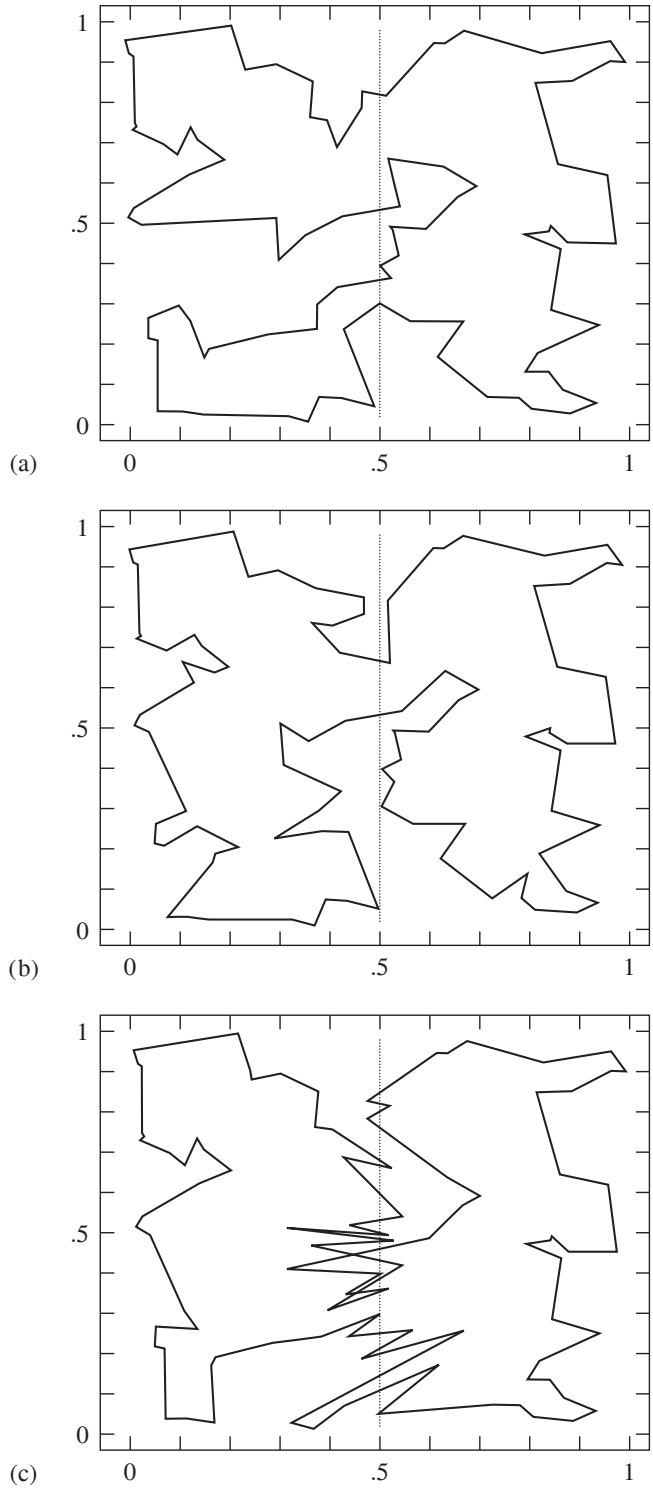

Figure 10.9.1. Traveling salesman problem solved by simulated annealing. The (nearly) shortest path among 100 randomly positioned cities is shown in (a). The dotted line is a river, but there is no penalty in crossing. In (b) the river-crossing penalty is made large, and the solution restricts itself to the minimum number of crossings, two. In (c) the penalty has been made negative: the salesman is actually a smuggler who crosses the river on the flimsiest excuse!

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 127 of 181

```
#include <stdio.h>
#include <math.h>
#define TFACTR 0.9                Annealing schedule: reduce t by this factor on each step.
#define ALEN(a,b,c,d) sqrt(((b)-(a))*((b)-(a))+((d)-(c))*((d)-(c)))

void anneal(float x[], float y[], int iorder[], int ncity)
This algorithm finds the shortest round-trip path to ncity cities whose coordinates are in the
arrays x[1..ncity],y[1..ncity]. The array iorder[1..ncity] specifies the order in
which the cities are visited. On input, the elements of iorder may be set to any permutation
of the numbers 1 to ncity. This routine will return the best alternative path it can find.
{
    int irbit1(unsigned long *iseed);
    int metrop(float de, float t);
    float ran3(long *idum);
    float revcst(float x[], float y[], int iorder[], int ncity, int n[]);
    void reverse(int iorder[], int ncity, int n[]);
    float trncst(float x[], float y[], int iorder[], int ncity, int n[]);
    void trnspt(int iorder[], int ncity, int n[]);
    int ans,nover,nlimit,i1,i2;
    int i,j,k,nsucc,nn,idec;
    static int n[7];
    long idum;
    unsigned long iseed;
    float path,de,t;

    nover=100*ncity;              Maximum number of paths tried at any temperature.
    nlimit=10*ncity;              Maximum number of successful path changes before con-
    path=0.0;                         tinuing.
    t=0.5;
    for (i=1;i<ncity;i++) {       Calculate initial path length.
        i1=iorder[i];
        i2=iorder[i+1];
        path += ALEN(x[i1],x[i2],y[i1],y[i2]);
    }
    i1=iorder[ncity];             Close the loop by tying path ends together.
    i2=iorder[1];
    path += ALEN(x[i1],x[i2],y[i1],y[i2]);
    idum = -1;
    iseed=111;
    for (j=1;j<=100;j++) {        Try up to 100 temperature steps.
        nsucc=0;
        for (k=1;k<=nover;k++) {
            do {
                n[1]=1+(int) (ncity*ran3(&idum));        Choose beginning of segment
                n[2]=1+(int) ((ncity-1)*ran3(&idum));    ..and. end of segment.
                if (n[2] >= n[1]) ++n[2];
                nn=1+((n[1]-n[2]+ncity-1) % ncity);      nn is the number of cities
            } while (nn<3);                                  not on the segment.
            idec=irbit1(&iseed);
            Decide whether to do a segment reversal or transport.
            if (idec == 0) {                         Do a transport.
                n[3]=n[2]+(int) (abs(nn-2)*ran3(&idum))+1;
                n[3]=1+((n[3]-1) % ncity);
                Transport to a location not on the path.
                de=trncst(x,y,iorder,ncity,n);       Calculate cost.
                ans=metrop(de,t);                    Consult the oracle.
                if (ans) {
                    ++nsucc;
                    path += de;
                    trnspt(iorder,ncity,n);          Carry out the transport.
                }
            } else {                                 Do a path reversal.
                de=revcst(x,y,iorder,ncity,n);       Calculate cost.
                ans=metrop(de,t);                    Consult the oracle.
```

```
                if (ans) {
                    ++nsucc;
                    path += de;
                    reverse(iorder,ncity,n);        Carry out the reversal.
                }
            }
            if (nsucc >= nlimit) break;             Finish early if we have enough suc-
        }                                               cessful changes.
        printf("\n %s %10.6f %s %12.6f \n","T =",t,
            "     Path Length =",path);
        printf("Successful Moves: %6d\n",nsucc);
        t *= TFACTR;                                Annealing schedule.
        if (nsucc == 0) return;                     If no success, we are done.
    }
}
```

```
#include <math.h>
#define ALEN(a,b,c,d) sqrt(((b)-(a))*((b)-(a))+((d)-(c))*((d)-(c)))

float revcst(float x[], float y[], int iorder[], int ncity, int n[])
```

This function returns the value of the cost function for a proposed path reversal. `ncity` is the number of cities, and arrays `x[1..ncity],y[1..ncity]` give the coordinates of these cities. `iorder[1..ncity]` holds the present itinerary. The first two values `n[1]` and `n[2]` of array `n` give the starting and ending cities along the path segment which is to be reversed. On output, `de` is the cost of making the reversal. The actual reversal is not performed by this routine.

```
{
    float xx[5],yy[5],de;
    int j,ii;

    n[3]=1 + ((n[1]+ncity-2) % ncity);          Find the city before n[1] ..
    n[4]=1 + (n[2] % ncity);                    .. and the city after n[2].
    for (j=1;j<=4;j++) {
        ii=iorder[n[j]];                        Find coordinates for the four cities in-
        xx[j]=x[ii];                                volved.
        yy[j]=y[ii];
    }
    de = -ALEN(xx[1],xx[3],yy[1],yy[3]);        Calculate cost of disconnecting the seg-
    de -= ALEN(xx[2],xx[4],yy[2],yy[4]);            ment at both ends and reconnecting
    de += ALEN(xx[1],xx[4],yy[1],yy[4]);            in the opposite order.
    de += ALEN(xx[2],xx[3],yy[2],yy[3]);
    return de;
}
```

```
void reverse(int iorder[], int ncity, int n[])
```

This routine performs a path segment reversal. `iorder[1..ncity]` is an input array giving the present itinerary. The vector `n` has as its first four elements the first and last cities `n[1],n[2]` of the path segment to be reversed, and the two cities `n[3]` and `n[4]` that immediately precede and follow this segment. `n[3]` and `n[4]` are found by function `revcst`. On output, `iorder[1..ncity]` contains the segment from `n[1]` to `n[2]` in reversed order.

```
{
    int nn,j,k,l,itmp;

    nn=(1+((n[2]-n[1]+ncity) % ncity))/2;       This many cities must be swapped to
    for (j=1;j<=nn;j++) {                           effect the reversal.
        k=1 + ((n[1]+j-2) % ncity);             Start at the ends of the segment and
        l=1 + ((n[2]-j+ncity) % ncity);             swap pairs of cities, moving toward
        itmp=iorder[k];                             the center.
        iorder[k]=iorder[l];
        iorder[l]=itmp;
    }
}
```

```
#include <math.h>
#define ALEN(a,b,c,d) sqrt(((b)-(a))*((b)-(a))+((d)-(c))*((d)-(c)))

float trncst(float x[], float y[], int iorder[], int ncity, int n[])
```

This routine returns the value of the cost function for a proposed path segment transport. `ncity` is the number of cities, and arrays `x[1..ncity]` and `y[1..ncity]` give the city coordinates. `iorder[1..ncity]` is an array giving the present itinerary. The first three elements of array `n` give the starting and ending cities of the path to be transported, and the point among the remaining cities after which it is to be inserted. On output, `de` is the cost of the change. The actual transport is not performed by this routine.

```
{
    float xx[7],yy[7],de;
    int j,ii;

    n[4]=1 + (n[3] % ncity);                    Find the city following n[3]..
    n[5]=1 + ((n[1]+ncity-2) % ncity);          ..and the one preceding n[1]..
    n[6]=1 + (n[2] % ncity);                    ..and the one following n[2].
    for (j=1;j<=6;j++) {
        ii=iorder[n[j]];                        Determine coordinates for the six cities
        xx[j]=x[ii];                                involved.
        yy[j]=y[ii];
    }
    de = -ALEN(xx[2],xx[6],yy[2],yy[6]);        Calculate the cost of disconnecting the
    de -= ALEN(xx[1],xx[5],yy[1],yy[5]);            path segment from n[1] to n[2],
    de -= ALEN(xx[3],xx[4],yy[3],yy[4]);            opening a space between n[3] and
    de += ALEN(xx[1],xx[3],yy[1],yy[3]);            n[4], connecting the segment in the
    de += ALEN(xx[2],xx[4],yy[2],yy[4]);            space, and connecting n[5] to n[6].
    de += ALEN(xx[5],xx[6],yy[5],yy[6]);
    return de;
}
```

```
#include "nrutil.h"

void trnspt(int iorder[], int ncity, int n[])
```

This routine does the actual path transport, once `metrop` has approved. `iorder[1..ncity]` is an input array giving the present itinerary. The array `n` has as its six elements the beginning `n[1]` and end `n[2]` of the path to be transported, the adjacent cities `n[3]` and `n[4]` between which the path is to be placed, and the cities `n[5]` and `n[6]` that precede and follow the path. `n[4]`, `n[5]`, and `n[6]` are calculated by function `trncst`. On output, `iorder` is modified to reflect the movement of the path segment.

```
{
    int m1,m2,m3,nn,j,jj,*jorder;

    jorder=ivector(1,ncity);
    m1=1 + ((n[2]-n[1]+ncity) % ncity);         Find number of cities from n[1] to n[2]
    m2=1 + ((n[5]-n[4]+ncity) % ncity);         ...and the number from n[4] to n[5]
    m3=1 + ((n[3]-n[6]+ncity) % ncity);         ...and the number from n[6] to n[3].
    nn=1;
    for (j=1;j<=m1;j++) {
        jj=1 + ((j+n[1]-2) % ncity);            Copy the chosen segment.
        jorder[nn++]=iorder[jj];
    }
    for (j=1;j<=m2;j++) {                       Then copy the segment from n[4] to
        jj=1+((j+n[4]-2) % ncity);                  n[5].
        jorder[nn++]=iorder[jj];
    }
    for (j=1;j<=m3;j++) {                       Finally, the segment from n[6] to n[3].
        jj=1 + ((j+n[6]-2) % ncity);
        jorder[nn++]=iorder[jj];
    }
    for (j=1;j<=ncity;j++)                      Copy jorder back into iorder.
        iorder[j]=jorder[j];
```

```
    free_ivector(jorder,1,ncity);
}
```

```
#include <math.h>

int metrop(float de, float t)
```

Metropolis algorithm. `metrop` returns a boolean variable that issues a verdict on whether to accept a reconfiguration that leads to a change `de` in the objective function `e`. If `de<0`, `metrop` = 1 (true), while if `de>0`, `metrop` is only true with probability `exp(-de/t)`, where `t` is a temperature determined by the annealing schedule.

```
{
    float ran3(long *idum);
    static long gljdum=1;

    return de < 0.0 || ran3(&gljdum) < exp(-de/t);
}
```

Continuous Minimization by Simulated Annealing

The basic ideas of simulated annealing are also applicable to optimization problems with continuous N-dimensional control spaces, e.g., finding the (ideally, global) minimum of some function $f(\mathbf{x})$, in the presence of many local minima, where $\mathbf{x}$ is an N-dimensional vector. The four elements required by the Metropolis procedure are now as follows: The value of f is the objective function. The system state is the point $\mathbf{x}$. The control parameter T is, as before, something like a temperature, with an annealing schedule by which it is gradually reduced. And there must be a generator of random changes in the configuration, that is, a procedure for taking a random step from $\mathbf{x}$ to $\mathbf{x} + \Delta\mathbf{x}$.

The last of these elements is the most problematical. The literature to date [7-10] describes several different schemes for choosing $\Delta\mathbf{x}$, none of which, in our view, inspire complete confidence. The problem is one of efficiency: A generator of random changes is inefficient if, *when local downhill moves exist*, it nevertheless almost always proposes an uphill move. A good generator, we think, should not become inefficient in narrow valleys; nor should it become more and more inefficient as convergence to a minimum is approached. Except possibly for [7], all of the schemes that we have seen are inefficient in one or both of these situations.

Our own way of doing simulated annealing minimization on continuous control spaces is to use a modification of the downhill simplex method (§10.4). This amounts to replacing the single point $\mathbf{x}$ as a description of the system state by a simplex of $N + 1$ points. The "moves" are the same as described in §10.4, namely reflections, expansions, and contractions of the simplex. The implementation of the Metropolis procedure is slightly subtle: We *add* a positive, logarithmically distributed random variable, proportional to the temperature T, to the stored function value associated with every vertex of the simplex, and we *subtract* a similar random variable from the function value of every new point that is tried as a replacement point. Like the ordinary Metropolis procedure, this method always accepts a true downhill step, but

sometimes accepts an uphill one. In the limit $T \to 0$, this algorithm reduces exactly to the downhill simplex method and converges to a local minimum.

At a finite value of T, the simplex expands to a scale that approximates the size of the region that can be reached at this temperature, and then executes a stochastic, tumbling Brownian motion within that region, sampling new, approximately random, points as it does so. The efficiency with which a region is explored is independent of its narrowness (for an ellipsoidal valley, the ratio of its principal axes) and orientation. If the temperature is reduced sufficiently slowly, it becomes highly likely that the simplex will shrink into that region containing the lowest relative minimum encountered.

As in all applications of simulated annealing, there can be quite a lot of problem-dependent subtlety in the phrase "sufficiently slowly"; success or failure is quite often determined by the choice of annealing schedule. Here are some possibilities worth trying:

- Reduce T to $(1-\epsilon)T$ after every m moves, where ϵ/m is determined by experiment.
- Budget a total of K moves, and reduce T after every m moves to a value $T = T_0(1-k/K)^\alpha$, where k is the cumulative number of moves thus far, and α is a constant, say 1, 2, or 4. The optimal value for α depends on the statistical distribution of relative minima of various depths. Larger values of α spend more iterations at lower temperature.
- After every m moves, set T to β times $f_1 - f_b$, where β is an experimentally determined constant of order 1, f_1 is the smallest function value currently represented in the simplex, and f_b is the best function ever encountered. However, never reduce T by more than some fraction γ at a time.

Another strategic question is whether to do an occasional *restart*, where a vertex of the simplex is discarded in favor of the "best-ever" point. (You must be sure that the best-ever point is not currently in the simplex when you do this!) We have found problems for which restarts — every time the temperature has decreased by a factor of 3, say — are highly beneficial; we have found other problems for which restarts have no positive, or a somewhat negative, effect.

You should compare the following routine, `amebsa`, with its counterpart `amoeba` in §10.4. Note that the argument `iter` is used in a somewhat different manner.

```
#include <math.h>
#include "nrutil.h"
#define GET_PSUM \
                  for (n=1;n<=ndim;n++) {\
                  for (sum=0.0,m=1;m<=mpts;m++) sum += p[m][n];\
                  psum[n]=sum;}
extern long idum;                          Defined and initialized in main.
float tt;                                  Communicates with amotsa.

void amebsa(float **p, float y[], int ndim, float pb[], float *yb, float ftol,
    float (*funk)(float []), int *iter, float temptr)
```

Multidimensional minimization of the function `funk(x)` where `x[1..ndim]` is a vector in `ndim` dimensions, by simulated annealing combined with the downhill simplex method of Nelder and Mead. The input matrix `p[1..ndim+1][1..ndim]` has `ndim+1` rows, each an `ndim`-dimensional vector which is a vertex of the starting simplex. Also input are the following: the vector `y[1..ndim+1]`, whose components must be pre-initialized to the values of `funk` evaluated at the `ndim+1` vertices (rows) of `p`; `ftol`, the fractional convergence tolerance to be achieved in the function value for an early return; `iter`, and `temptr`. The routine makes `iter` function evaluations at an annealing temperature `temptr`, then returns. You should then de-

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 132 of 181

crease temptr according to your annealing schedule, reset iter, and call the routine again (leaving other arguments unaltered between calls). If iter is returned with a positive value, then early convergence and return occurred. If you initialize yb to a very large value on the first call, then yb and pb[1..ndim] will subsequently return the best function value and point ever encountered (even if it is no longer a point in the simplex).

```
{
    float amotsa(float **p, float y[], float psum[], int ndim, float pb[],
        float *yb, float (*funk)(float []), int ihi, float *yhi, float fac);
    float ran1(long *idum);
    int i,ihi,ilo,j,m,n,mpts=ndim+1;
    float rtol,sum,swap,yhi,ylo,ynhi,ysave,yt,ytry,*psum;

    psum=vector(1,ndim);
    tt = -temptr;
    GET_PSUM
    for (;;) {
        ilo=1;                                  Determine which point is the highest (worst),
        ihi=2;                                      next-highest, and lowest (best).
        ynhi=ylo=y[1]+tt*log(ran1(&idum));      Whenever we "look at" a vertex, it gets
        yhi=y[2]+tt*log(ran1(&idum));               a random thermal fluctuation.
        if (ylo > yhi) {
            ihi=1;
            ilo=2;
            ynhi=yhi;
            yhi=ylo;
            ylo=ynhi;
        }
        for (i=3;i<=mpts;i++) {                 Loop over the points in the simplex.
            yt=y[i]+tt*log(ran1(&idum));        More thermal fluctuations.
            if (yt <= ylo) {
                ilo=i;
                ylo=yt;
            }
            if (yt > yhi) {
                ynhi=yhi;
                ihi=i;
                yhi=yt;
            } else if (yt > ynhi) {
                ynhi=yt;
            }
        }
        rtol=2.0*fabs(yhi-ylo)/(fabs(yhi)+fabs(ylo));
        Compute the fractional range from highest to lowest and return if satisfactory.
        if (rtol < ftol || *iter < 0) {         If returning, put best point and value in
            swap=y[1];                              slot 1.
            y[1]=y[ilo];
            y[ilo]=swap;
            for (n=1;n<=ndim;n++) {
                swap=p[1][n];
                p[1][n]=p[ilo][n];
                p[ilo][n]=swap;
            }
            break;
        }
        *iter -= 2;
        Begin a new iteration. First extrapolate by a factor -1 through the face of the simplex
        across from the high point, i.e., reflect the simplex from the high point.
        ytry=amotsa(p,y,psum,ndim,pb,yb,funk,ihi,&yhi,-1.0);
        if (ytry <= ylo) {
            Gives a result better than the best point, so try an additional extrapolation by a
            factor of 2.
            ytry=amotsa(p,y,psum,ndim,pb,yb,funk,ihi,&yhi,2.0);
        } else if (ytry >= ynhi) {
            The reflected point is worse than the second-highest, so look for an intermediate
```

```
            lower point, i.e., do a one-dimensional contraction.
            ysave=yhi;
            ytry=amotsa(p,y,psum,ndim,pb,yb,funk,ihi,&yhi,0.5);
            if (ytry >= ysave) {                Can't seem to get rid of that high point.
                for (i=1;i<=mpts;i++) {             Better contract around the lowest
                    if (i != ilo) {                 (best) point.
                        for (j=1;j<=ndim;j++) {
                            psum[j]=0.5*(p[i][j]+p[ilo][j]);
                            p[i][j]=psum[j];
                        }
                        y[i]=(*funk)(psum);
                    }
                }
                *iter -= ndim;
                GET_PSUM                        Recompute psum.
            }
        } else ++(*iter);                       Correct the evaluation count.
    }
    free_vector(psum,1,ndim);
}
```

```
#include <math.h>
#include "nrutil.h"

extern long idum;                       Defined and initialized in main.
extern float tt;                        Defined in amebsa.

float amotsa(float **p, float y[], float psum[], int ndim, float pb[],
    float *yb, float (*funk)(float []), int ihi, float *yhi, float fac)
Extrapolates by a factor fac through the face of the simplex across from the high point, tries
it, and replaces the high point if the new point is better.
{
    float ran1(long *idum);
    int j;
    float fac1,fac2,yflu,ytry,*ptry;

    ptry=vector(1,ndim);
    fac1=(1.0-fac)/ndim;
    fac2=fac1-fac;
    for (j=1;j<=ndim;j++)
        ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
    ytry=(*funk)(ptry);
    if (ytry <= *yb) {                  Save the best-ever.
        for (j=1;j<=ndim;j++) pb[j]=ptry[j];
        *yb=ytry;
    }
    yflu=ytry-tt*log(ran1(&idum));      We added a thermal fluctuation to all the current
    if (yflu < *yhi) {                      vertices, but we subtract it here, so as to give
        y[ihi]=ytry;                        the simplex a thermal Brownian motion: It
        *yhi=yflu;                          likes to accept any suggested change.
        for (j=1;j<=ndim;j++) {
            psum[j] += ptry[j]-p[ihi][j];
            p[ihi][j]=ptry[j];
        }
    }
    free_vector(ptry,1,ndim);
    return yflu;
}
```

There is not yet enough practical experience with the method of simulated annealing to say definitively what its future place among optimization methods

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 134 of 181

will be. The method has several extremely attractive features, rather unique when compared with other optimization techniques.

First, it is not "greedy," in the sense that it is not easily fooled by the quick payoff achieved by falling into unfavorable local minima. Provided that sufficiently general reconfigurations are given, it wanders freely among local minima of depth less than about T. As T is lowered, the number of such minima qualifying for frequent visits is gradually reduced.

Second, configuration decisions tend to proceed in a logical order. Changes that cause the greatest energy differences are sifted over when the control parameter T is large. These decisions become more permanent as T is lowered, and attention then shifts more to smaller refinements in the solution. For example, in the traveling salesman problem with the Mississippi River twist, if λ is large, a decision to cross the Mississippi only twice is made at high T, while the specific routes on each side of the river are determined only at later stages.

The analogies to thermodynamics may be pursued to a greater extent than we have done here. Quantities analogous to specific heat and entropy may be defined, and these can be useful in monitoring the progress of the algorithm towards an acceptable solution. Information on this subject is found in [1].

CITED REFERENCES AND FURTHER READING:

Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. 1983, *Science*, vol. 220, pp. 671–680. [1]

Kirkpatrick, S. 1984, *Journal of Statistical Physics*, vol. 34, pp. 975–986. [2]

Vecchi, M.P. and Kirkpatrick, S. 1983, *IEEE Transactions on Computer Aided Design*, vol. CAD-2, pp. 215–222. [3]

Otten, R.H.J.M., and van Ginneken, L.P.P.P. 1989, *The Annealing Algorithm* (Boston: Kluwer) [contains many references to the literature]. [4]

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller A., and Teller, E. 1953, *Journal of Chemical Physics*, vol. 21, pp. 1087–1092. [5]

Lin, S. 1965, *Bell System Technical Journal*, vol. 44, pp. 2245–2269. [6]

Vanderbilt, D., and Louie, S.G. 1984, *Journal of Computational Physics*, vol. 56, pp. 259–271. [7]

Bohachevsky, I.O., Johnson, M.E., and Stein, M.L. 1986, *Technometrics*, vol. 28, pp. 209–217. [8]

Corana, A., Marchesi, M., Martini, C., and Ridella, S. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 262–280. [9]

Bélisle, C.J.P., Romeijn, H.E., and Smith, R.L. 1990, Technical Report 90–25, Department of Industrial and Operations Engineering, University of Michigan, submitted to *Mathematical Programming*. [10]

Christofides, N., Mingozzi, A., Toth, P., and Sandi, C. (eds.) 1979, *Combinatorial Optimization* (London and New York: Wiley-Interscience) [not simulated annealing, but other topics and algorithms].

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 135 of 181

Chapter 16. Integration of Ordinary Differential Equations

16.0 Introduction

Problems involving ordinary differential equations (ODEs) can always be reduced to the study of sets of first-order differential equations. For example the second-order equation

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \tag{16.0.1}$$

can be rewritten as two first-order equations

$$\begin{aligned} \frac{dy}{dx} &= z(x) \\ \frac{dz}{dx} &= r(x) - q(x)z(x) \end{aligned} \tag{16.0.2}$$

where z is a new variable. This exemplifies the procedure for an arbitrary ODE. The usual choice for the new variables is to let them be just derivatives of each other (and of the original variable). Occasionally, it is useful to incorporate into their definition some other factors in the equation, or some powers of the independent variable, for the purpose of mitigating singular behavior that could result in overflows or increased roundoff error. Let common sense be your guide: If you find that the original variables are smooth in a solution, while your auxiliary variables are doing crazy things, then figure out why and choose different auxiliary variables.

The generic problem in ordinary differential equations is thus reduced to the study of a set of N coupled *first-order* differential equations for the functions $y_i,\ i = 1, 2, \ldots, N$, having the general form

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \ldots, y_N), \qquad i = 1, \ldots, N \tag{16.0.3}$$

where the functions f_i on the right-hand side are known.

A problem involving ODEs is not completely specified by its equations. Even more crucial in determining how to attack the problem numerically is the nature of the problem's boundary conditions. Boundary conditions are algebraic conditions on the values of the functions y_i in (16.0.3). In general they can be satisfied at

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 136 of 181

discrete specified points, but do not hold between those points, i.e., are not preserved automatically by the differential equations. Boundary conditions can be as simple as requiring that certain variables have certain numerical values, or as complicated as a set of nonlinear algebraic equations among the variables.

Usually, it is the nature of the boundary conditions that determines which numerical methods will be feasible. Boundary conditions divide into two broad categories.

- In *initial value problems* all the y_i are given at some starting value x_s, and it is desired to find the y_i's at some final point x_f, or at some discrete list of points (for example, at tabulated intervals).
- In *two-point boundary value problems*, on the other hand, boundary conditions are specified at more than one x. Typically, some of the conditions will be specified at x_s and the remainder at x_f.

This chapter will consider exclusively the initial value problem, deferring two-point boundary value problems, which are generally more difficult, to Chapter 17.

The underlying idea of any routine for solving the initial value problem is always this: Rewrite the dy's and dx's in (16.0.3) as finite steps Δy and Δx, and multiply the equations by Δx. This gives algebraic formulas for the change in the functions when the independent variable x is "stepped" by one "stepsize" Δx. In the limit of making the stepsize very small, a good approximation to the underlying differential equation is achieved. Literal implementation of this procedure results in *Euler's method* (16.1.1, below), which is, however, *not* recommended for any practical use. Euler's method is conceptually important, however; one way or another, practical methods all come down to this same idea: Add small increments to your functions corresponding to derivatives (right-hand sides of the equations) multiplied by stepsizes.

In this chapter we consider three major types of practical numerical methods for solving initial value problems for ODEs:

- Runge-Kutta methods
- Richardson extrapolation and its particular implementation as the Bulirsch-Stoer method
- predictor-corrector methods.

A brief description of each of these types follows.

1. *Runge-Kutta* methods propagate a solution over an interval by combining the information from several Euler-style steps (each involving one evaluation of the right-hand f's), and then using the information obtained to match a Taylor series expansion up to some higher order.

2. *Richardson extrapolation* uses the powerful idea of extrapolating a computed result to the value that *would* have been obtained if the stepsize had been very much smaller than it actually was. In particular, extrapolation to zero stepsize is the desired goal. The first practical ODE integrator that implemented this idea was developed by Bulirsch and Stoer, and so extrapolation methods are often called Bulirsch-Stoer methods.

3. *Predictor-corrector* methods store the solution along the way, and use those results to extrapolate the solution one step advanced; they then correct the extrapolation using derivative information at the new point. These are best for very smooth functions.

Runge-Kutta is what you use when (i) you don't know any better, or (ii) you have an intransigent problem where Bulirsch-Stoer is failing, or (iii) you have a trivial

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 137 of 181

problem where computational efficiency is of no concern. Runge-Kutta succeeds virtually always; but it is not usually fastest, except when evaluating f_i is cheap and moderate accuracy ($\lesssim 10^{-5}$) is required. Predictor-corrector methods, since they use past information, are somewhat more difficult to start up, but, for many smooth problems, they are computationally more efficient than Runge-Kutta. In recent years Bulirsch-Stoer has been replacing predictor-corrector in many applications, but it is too soon to say that predictor-corrector is dominated in all cases. However, it appears that only rather sophisticated predictor-corrector routines are competitive. Accordingly, we have chosen *not* to give an implementation of predictor-corrector in this book. We discuss predictor-corrector further in §16.7, so that you can use a canned routine should you encounter a suitable problem. In our experience, the relatively simple Runge-Kutta and Bulirsch-Stoer routines we give are adequate for most problems.

Each of the three types of methods can be organized to monitor internal consistency. This allows numerical errors which are inevitably introduced into the solution to be controlled by automatic, (*adaptive*) changing of the fundamental stepsize. We always recommend that adaptive stepsize control be implemented, and we will do so below.

In general, all three types of methods can be applied to any initial value problem. Each comes with its own set of debits and credits that must be understood before it is used.

We have organized the routines in this chapter into three nested levels. The lowest or "nitty-gritty" level is the piece we call the *algorithm* routine. This implements the basic formulas of the method, starts with dependent variables y_i at x, and calculates new values of the dependent variables at the value $x + h$. The algorithm routine also yields up some information about the quality of the solution after the step. The routine is dumb, however, and it is unable to make any adaptive decision about whether the solution is of acceptable quality or not.

That quality-control decision we encode in a *stepper* routine. The stepper routine calls the algorithm routine. It may reject the result, set a smaller stepsize, and call the algorithm routine again, until compatibility with a predetermined accuracy criterion has been achieved. The stepper's fundamental task is to take the largest stepsize consistent with specified performance. Only when this is accomplished does the true power of an algorithm come to light.

Above the stepper is the *driver* routine, which starts and stops the integration, stores intermediate results, and generally acts as an interface with the user. There is nothing at all canonical about our driver routines. You should consider them to be examples, and you can customize them for your particular application.

Of the routines that follow, `rk4`, `rkck`, `mmid`, `stoerm`, and `simpr` are algorithm routines; `rkqs`, `bsstep`, `stiff`, and `stifbs` are steppers; `rkdumb` and `odeint` are drivers.

Section 16.6 of this chapter treats the subject of *stiff equations*, relevant both to ordinary differential equations and also to partial differential equations (Chapter 19).

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 138 of 181

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 5.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 7.

Lambert, J. 1973, *Computational Methods in Ordinary Differential Equations* (New York: Wiley).

Lapidus, L., and Seinfeld, J. 1971, *Numerical Solution of Ordinary Differential Equations* (New York: Academic Press).

16.1 Runge-Kutta Method

The formula for the Euler method is

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{16.1.1}$$

which advances a solution from x_n to $x_{n+1} \equiv x_n + h$. The formula is unsymmetrical: It advances the solution through an interval h, but uses derivative information only at the beginning of that interval (see Figure 16.1.1). That means (and you can verify by expansion in power series) that the step's error is only one power of h smaller than the correction, i.e $O(h^2)$ added to (16.1.1).

There are several reasons that Euler's method is not recommended for practical use, among them, (i) the method is not very accurate when compared to other, fancier, methods run at the equivalent stepsize, and (ii) neither is it very stable (see §16.6 below).

Consider, however, the use of a step like (16.1.1) to take a "trial" step to the midpoint of the interval. Then use the value of both x and y at that midpoint to compute the "real" step across the whole interval. Figure 16.1.2 illustrates the idea. In equations,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{16.1.2}$$

As indicated in the error term, this symmetrization cancels out the first-order error term, making the method *second order*. [A method is conventionally called nth order if its error term is $O(h^{n+1})$.] In fact, (16.1.2) is called the *second-order Runge-Kutta* or *midpoint* method.

We needn't stop there. There are many ways to evaluate the right-hand side $f(x, y)$ that all agree to first order, but that have different coefficients of higher-order error terms. Adding up the right combination of these, we can eliminate the error terms order by order. That is the basic idea of the Runge-Kutta method. Abramowitz and Stegun [1], and Gear [2], give various specific formulas that derive from this basic

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 139 of 181

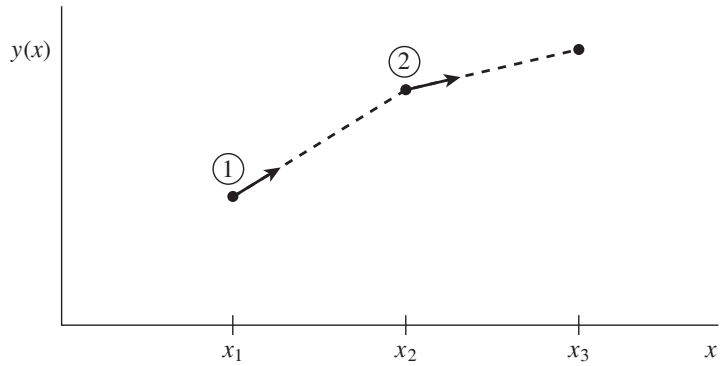

Figure 16.1.1. Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

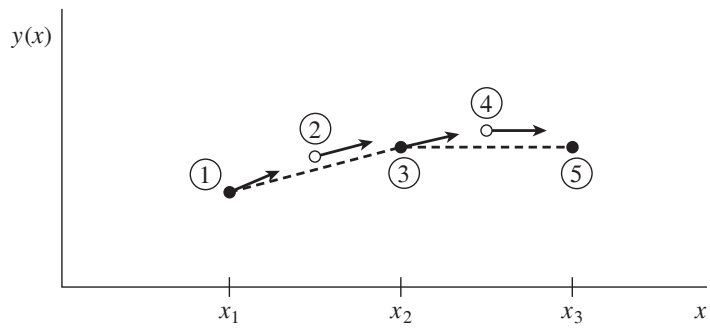

Figure 16.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open dots represent function values that are discarded once their derivatives have been calculated and used.

idea. By far the most often used is the classical *fourth-order Runge-Kutta formula*, which has a certain sleekness of organization about it:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5) \end{aligned} \tag{16.1.3}$$

The fourth-order Runge-Kutta method requires four evaluations of the right-hand side per step h (see Figure 16.1.3). This will be superior to the midpoint method (16.1.2) *if* at least twice as large a step is possible with (16.1.3) for the same accuracy. Is that so? The answer is: often, perhaps even usually, but surely not always! This takes us back to a central theme, namely that *high order* does not always mean *high accuracy*. The statement "fourth-order Runge-Kutta is generally superior to second-order" is a true one, but you should recognize it as a statement about the

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 140 of 181

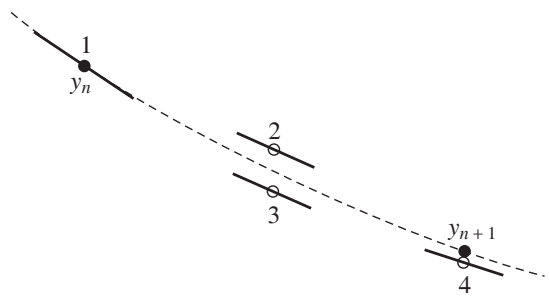

Figure 16.1.3. Fourth-order Runge-Kutta method. In each step the derivative is evaluated four times: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value (shown as a filled dot) is calculated. (See text for details.)

contemporary practice of science rather than as a statement about strict mathematics. That is, it reflects the nature of the problems that contemporary scientists like to solve.

For many scientific users, fourth-order Runge-Kutta is not just the first word on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm. Keep in mind, however, that the old workhorse's last trip may well be to take you to the poorhouse: Bulirsch-Stoer or predictor-corrector methods can be very much more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields. However, even the old workhorse is more nimble with new horseshoes. In §16.2 we will give a modern implementation of a Runge-Kutta method that is quite competitive as long as very high accuracy is not required. An excellent discussion of the pitfalls in constructing a good Runge-Kutta code is given in [3].

Here is the routine for carrying out one classical Runge-Kutta step on a set of `n` differential equations. You input the values of the independent variables, and you get out new values which are stepped by a stepsize `h` (which can be positive or negative). You will notice that the routine requires you to supply not only function `derivs` for calculating the right-hand side, but also values of the derivatives at the starting point. Why not let the routine call `derivs` for this first value? The answer will become clear only in the next section, but in brief is this: This call may not be your only one with these starting conditions. You may have taken a previous step with too large a stepsize, and this is your replacement. In that case, you do not want to call `derivs` unnecessarily at the start. Note that the routine that follows has, therefore, only three calls to `derivs`.

```
#include "nrutil.h"

void rk4(float y[], float dydx[], int n, float x, float h, float yout[],
    void (*derivs)(float, float [], float []))
Given values for the variables y[1..n] and their derivatives dydx[1..n] known at x, use the
fourth-order Runge-Kutta method to advance the solution over an interval h and return the
incremented variables as yout[1..n], which need not be a distinct array from y. The user
supplies the routine derivs(x,y,dydx), which returns derivatives dydx at x.
{
    int i;
    float xh,hh,h6,*dym,*dyt,*yt;

    dym=vector(1,n);
    dyt=vector(1,n);
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 141 of 181

```
    yt=vector(1,n);
    hh=h*0.5;
    h6=h/6.0;
    xh=x+hh;
    for (i=1;i<=n;i++) yt[i]=y[i]+hh*dydx[i];        First step.
    (*derivs)(xh,yt,dyt);                            Second step.
    for (i=1;i<=n;i++) yt[i]=y[i]+hh*dyt[i];
    (*derivs)(xh,yt,dym);                            Third step.
    for (i=1;i<=n;i++) {
        yt[i]=y[i]+h*dym[i];
        dym[i] += dyt[i];
    }
    (*derivs)(x+h,yt,dyt);                           Fourth step.
    for (i=1;i<=n;i++)                               Accumulate increments with proper
        yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);     weights.
    free_vector(yt,1,n);
    free_vector(dyt,1,n);
    free_vector(dym,1,n);
}
```

The Runge-Kutta method treats every step in a sequence of steps in identical manner. Prior behavior of a solution is not used in its propagation. This is mathematically proper, since any point along the trajectory of an ordinary differential equation can serve as an initial point. The fact that all steps are treated identically also makes it easy to incorporate Runge-Kutta into relatively simple "driver" schemes.

We consider adaptive stepsize control, discussed in the next section, an essential for serious computing. Occasionally, however, you just want to tabulate a function at equally spaced intervals, and without particularly high accuracy. In the most common case, you want to produce a graph of the function. Then all you need may be a simple driver program that goes from an initial x_s to a final x_f in a specified number of steps. To check accuracy, double the number of steps, repeat the integration, and compare results. This approach surely does not minimize computer time, and it can fail for problems whose nature *requires* a variable stepsize, but it may well minimize user effort. On small problems, this may be the paramount consideration.

Here is such a driver, self-explanatory, which tabulates the integrated functions in the global arrays `*x` and `**y`; be sure to allocate memory for them with the routines `vector()` and `matrix()`, respectively.

```
#include "nrutil.h"

float **y,*xx;                          For communication back to main.

void rkdumb(float vstart[], int nvar, float x1, float x2, int nstep,
    void (*derivs)(float, float [], float []))
Starting from initial values vstart[1..nvar] known at x1 use fourth-order Runge-Kutta
to advance nstep equal increments to x2. The user-supplied routine derivs(x,v,dvdx)
evaluates derivatives. Results are stored in the global variables y[1..nvar][1..nstep+1]
and xx[1..nstep+1].
{
    void rk4(float y[], float dydx[], int n, float x, float h, float yout[],
        void (*derivs)(float, float [], float []));
    int i,k;
    float x,h;
    float *v,*vout,*dv;

    v=vector(1,nvar);
    vout=vector(1,nvar);
```

```
    dv=vector(1,nvar);
    for (i=1;i<=nvar;i++) {          Load starting values.
        v[i]=vstart[i];
        y[i][1]=v[i];
    }
    xx[1]=x1;
    x=x1;
    h=(x2-x1)/nstep;
    for (k=1;k<=nstep;k++) {         Take nstep steps.
        (*derivs)(x,v,dv);
        rk4(v,dv,nvar,x,h,vout,derivs);
        if ((float)(x+h) == x) nrerror("Step size too small in routine rkdumb");
        x += h;
        xx[k+1]=x;                   Store intermediate steps.
        for (i=1;i<=nvar;i++) {
            v[i]=vout[i];
            y[i][k+1]=v[i];
        }
    }
    free_vector(dv,1,nvar);
    free_vector(vout,1,nvar);
    free_vector(v,1,nvar);
}
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.5. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121. [3]

Rice, J.R. 1983, *Numerical Methods, Software, and Analysis* (New York: McGraw-Hill), §9.2.

16.2 Adaptive Stepsize Control for Runge-Kutta

A good ODE integrator should exert some adaptive control over its own progress, making frequent changes in its stepsize. Usually the purpose of this adaptive stepsize control is to achieve some predetermined accuracy in the solution with minimum computational effort. Many small steps should tiptoe through treacherous terrain, while a few great strides should speed through smooth uninteresting countryside. The resulting gains in efficiency are not mere tens of percents or factors of two; they can sometimes be factors of ten, a hundred, or more. Sometimes accuracy may be demanded not directly in the solution itself, but in some related conserved quantity that can be monitored.

Implementation of adaptive stepsize control requires that the stepping algorithm signal information about its performance, most important, an estimate of its truncation error. In this section we will learn how such information can be obtained. Obviously,

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 143 of 181

the calculation of this information will add to the computational overhead, but the investment will generally be repaid handsomely.

With fourth-order Runge-Kutta, the most straightforward technique by far is *step doubling* (see, e.g., [1]). We take each step twice, once as a full step, then, independently, as two half steps (see Figure 16.2.1). How much overhead is this, say in terms of the number of evaluations of the right-hand sides? Each of the three separate Runge-Kutta steps in the procedure requires 4 evaluations, but the single and double sequences share a starting point, so the total is 11. This is to be compared not to 4, but to 8 (the two half-steps), since — stepsize control aside — we are achieving the accuracy of the smaller (half) stepsize. The overhead cost is therefore a factor 1.375. What does it buy us?

Let us denote the exact solution for an advance from x to $x + 2h$ by $y(x + 2h)$ and the two approximate solutions by y_1 (one step $2h$) and y_2 (2 steps each of size h). Since the basic method is fourth order, the true solution and the two numerical approximations are related by

$$\begin{aligned} y(x+2h) &= y_1 + (2h)^5\phi + O(h^6) + \dots \\ y(x+2h) &= y_2 + 2(h^5)\phi + O(h^6) + \dots \end{aligned} \tag{16.2.1}$$

where, to order h^5, the value ϕ remains constant over the step. [Taylor series expansion tells us the ϕ is a number whose order of magnitude is $y^{(5)}(x)/5!$.] The first expression in (16.2.1) involves $(2h)^5$ since the stepsize is $2h$, while the second expression involves $2(h^5)$ since the error on each step is $h^5\phi$. The difference between the two numerical estimates is a convenient indicator of truncation error

$$\Delta \equiv y_2 - y_1 \tag{16.2.2}$$

It is this difference that we shall endeavor to keep to a desired degree of accuracy, neither too large nor too small. We do this by adjusting h.

It might also occur to you that, ignoring terms of order h^6 and higher, we can solve the two equations in (16.2.1) to improve our numerical estimate of the true solution $y(x + 2h)$, namely,

$$y(x+2h) = y_2 + \frac{\Delta}{15} + O(h^6) \tag{16.2.3}$$

This estimate is accurate to *fifth order*, one order higher than the original Runge-Kutta steps. However, we can't have our cake and eat it: (16.2.3) may be fifth-order accurate, but we have no way of monitoring *its* truncation error. Higher order is not always higher accuracy! Use of (16.2.3) rarely does harm, but we have no way of directly knowing whether it is doing any good. Therefore we should use Δ as the error estimate and take as "gravy" any additional accuracy gain derived from (16.2.3). In the technical literature, use of a procedure like (16.2.3) is called "local extrapolation."

An alternative stepsize adjustment algorithm is based on the *embedded Runge-Kutta formulas*, originally invented by Fehlberg. An interesting fact about Runge-Kutta formulas is that for orders M higher than four, more than M function evaluations (though never more than $M + 2$) are required. This accounts for the

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 144 of 181

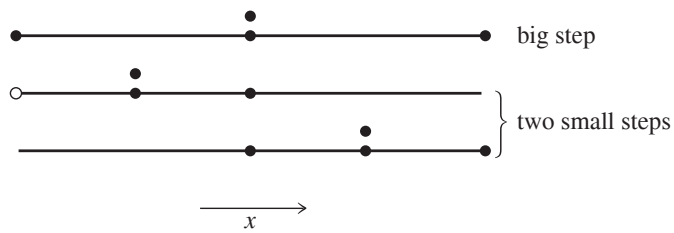

Figure 16.2.1. Step-doubling as a means for adaptive stepsize control in fourth-order Runge-Kutta. Points where the derivative is evaluated are shown as filled circles. The open circle represents the same derivatives as the filled circle immediately above it, so the total number of evaluations is 11 per two steps. Comparing the accuracy of the big step with the two small steps gives a criterion for adjusting the stepsize on the next step, or for rejecting the current step as inaccurate.

popularity of the classical fourth-order method: It seems to give the most bang for the buck. However, Fehlberg discovered a fifth-order method with six function evaluations where another combination of the six functions gives a fourth-order method. The difference between the two estimates of $y(x+h)$ can then be used as an estimate of the truncation error to adjust the stepsize. Since Fehlberg's original formula, several other embedded Runge-Kutta formulas have been found.

Many practitioners were at one time wary of the robustness of Runge-Kutta-Fehlberg methods. The feeling was that using the same evaluation points to advance the function and to estimate the error was riskier than step-doubling, where the error estimate is based on independent function evaluations. However, experience has shown that this concern is not a problem in practice. Accordingly, embedded Runge-Kutta formulas, which are roughly a factor of two more efficient, have superseded algorithms based on step-doubling.

The general form of a fifth-order Runge-Kutta formula is

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + a_2 h, y_n + b_{21} k_1) \\ &\cdots \\ k_6 &= hf(x_n + a_6 h, y_n + b_{61} k_1 + \cdots + b_{65} k_5) \\ y_{n+1} &= y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 + O(h^6) \end{aligned} \tag{16.2.4}$$

The embedded fourth-order formula is

$$y^*_{n+1} = y_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4 + c^*_5 k_5 + c^*_6 k_6 + O(h^5) \tag{16.2.5}$$

and so the error estimate is

$$\Delta \equiv y_{n+1} - y^*_{n+1} = \sum_{i=1}^{6} (c_i - c^*_i) k_i \tag{16.2.6}$$

The particular values of the various constants that we favor are those found by Cash and Karp [2], and given in the accompanying table. These give a more efficient method than Fehlberg's original values, with somewhat better error properties.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 145 of 181

Cash-Karp Parameters for Embedded Runga-Kutta Method								
i	a_i	b_{ij}					c_i	c_i^*
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$
$j =$		1	2	3	4	5		

Now that we know, at least approximately, what our error is, we need to consider how to keep it within desired bounds. What is the relation between Δ and h? According to (16.2.4) – (16.2.5), Δ scales as h^5. If we take a step h_1 and produce an error Δ_1, therefore, the step h_0 that *would have given* some other value Δ_0 is readily estimated as

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2} \tag{16.2.7}$$

Henceforth we will let Δ_0 denote the *desired* accuracy. Then equation (16.2.7) is used in two ways: If Δ_1 is larger than Δ_0 in magnitude, the equation tells how much to decrease the stepsize *when we retry the present (failed) step*. If Δ_1 is smaller than Δ_0, on the other hand, then the equation tells how much we can safely increase the stepsize *for the next step*. Local extrapolation consists in accepting the fifth order value y_{n+1}, even though the error estimate actually applies to the fourth order value y^*_{n+1}.

Our notation hides the fact that Δ_0 is actually a vector of desired accuracies, one for each equation in the set of ODEs. In general, our accuracy requirement will be that all equations are within their respective allowed errors. In other words, we will rescale the stepsize according to the needs of the "worst-offender" equation.

How is Δ_0, the desired accuracy, related to some looser prescription like "get a solution good to one part in 10^6"? That can be a subtle question, and it depends on exactly what your application is! You may be dealing with a set of equations whose dependent variables differ enormously in magnitude. In that case, you probably want to use fractional errors, $\Delta_0 = \epsilon y$, where ϵ is the number like 10^{-6} or whatever. On the other hand, you may have oscillatory functions that pass through zero but are bounded by some maximum values. In that case you probably want to set Δ_0 equal to ϵ times those maximum values.

A convenient way to fold these considerations into a generally useful stepper routine is this: One of the arguments of the routine will of course be the vector of dependent variables at the beginning of a proposed step. Call that `y[1..n]`. Let us require the user to specify for each step another, corresponding, vector argument `yscal[1..n]`, and also an overall tolerance level `eps`. Then the desired accuracy

for the ith equation will be taken to be

$$\Delta_0 = \texttt{eps} \times \texttt{yscal[i]} \tag{16.2.8}$$

If you desire constant fractional errors, plug a pointer to `y` into the pointer to `yscal` calling slot (no need to copy the values into a different array). If you desire constant absolute errors relative to some maximum values, set the elements of `yscal` equal to those maximum values. A useful "trick" for getting constant fractional errors *except* "very" near zero crossings is to set `yscal[i]` equal to $|\texttt{y[i]}| + |h \times \texttt{dydx[i]}|$. (The routine `odeint`, below, does this.)

Here is a more technical point. We have to consider one additional possibility for `yscal`. The error criteria mentioned thus far are "local," in that they bound the error of each step individually. In some applications you may be unusually sensitive about a "global" accumulation of errors, from beginning to end of the integration and in the worst possible case where the errors all are presumed to add with the same sign. Then, the smaller the stepsize h, the smaller the value Δ_0 that you will need to impose. Why? Because there will be *more steps* between your starting and ending values of x. In such cases you will want to set `yscal` proportional to h, typically to something like

$$\Delta_0 = \epsilon h \times \texttt{dydx[i]} \tag{16.2.9}$$

This enforces fractional accuracy ϵ not on the values of `y` but (much more stringently) on the *increments* to those values at each step. But now look back at (16.2.7). If Δ_0 has an implicit scaling with h, then the exponent 0.20 is no longer correct: When the stepsize is reduced from a too-large value, the new predicted value h_1 will fail to meet the desired accuracy when `yscal` is also altered to this new h_1 value. Instead of $0.20 = 1/5$, we must scale by the exponent $0.25 = 1/4$ for things to work out.

The exponents 0.20 and 0.25 are not really very different. This motivates us to adopt the following pragmatic approach, one that frees us from having to know in advance whether or not you, the user, plan to scale your `yscal`'s with stepsize. Whenever we decrease a stepsize, let us use the larger value of the exponent (whether we need it or not!), and whenever we increase a stepsize, let us use the smaller exponent. Furthermore, because our estimates of error are not exact, but only accurate to the leading order in h, we are advised to put in a safety factor S which is a few percent smaller than unity. Equation (16.2.7) is thus replaced by

$$h_0 = \begin{cases} Sh_1 \left|\dfrac{\Delta_0}{\Delta_1}\right|^{0.20} & \Delta_0 \geq \Delta_1 \\ Sh_1 \left|\dfrac{\Delta_0}{\Delta_1}\right|^{0.25} & \Delta_0 < \Delta_1 \end{cases} \tag{16.2.10}$$

We have found this prescription to be a reliable one in practice.

Here, then, is a stepper program that takes one "quality-controlled" Runge-Kutta step.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 147 of 181

```
#include <math.h>
#include "nrutil.h"
#define SAFETY 0.9
#define PGROW -0.2
#define PSHRNK -0.25
#define ERRCON 1.89e-4
```

The value ERRCON equals (5/SAFETY) raised to the power (1/PGROW), see use below.

```
void rkqs(float y[], float dydx[], int n, float *x, float htry, float eps,
    float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []))
```

Fifth-order Runge-Kutta step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector y[1..n] and its derivative dydx[1..n] at the starting value of the independent variable x. Also input are the stepsize to be attempted htry, the required accuracy eps, and the vector yscal[1..n] against which the error is scaled. On output, y and x are replaced by their new values, hdid is the stepsize that was actually accomplished, and hnext is the estimated next stepsize. derivs is the user-supplied routine that computes the right-hand side derivatives.

```
{
    void rkck(float y[], float dydx[], int n, float x, float h,
        float yout[], float yerr[], void (*derivs)(float, float [], float []));
    int i;
    float errmax,h,htemp,xnew,*yerr,*ytemp;

    yerr=vector(1,n);
    ytemp=vector(1,n);
    h=htry;                                 Set stepsize to the initial trial value.
    for (;;) {
        rkck(y,dydx,n,*x,h,ytemp,yerr,derivs);        Take a step.
        errmax=0.0;                         Evaluate accuracy.
        for (i=1;i<=n;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
        errmax /= eps;                      Scale relative to required tolerance.
        if (errmax <= 1.0) break;           Step succeeded. Compute size of next step.
        htemp=SAFETY*h*pow(errmax,PSHRNK);
        Truncation error too large, reduce stepsize.
        h=(h >= 0.0 ? FMAX(htemp,0.1*h) : FMIN(htemp,0.1*h));
        No more than a factor of 10.
        xnew=(*x)+h;
        if (xnew == *x) nrerror("stepsize underflow in rkqs");
    }
    if (errmax > ERRCON) *hnext=SAFETY*h*pow(errmax,PGROW);
    else *hnext=5.0*h;                      No more than a factor of 5 increase.
    *x += (*hdid=h);
    for (i=1;i<=n;i++) y[i]=ytemp[i];
    free_vector(ytemp,1,n);
    free_vector(yerr,1,n);
}
```

The routine rkqs calls the routine rkck to take a Cash-Karp Runge-Kutta step:

```
#include "nrutil.h"

void rkck(float y[], float dydx[], int n, float x, float h, float yout[],
    float yerr[], void (*derivs)(float, float [], float []))
```

Given values for n variables y[1..n] and their derivatives dydx[1..n] known at x, use the fifth-order Cash-Karp Runge-Kutta method to advance the solution over an interval h and return the incremented variables as yout[1..n]. Also return an estimate of the local truncation error in yout using the embedded fourth-order method. The user supplies the routine derivs(x,y,dydx), which returns derivatives dydx at x.

```
{
    int i;
```

```
    static float a2=0.2,a3=0.3,a4=0.6,a5=1.0,a6=0.875,b21=0.2,
        b31=3.0/40.0,b32=9.0/40.0,b41=0.3,b42 = -0.9,b43=1.2,
        b51 = -11.0/54.0, b52=2.5,b53 = -70.0/27.0,b54=35.0/27.0,
        b61=1631.0/55296.0,b62=175.0/512.0,b63=575.0/13824.0,
        b64=44275.0/110592.0,b65=253.0/4096.0,c1=37.0/378.0,
        c3=250.0/621.0,c4=125.0/594.0,c6=512.0/1771.0,
        dc5 = -277.00/14336.0;
    float dc1=c1-2825.0/27648.0,dc3=c3-18575.0/48384.0,
        dc4=c4-13525.0/55296.0,dc6=c6-0.25;
    float *ak2,*ak3,*ak4,*ak5,*ak6,*ytemp;

    ak2=vector(1,n);
    ak3=vector(1,n);
    ak4=vector(1,n);
    ak5=vector(1,n);
    ak6=vector(1,n);
    ytemp=vector(1,n);
    for (i=1;i<=n;i++)                  First step.
        ytemp[i]=y[i]+b21*h*dydx[i];
    (*derivs)(x+a2*h,ytemp,ak2);        Second step.
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b31*dydx[i]+b32*ak2[i]);
    (*derivs)(x+a3*h,ytemp,ak3);        Third step.
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b41*dydx[i]+b42*ak2[i]+b43*ak3[i]);
    (*derivs)(x+a4*h,ytemp,ak4);        Fourth step.
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b51*dydx[i]+b52*ak2[i]+b53*ak3[i]+b54*ak4[i]);
    (*derivs)(x+a5*h,ytemp,ak5);        Fifth step.
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b61*dydx[i]+b62*ak2[i]+b63*ak3[i]+b64*ak4[i]+b65*ak5[i]);
    (*derivs)(x+a6*h,ytemp,ak6);        Sixth step.
    for (i=1;i<=n;i++)                  Accumulate increments with proper weights.
        yout[i]=y[i]+h*(c1*dydx[i]+c3*ak3[i]+c4*ak4[i]+c6*ak6[i]);
    for (i=1;i<=n;i++)
        yerr[i]=h*(dc1*dydx[i]+dc3*ak3[i]+dc4*ak4[i]+dc5*ak5[i]+dc6*ak6[i]);
        Estimate error as difference between fourth and fifth order methods.
    free_vector(ytemp,1,n);
    free_vector(ak6,1,n);
    free_vector(ak5,1,n);
    free_vector(ak4,1,n);
    free_vector(ak3,1,n);
    free_vector(ak2,1,n);
}
```

Noting that the above routines are all in single precision, don't be too greedy in specifying `eps`. The punishment for excessive greediness is interesting and worthy of Gilbert and Sullivan's *Mikado*: The routine can always achieve an apparent *zero* error by making the stepsize so small that quantities of order hy' add to quantities of order y as if they were zero. Then the routine chugs happily along taking infinitely many infinitesimal steps and never changing the dependent variables one iota. (You guard against this catastrophic loss of your computer budget by signaling on abnormally small stepsizes or on the dependent variable vector remaining unchanged from step to step. On a personal workstation you guard against it by not taking too long a lunch hour while your program is running.)

Here is a full-fledged "driver" for Runge-Kutta with adaptive stepsize control. We warmly recommend this routine, or one like it, for a variety of problems, notably

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 149 of 181

including garden-variety ODEs or sets of ODEs, and definite integrals (augmenting the methods of Chapter 4). For storage of intermediate results (if you desire to inspect them) we assume that the top-level pointer references `*xp` and `**yp` have been validly initialized (e.g., by the utilities `vector()` and `matrix()`). Because steps occur at unequal intervals results are only stored at intervals greater than `dxsav`. The top-level variable `kmax` indicates the maximum number of steps that can be stored. If `kmax=0` there is no intermediate storage, and the pointers `*xp` and `**yp` need not point to valid memory. Storage of steps stops if `kmax` is exceeded, except that the ending values are always stored. Again, these controls are merely indicative of what you might need. The routine `odeint` should be customized to the problem at hand.

```
#include <math.h>
#include "nrutil.h"
#define MAXSTP 10000
#define TINY 1.0e-30

extern int kmax,kount;
extern float *xp,**yp,dxsav;
```

User storage for intermediate results. Preset `kmax` and `dxsav` in the calling program. If `kmax` $\neq 0$ results are stored at approximate intervals `dxsav` in the arrays `xp[1..kount]`, `yp[1..nvar][1..kount]`, where `kount` is output by `odeint`. Defining declarations for these variables, with memory allocations `xp[1..kmax]` and `yp[1..nvar][1..kmax]` for the arrays, should be in the calling program.

```
void odeint(float ystart[], int nvar, float x1, float x2, float eps, float h1,
    float hmin, int *nok, int *nbad,
    void (*derivs)(float, float [], float []),
    void (*rkqs)(float [], float [], int, float *, float, float, float [],
    float *, float *, void (*)(float, float [], float [])))
```

Runge-Kutta driver with adaptive stepsize control. Integrate starting values `ystart[1..nvar]` from `x1` to `x2` with accuracy `eps`, storing intermediate results in global variables. `h1` should be set as a guessed first stepsize, `hmin` as the minimum allowed stepsize (can be zero). On output `nok` and `nbad` are the number of good and bad (but retried and fixed) steps taken, and `ystart` is replaced by values at the end of the integration interval. `derivs` is the user-supplied routine for calculating the right-hand side derivative, while `rkqs` is the name of the stepper routine to be used.

```
{
    int nstp,i;
    float xsav,x,hnext,hdid,h;
    float *yscal,*y,*dydx;

    yscal=vector(1,nvar);
    y=vector(1,nvar);
    dydx=vector(1,nvar);
    x=x1;
    h=SIGN(h1,x2-x1);
    *nok = (*nbad) = kount = 0;
    for (i=1;i<=nvar;i++) y[i]=ystart[i];
    if (kmax > 0) xsav=x-dxsav*2.0;              Assures storage of first step.
    for (nstp=1;nstp<=MAXSTP;nstp++) {           Take at most MAXSTP steps.
        (*derivs)(x,y,dydx);
        for (i=1;i<=nvar;i++)
            Scaling used to monitor accuracy. This general-purpose choice can be modified
            if need be.
            yscal[i]=fabs(y[i])+fabs(dydx[i]*h)+TINY;
        if (kmax > 0 && kount < kmax-1 && fabs(x-xsav) > fabs(dxsav)) {
            xp[++kount]=x;                       Store intermediate results.
            for (i=1;i<=nvar;i++) yp[i][kount]=y[i];
            xsav=x;
        }
        if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x;     If stepsize can overshoot, decrease.
```

```
        (*rkqs)(y,dydx,nvar,&x,h,eps,yscal,&hdid,&hnext,derivs);
        if (hdid == h) ++(*nok); else ++(*nbad);
        if ((x-x2)*(x2-x1) >= 0.0) {              Are we done?
            for (i=1;i<=nvar;i++) ystart[i]=y[i];
            if (kmax) {
                xp[++kount]=x;                    Save final step.
                for (i=1;i<=nvar;i++) yp[i][kount]=y[i];
            }
            free_vector(dydx,1,nvar);
            free_vector(y,1,nvar);
            free_vector(yscal,1,nvar);
            return;                               Normal exit.
        }
        if (fabs(hnext) <= hmin) nrerror("Step size too small in odeint");
        h=hnext;
    }
    nrerror("Too many steps in routine odeint");
}
```

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Cash, J.R., and Karp, A.H. 1990, *ACM Transactions on Mathematical Software*, vol. 16, pp. 201–222. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall).

16.3 Modified Midpoint Method

This section discusses the *modified midpoint method*, which advances a vector of dependent variables $y(x)$ from a point x to a point $x + H$ by a sequence of n substeps each of size h,

$$h = H/n \tag{16.3.1}$$

In principle, one could use the modified midpoint method in its own right as an ODE integrator. In practice, the method finds its most important application as a part of the more powerful Bulirsch-Stoer technique, treated in §16.4. You can therefore consider this section as a preamble to §16.4.

The number of right-hand side evaluations required by the modified midpoint method is $n + 1$. The formulas for the method are

$$\begin{aligned} z_0 &\equiv y(x) \\ z_1 &= z_0 + hf(x, z_0) \\ z_{m+1} &= z_{m-1} + 2hf(x + mh, z_m) \qquad \text{for} \quad m = 1, 2, \ldots, n-1 \\ y(x+H) \approx y_n &\equiv \frac{1}{2}[z_n + z_{n-1} + hf(x + H, z_n)] \end{aligned} \tag{16.3.2}$$

Here the z's are intermediate approximations which march along in steps of h, while y_n is the final approximation to $y(x+H)$. The method is basically a "centered difference" or "midpoint" method (compare equation 16.1.2), except at the first and last points. Those give the qualifier "modified."

The modified midpoint method is a second-order method, like (16.1.2), but with the advantage of requiring (asymptotically for large n) only one derivative evaluation per step h instead of the two required by second-order Runge-Kutta. Perhaps there are applications where the simplicity of (16.3.2), easily coded in-line in some other program, recommends it. In general, however, use of the modified midpoint method by itself will be dominated by the embedded Runge-Kutta method with adaptive stepsize control, as implemented in the preceding section.

The usefulness of the modified midpoint method to the Bulirsch-Stoer technique (§16.4) derives from a "deep" result about equations (16.3.2), due to Gragg. It turns out that the error of (16.3.2), expressed as a power series in h, the stepsize, contains only *even* powers of h,

$$y_n - y(x+H) = \sum_{i=1}^{\infty} \alpha_i h^{2i} \tag{16.3.3}$$

where H is held constant, but h changes by varying n in (16.3.1). The importance of this even power series is that, if we play our usual tricks of combining steps to knock out higher-order error terms, we can gain *two* orders at a time!

For example, suppose n is even, and let $y_{n/2}$ denote the result of applying (16.3.1) and (16.3.2) with half as many steps, $n \to n/2$. Then the estimate

$$y(x+H) \approx \frac{4y_n - y_{n/2}}{3} \tag{16.3.4}$$

is *fourth-order* accurate, the same as fourth-order Runge-Kutta, but requires only about 1.5 derivative evaluations per step h instead of Runge-Kutta's 4 evaluations. Don't be too anxious to implement (16.3.4), since we will soon do even better.

Now would be a good time to look back at the routine `qsimp` in §4.2, and especially to compare equation (4.2.4) with equation (16.3.4) above. You will see that the transition in Chapter 4 to the idea of Richardson extrapolation, as embodied in Romberg integration of §4.3, is exactly analogous to the transition in going from this section to the next one.

Here is the routine that implements the modified midpoint method, which will be used below.

```
#include "nrutil.h"

void mmid(float y[], float dydx[], int nvar, float xs, float htot, int nstep,
    float yout[], void (*derivs)(float, float[], float[]))
Modified midpoint step. At xs, input the dependent variable vector y[1..nvar] and its deriva-
tive vector dydx[1..nvar]. Also input is htot, the total step to be made, and nstep, the
number of substeps to be used. The output is returned as yout[1..nvar], which need not
be a distinct array from y; if it is distinct, however, then y and dydx are returned undamaged.
{
    int n,i;
    float x,swap,h2,h,*ym,*yn;
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 152 of 181

```
    ym=vector(1,nvar);
    yn=vector(1,nvar);
    h=htot/nstep;                       Stepsize this trip.
    for (i=1;i<=nvar;i++) {
        ym[i]=y[i];
        yn[i]=y[i]+h*dydx[i];           First step.
    }
    x=xs+h;
    (*derivs)(x,yn,yout);               Will use yout for temporary storage of deriva-
    h2=2.0*h;                               tives.
    for (n=2;n<=nstep;n++) {            General step.
        for (i=1;i<=nvar;i++) {
            swap=ym[i]+h2*yout[i];
            ym[i]=yn[i];
            yn[i]=swap;
        }
        x += h;
        (*derivs)(x,yn,yout);
    }
    for (i=1;i<=nvar;i++)               Last step.
        yout[i]=0.5*(ym[i]+yn[i]+h*yout[i]);
    free_vector(yn,1,nvar);
    free_vector(ym,1,nvar);
}
```

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.4.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §7.2.12.

16.4 Richardson Extrapolation and the Bulirsch-Stoer Method

The techniques described in this section are not for differential equations containing nonsmooth functions. For example, you might have a differential equation whose right-hand side involves a function that is evaluated by table look-up and interpolation. If so, go back to Runge-Kutta with adaptive stepsize choice: That method does an excellent job of feeling its way through rocky or discontinuous terrain. It is also an excellent choice for quick-and-dirty, low-accuracy solution of a set of equations. A second warning is that the techniques in this section are not particularly good for differential equations that have singular points *inside* the interval of integration. A regular solution must tiptoe very carefully across such points. Runge-Kutta with adaptive stepsize can sometimes effect this; more generally, there are special techniques available for such problems, beyond our scope here.

Apart from those two caveats, we believe that the Bulirsch-Stoer method, discussed in this section, is the best known way to obtain high-accuracy solutions to ordinary differential equations with minimal computational effort. (A possible exception, infrequently encountered in practice, is discussed in §16.7.)

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 153 of 181

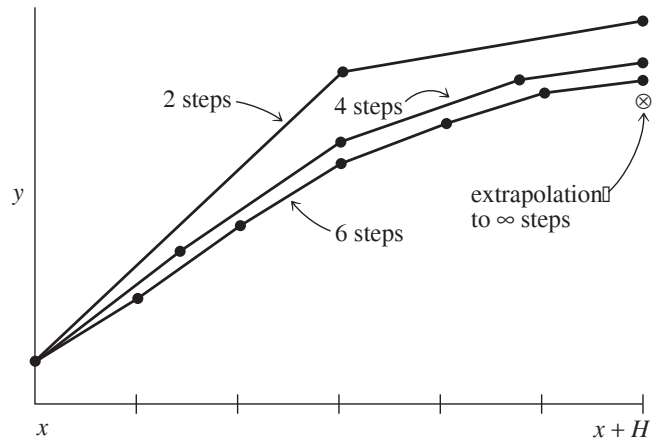

Figure 16.4.1. Richardson extrapolation as used in the Bulirsch-Stoer method. A large interval H is spanned by different sequences of finer and finer substeps. Their results are extrapolated to an answer that is supposed to correspond to infinitely fine substeps. In the Bulirsch-Stoer method, the integrations are done by the modified midpoint method, and the extrapolation technique is rational function or polynomial extrapolation.

Three key ideas are involved. The first is *Richardson's deferred approach to the limit*, which we already met in §4.3 on Romberg integration. The idea is to consider the final answer of a numerical calculation as itself being an analytic function (if a complicated one) of an adjustable parameter like the stepsize h. That analytic function can be probed by performing the calculation with various values of h, *none* of them being necessarily small enough to yield the accuracy that we desire. When we know enough about the function, we *fit* it to some analytic form, and then *evaluate* it at that mythical and golden point $h = 0$ (see Figure 16.4.1). Richardson extrapolation is a method for turning straw into gold! (Lead into gold for alchemist readers.)

The second idea has to do with what kind of fitting function is used. Bulirsch and Stoer first recognized the strength of *rational function extrapolation* in Richardson-type applications. That strength is to break the shackles of the power series and its limited radius of convergence, out only to the distance of the first pole in the complex plane. Rational function fits can remain good approximations to analytic functions even after the various terms in powers of h all have comparable magnitudes. In other words, h can be so large as to make the whole notion of the "order" of the method meaningless — and the method can still work superbly. Nevertheless, more recent experience suggests that for smooth problems straightforward polynomial extrapolation is slightly more efficient than rational function extrapolation. We will accordingly adopt polynomial extrapolation as the default, but the routine `bsstep` below allows easy substitution of one kind of extrapolation for the other. You might wish at this point to review §3.1–§3.2, where polynomial and rational function extrapolation were already discussed.

The third idea was discussed in the section before this one, namely to use a method whose error function is strictly even, allowing the rational function or polynomial approximation to be in terms of the variable h^2 instead of just h.

Put these ideas together and you have the *Bulirsch-Stoer method* [1]. A single Bulirsch-Stoer step takes us from x to $x + H$, where H is supposed to be quite a large

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 154 of 181

— not at all infinitesimal — distance. That single step is a grand leap consisting of many (e.g., dozens to hundreds) substeps of modified midpoint method, which are then extrapolated to zero stepsize.

The sequence of separate attempts to cross the interval H is made with increasing values of n, the number of substeps. Bulirsch and Stoer originally proposed the sequence

$$n = 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, \ldots, [n_j = 2n_{j-2}], \ldots \tag{16.4.1}$$

More recent work by Deuflhard [2,3] suggests that the sequence

$$n = 2, 4, 6, 8, 10, 12, 14, \ldots, [n_j = 2j], \ldots \tag{16.4.2}$$

is usually more efficient. For each step, we do not know in advance how far up this sequence we will go. After each successive n is tried, a polynomial extrapolation is attempted. That extrapolation gives both extrapolated values and error estimates. If the errors are not satisfactory, we go higher in n. If they are satisfactory, we go on to the next step and begin anew with $n = 2$.

Of course there must be some upper limit, beyond which we conclude that there is some obstacle in our path in the interval H, so that we must reduce H rather than just subdivide it more finely. In the implementations below, the maximum number of n's to be tried is called `KMAXX`. For reasons described below we usually take this equal to 8; the 8th value of the sequence (16.4.2) is 16, so this is the maximum number of subdivisions of H that we allow.

We enforce error control, as in the Runge-Kutta method, by monitoring internal consistency, and adapting stepsize to match a prescribed bound on the local truncation error. Each new result from the sequence of modified midpoint integrations allows a tableau like that in §3.1 to be extended by one additional set of diagonals. The size of the new correction added at each stage is taken as the (conservative) error estimate. How should we use this error estimate to adjust the stepsize? The best strategy now known is due to Deuflhard [2,3]. For completeness we describe it here:

Suppose the absolute value of the error estimate returned from the kth column (and hence the $k+1$st row) of the extrapolation tableau is $\epsilon_{k+1,k}$. Error control is enforced by requiring

$$\epsilon_{k+1,k} < \epsilon \tag{16.4.3}$$

as the criterion for accepting the current step, where ϵ is the required tolerance. For the even sequence (16.4.2) the order of the method is $2k+1$:

$$\epsilon_{k+1,k} \sim H^{2k+1} \tag{16.4.4}$$

Thus a simple estimate of a new stepsize H_k to obtain convergence in a fixed column k would be

$$H_k = H\left(\frac{\epsilon}{\epsilon_{k+1,k}}\right)^{1/(2k+1)} \tag{16.4.5}$$

Which column k should we aim to achieve convergence in? Let's compare the work required for different k. Suppose A_k is the work to obtain row k of the extrapolation tableau, so A_{k+1} is the work to obtain column k. We will assume the work is dominated by the cost of evaluating the functions defining the right-hand sides of the differential equations. For n_k subdivisions in H, the number of function evaluations can be found from the recurrence

$$\begin{aligned} A_1 &= n_1 + 1 \\ A_{k+1} &= A_k + n_{k+1} \end{aligned} \tag{16.4.6}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 155 of 181

The work per unit step to get column k is A_{k+1}/H_k, which we nondimensionalize with a factor of H and write as

$$W_k = \frac{A_{k+1}}{H_k} H \tag{16.4.7}$$

$$= A_{k+1} \left(\frac{\epsilon_{k+1,k}}{\epsilon} \right)^{1/(2k+1)} \tag{16.4.8}$$

The quantities W_k can be calculated during the integration. The optimal column index q is then defined by

$$W_q = \min_{k=1,\ldots,k_f} W_k \tag{16.4.9}$$

where k_f is the final column, in which the error criterion (16.4.3) was satisfied. The q determined from (16.4.9) defines the stepsize H_q to be used as the next basic stepsize, so that we can expect to get convergence in the optimal column q.

Two important refinements have to be made to the strategy outlined so far:

- If the current H is "too small," then k_f will be "too small," and so q remains "too small." It may be desirable to increase H and aim for convergence in a column $q > k_f$.
- If the current H is "too big," we may not converge at all on the current step and we will have to decrease H. We would like to detect this by monitoring the quantities $\epsilon_{k+1,k}$ for each k so we can stop the current step as soon as possible.

Deuflhard's prescription for dealing with these two problems uses ideas from communication theory to determine the "average expected convergence behavior" of the extrapolation. His model produces certain correction factors $\alpha(k,q)$ by which H_k is to be multiplied to try to get convergence in column q. The factors $\alpha(k,q)$ depend only on ϵ and the sequence $\{n_i\}$ and so can be computed once during initialization:

$$\alpha(k,q) = \epsilon^{\frac{A_{k+1}-A_{q+1}}{(2k+1)(A_{q+1}-A_1+1)}} \qquad \text{for} \quad k < q \tag{16.4.10}$$

with $\alpha(q,q) = 1$.

Now to handle the first problem, suppose convergence occurs in column $q = k_f$. Then rather than taking H_q for the next step, we might aim to increase the stepsize to get convergence in column $q+1$. Since we don't have H_{q+1} available from the computation, we estimate it as

$$H_{q+1} = H_q \alpha(q, q+1) \tag{16.4.11}$$

By equation (16.4.7) this replacement is efficient, i.e., reduces the work per unit step, if

$$\frac{A_{q+1}}{H_q} > \frac{A_{q+2}}{H_{q+1}} \tag{16.4.12}$$

or

$$A_{q+1}\alpha(q, q+1) > A_{q+2} \tag{16.4.13}$$

During initialization, this inequality can be checked for $q = 1, 2, \ldots$ to determine $k_{\max}$, the largest allowed column. Then when (16.4.12) is satisfied it will always be efficient to use H_{q+1}. (In practice we limit $k_{\max}$ to 8 even when ϵ is very small as there is very little further gain in efficiency whereas roundoff can become a problem.)

The problem of stepsize reduction is handled by computing stepsize estimates

$$\bar{H}_k \equiv H_k \alpha(k,q), \qquad k = 1, \ldots, q-1 \tag{16.4.14}$$

during the current step. The $\bar{H}$'s are estimates of the stepsize to get convergence in the optimal column q. If any $\bar{H}_k$ is "too small," we abandon the current step and restart using $\bar{H}_k$. The criterion of being "too small" is taken to be

$$H_k \alpha(k, q+1) < H \tag{16.4.15}$$

The α's satisfy $\alpha(k, q+1) > \alpha(k,q)$.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 156 of 181

During the first step, when we have no information about the solution, the stepsize reduction check is made for all k. Afterwards, we test for convergence and for possible stepsize reduction only in an "order window"

$$\max(1, q-1) \le k \le \min(k_{\max}, q+1) \tag{16.4.16}$$

The rationale for the order window is that if convergence appears to occur for $k < q-1$ it is often spurious, resulting from some fortuitously small error estimate in the extrapolation. On the other hand, if you need to go beyond $k = q+1$ to obtain convergence, your local model of the convergence behavior is obviously not very good and you need to cut the stepsize and reestablish it.

In the routine `bsstep`, these various tests are actually carried out using quantities

$$\epsilon(k) \equiv \frac{H}{H_k} = \left(\frac{\epsilon_{k+1,k}}{\epsilon}\right)^{1/(2k+1)} \tag{16.4.17}$$

called `err[k]` in the code. As usual, we include a "safety factor" in the stepsize selection. This is implemented by replacing ϵ by 0.25ϵ. Other safety factors are explained in the program comments.

Note that while the optimal convergence column is restricted to increase by at most one on each step, a sudden drop in order is allowed by equation (16.4.9). This gives the method a degree of robustness for problems with discontinuities.

Let us remind you once again that *scaling* of the variables is often crucial for successful integration of differential equations. The scaling "trick" suggested in the discussion following equation (16.2.8) is a good general purpose choice, but not foolproof. Scaling by the maximum values of the variables is more robust, but requires you to have some prior information.

The following implementation of a Bulirsch-Stoer step has exactly the same calling sequence as the quality-controlled Runge-Kutta stepper `rkqs`. This means that the driver `odeint` in §16.2 can be used for Bulirsch-Stoer as well as Runge-Kutta: Just substitute `bsstep` for `rkqs` in `odeint`'s argument list. The routine `bsstep` calls `mmid` to take the modified midpoint sequences, and calls `pzextr`, given below, to do the polynomial extrapolation.

```
#include <math.h>
#include "nrutil.h"
#define KMAXX 8                    Maximum row number used in the extrapola-
#define IMAXX (KMAXX+1)                tion.
#define SAFE1 0.25                 Safety factors.
#define SAFE2 0.7
#define REDMAX 1.0e-5              Maximum factor for stepsize reduction.
#define REDMIN 0.7                 Minimum factor for stepsize reduction.
#define TINY 1.0e-30               Prevents division by zero.
#define SCALMX 0.1                 1/SCALMX is the maximum factor by which a
                                       stepsize can be increased.
float **d,*x;
```

Pointers to matrix and vector used by `pzextr` or `rzextr`.

```
void bsstep(float y[], float dydx[], int nv, float *xx, float htry, float eps,
    float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []))
```

Bulirsch-Stoer step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector `y[1..nv]` and its derivative `dydx[1..nv]` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal[1..nv]` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is the user-supplied routine that computes the right-hand side derivatives. Be sure to set `htry` on successive steps

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 157 of 181

to the value of hnext returned from the previous step, as is the case if the routine is called by odeint.

```
{
    void mmid(float y[], float dydx[], int nvar, float xs, float htot,
        int nstep, float yout[], void (*derivs)(float, float[], float[]));
    void pzextr(int iest, float xest, float yest[], float yz[], float dy[],
        int nv);
    int i,iq,k,kk,km;
    static int first=1,kmax,kopt;
    static float epsold = -1.0,xnew;
    float eps1,errmax,fact,h,red,scale,work,wrkmin,xest;
    float *err,*yerr,*ysav,*yseq;
    static float a[IMAXX+1];
    static float alf[KMAXX+1][KMAXX+1];
    static int nseq[IMAXX+1]={0,2,4,6,8,10,12,14,16,18};
    int reduct,exitflag=0;

    d=matrix(1,nv,1,KMAXX);
    err=vector(1,KMAXX);
    x=vector(1,KMAXX);
    yerr=vector(1,nv);
    ysav=vector(1,nv);
    yseq=vector(1,nv);
    if (eps != epsold) {                          A new tolerance, so reinitialize.
        *hnext = xnew = -1.0e29;                  "Impossible" values.
        eps1=SAFE1*eps;
        a[1]=nseq[1]+1;                           Compute work coefficients A_k.
        for (k=1;k<=KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
        for (iq=2;iq<=KMAXX;iq++) {               Compute α(k,q).
            for (k=1;k<iq;k++)
                alf[k][iq]=pow(eps1,(a[k+1]-a[iq+1])/
                    ((a[iq+1]-a[1]+1.0)*(2*k+1)));
        }
        epsold=eps;
        for (kopt=2;kopt<KMAXX;kopt++)            Determine optimal row number for
            if (a[kopt+1] > a[kopt]*alf[kopt-1][kopt]) break;      convergence.
        kmax=kopt;
    }
    h=htry;
    for (i=1;i<=nv;i++) ysav[i]=y[i];             Save the starting values.
    if (*xx != xnew || h != (*hnext)) {           A new stepsize or a new integration:
        first=1;                                      re-establish the order window.
        kopt=kmax;
    }
    reduct=0;
    for (;;) {
        for (k=1;k<=kmax;k++) {                   Evaluate the sequence of modified
            xnew=(*xx)+h;                             midpoint integrations.
            if (xnew == (*xx)) nrerror("step size underflow in bsstep");
            mmid(ysav,dydx,nv,*xx,h,nseq[k],yseq,derivs);
            xest=SQR(h/nseq[k]);                  Squared, since error series is even.
            pzextr(k,xest,yseq,y,yerr,nv);        Perform extrapolation.
            if (k != 1) {                         Compute normalized error estimate
                errmax=TINY;                          ε(k).
                for (i=1;i<=nv;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
                errmax /= eps;                    Scale error relative to tolerance.
                km=k-1;
                err[km]=pow(errmax/SAFE1,1.0/(2*km+1));
            }
            if (k != 1 && (k >= kopt-1 || first)) {      In order window.
                if (errmax < 1.0) {                      Converged.
                    exitflag=1;
                    break;
                }
```

```
                if (k == kmax || k == kopt+1) {          Check for possible stepsize
                    red=SAFE2/err[km];                       reduction.
                    break;
                }
                else if (k == kopt && alf[kopt-1][kopt] < err[km]) {
                        red=1.0/err[km];
                        break;
                    }
                else if (kopt == kmax && alf[km][kmax-1] < err[km]) {
                        red=alf[km][kmax-1]*SAFE2/err[km];
                        break;
                    }
                else if (alf[km][kopt] < err[km]) {
                    red=alf[km][kopt-1]/err[km];
                    break;
                }
            }
        }
        if (exitflag) break;
        red=FMIN(red,REDMIN);                      Reduce stepsize by at least REDMIN
        red=FMAX(red,REDMAX);                          and at most REDMAX.
        h *= red;
        reduct=1;
    }                                              Try again.
    *xx=xnew;                                      Successful step taken.
    *hdid=h;
    first=0;
    wrkmin=1.0e35;                                 Compute optimal row for convergence
    for (kk=1;kk<=km;kk++) {                           and corresponding stepsize.
        fact=FMAX(err[kk],SCALMX);
        work=fact*a[kk+1];
        if (work < wrkmin) {
            scale=fact;
            wrkmin=work;
            kopt=kk+1;
        }
    }
    *hnext=h/scale;
    if (kopt >= k && kopt != kmax && !reduct) {
    Check for possible order increase, but not if stepsize was just reduced.
        fact=FMAX(scale/alf[kopt-1][kopt],SCALMX);
        if (a[kopt+1]*fact <= wrkmin) {
            *hnext=h/fact;
            kopt++;
        }
    }
    free_vector(yseq,1,nv);
    free_vector(ysav,1,nv);
    free_vector(yerr,1,nv);
    free_vector(x,1,KMAXX);
    free_vector(err,1,KMAXX);
    free_matrix(d,1,nv,1,KMAXX);
}
```

The polynomial extrapolation routine is based on the same algorithm as `polint` §3.1. It is simpler in that it is always extrapolating to zero, rather than to an arbitrary value. However, it is more complicated in that it must individually extrapolate each component of a vector of quantities.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 159 of 181

```
#include "nrutil.h"

extern float **d,*x;                        Defined in bsstep.

void pzextr(int iest, float xest, float yest[], float yz[], float dy[], int nv)
Use polynomial extrapolation to evaluate nv functions at x = 0 by fitting a polynomial to a
sequence of estimates with progressively smaller values x = xest, and corresponding function
vectors yest[1..nv]. This call is number iest in the sequence of calls. Extrapolated function
values are output as yz[1..nv], and their estimated error is output as dy[1..nv].
{
    int k1,j;
    float q,f2,f1,delta,*c;

    c=vector(1,nv);
    x[iest]=xest;                           Save current independent variable.
    for (j=1;j<=nv;j++) dy[j]=yz[j]=yest[j];
    if (iest == 1) {                        Store first estimate in first column.
        for (j=1;j<=nv;j++) d[j][1]=yest[j];
    } else {
        for (j=1;j<=nv;j++) c[j]=yest[j];
        for (k1=1;k1<iest;k1++) {
            delta=1.0/(x[iest-k1]-xest);
            f1=xest*delta;
            f2=x[iest-k1]*delta;
            for (j=1;j<=nv;j++) {           Propagate tableau 1 diagonal more.
                q=d[j][k1];
                d[j][k1]=dy[j];
                delta=c[j]-q;
                dy[j]=f1*delta;
                c[j]=f2*delta;
                yz[j] += dy[j];
            }
        }
        for (j=1;j<=nv;j++) d[j][iest]=dy[j];
    }
    free_vector(c,1,nv);
}
```

Current wisdom favors polynomial extrapolation over rational function extrapolation in the Bulirsch-Stoer method. However, our feeling is that this view is guided more by the kinds of problems used for tests than by one method being actually "better." Accordingly, we provide the optional routine `rzextr` for rational function extrapolation, an exact substitution for `pzextr` above.

```
#include "nrutil.h"

extern float **d,*x;                    Defined in bsstep.

void rzextr(int iest, float xest, float yest[], float yz[], float dy[], int nv)
Exact substitute for pzextr, but uses diagonal rational function extrapolation instead of poly-
nomial extrapolation.
{
    int k,j;
    float yy,v,ddy,c,b1,b,*fx;

    fx=vector(1,iest);
    x[iest]=xest;                       Save current independent variable.
    if (iest == 1)
        for (j=1;j<=nv;j++) {
            yz[j]=yest[j];
            d[j][1]=yest[j];
```

```
            dy[j]=yest[j];
        }
    else {
        for (k=1;k<iest;k++)
            fx[k+1]=x[iest-k]/xest;
        for (j=1;j<=nv;j++) {          Evaluate next diagonal in tableau.
            v=d[j][1];
            d[j][1]=yy=c=yest[j];
            for (k=2;k<=iest;k++) {
                b1=fx[k]*v;
                b=b1-c;
                if (b) {
                    b=(c-v)/b;
                    ddy=c*b;
                    c=b1*b;
                } else                 Care needed to avoid division by 0.
                    ddy=v;
                if (k != iest) v=d[j][k];
                d[j][k]=ddy;
                yy += ddy;
            }
            dy[j]=ddy;
            yz[j]=yy;
        }
    }
    free_vector(fx,1,iest);
}
```

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §7.2.14. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.2.

Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 399–422. [2]

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535. [3]

16.5 Second-Order Conservative Equations

Usually when you have a system of high-order differential equations to solve it is best to reformulate them as a system of first-order equations, as discussed in §16.0. There is a particular class of equations that occurs quite frequently in practice where you can gain about a factor of two in efficiency by differencing the equations directly. The equations are second-order systems where the derivative does not appear on the right-hand side:

$$y'' = f(x, y), \qquad y(x_0) = y_0, \qquad y'(x_0) = z_0 \tag{16.5.1}$$

As usual, y can denote a vector of values.

Stoermer's rule, dating back to 1907, has been a popular method for discretizing such systems. With $h = H/m$ we have

$$\begin{aligned} y_1 &= y_0 + h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_{k+1} - 2y_k + y_{k-1} &= h^2 f(x_0 + kh, y_k), \qquad k = 1, \ldots, m-1 \\ z_m &= (y_m - y_{m-1})/h + \tfrac{1}{2}hf(x_0 + H, y_m) \end{aligned} \tag{16.5.2}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 161 of 181

Here z_m is $y'(x_0 + H)$. Henrici showed how to rewrite equations (16.5.2) to reduce roundoff error by using the quantities $\Delta_k \equiv y_{k+1} - y_k$. Start with

$$\begin{aligned} \Delta_0 &= h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_1 &= y_0 + \Delta_0 \end{aligned} \tag{16.5.3}$$

Then for $k = 1, \ldots, m-1$, set

$$\begin{aligned} \Delta_k &= \Delta_{k-1} + h^2 f(x_0 + kh, y_k) \\ y_{k+1} &= y_k + \Delta_k \end{aligned} \tag{16.5.4}$$

Finally compute the derivative from

$$z_m = \Delta_{m-1}/h + \tfrac{1}{2}hf(x_0 + H, y_m) \tag{16.5.5}$$

Gragg again showed that the error series for equations (16.5.3)–(16.5.5) contains only even powers of h, and so the method is a logical candidate for extrapolation à la Bulirsch-Stoer. We replace `mmid` by the following routine `stoerm`:

```
#include "nrutil.h"

void stoerm(float y[], float d2y[], int nv, float xs, float htot, int nstep,
    float yout[], void (*derivs)(float, float [], float []))
```

Stoermer's rule for integrating $y'' = f(x, y)$ for a system of $n = \mathtt{nv}/2$ equations. On input `y[1..nv]` contains y in its first n elements and y' in its second n elements, all evaluated at `xs`. `d2y[1..nv]` contains the right-hand side function f (also evaluated at `xs`) in its first n elements. Its second n elements are not referenced. Also input is `htot`, the total step to be taken, and `nstep`, the number of substeps to be used. The output is returned as `yout[1..nv]`, with the same storage arrangement as `y`. `derivs` is the user-supplied routine that calculates f.

```
{
    int i,n,neqns,nn;
    float h,h2,halfh,x,*ytemp;

    ytemp=vector(1,nv);
    h=htot/nstep;                       Stepsize this trip.
    halfh=0.5*h;
    neqns=nv/2;                         Number of equations.
    for (i=1;i<=neqns;i++) {            First step.
        n=neqns+i;
        ytemp[i]=y[i]+(ytemp[n]=h*(y[n]+halfh*d2y[i]));
    }
    x=xs+h;
    (*derivs)(x,ytemp,yout);            Use yout for temporary storage of derivatives.
    h2=h*h;
    for (nn=2;nn<=nstep;nn++) {         General step.
        for (i=1;i<=neqns;i++)
            ytemp[i] += (ytemp[(n=neqns+i)] += h2*yout[i]);
        x += h;
        (*derivs)(x,ytemp,yout);
    }
    for (i=1;i<=neqns;i++) {            Last step.
        n=neqns+i;
        yout[n]=ytemp[n]/h+halfh*yout[i];
        yout[i]=ytemp[i];
    }
    free_vector(ytemp,1,nv);
}
```

Note that for compatibility with `bsstep` the arrays `y` and `d2y` are of length $2n$ for a system of n second-order equations. The values of y are stored in the first n elements of `y`, while the first derivatives are stored in the second n elements. The right-hand side f is stored in the first n elements of the array `d2y`; the second n elements are unused. With this storage arrangement you can use `bsstep` simply by replacing the call to `mmid` with one to `stoerm` using the same arguments; just be sure that the argument `nv` of `bsstep` is set to $2n$. You should also use the more efficient sequence of stepsizes suggested by Deuflhard:

$$n = 1, 2, 3, 4, 5, \ldots \tag{16.5.6}$$

and set `KMAXX` $= 12$ in `bsstep`.

CITED REFERENCES AND FURTHER READING:
Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535.

16.6 Stiff Sets of Equations

As soon as one deals with more than one first-order differential equation, the possibility of a *stiff* set of equations arises. Stiffness occurs in a problem where there are two or more very different scales of the independent variable on which the dependent variables are changing. For example, consider the following set of equations [1]:

$$\begin{aligned} u' &= 998u + 1998v \\ v' &= -999u - 1999v \end{aligned} \tag{16.6.1}$$

with boundary conditions

$$u(0) = 1 \qquad v(0) = 0 \tag{16.6.2}$$

By means of the transformation

$$u = 2y - z \qquad v = -y + z \tag{16.6.3}$$

we find the solution

$$\begin{aligned} u &= 2e^{-x} - e^{-1000x} \\ v &= -e^{-x} + e^{-1000x} \end{aligned} \tag{16.6.4}$$

If we integrated the system (16.6.1) with any of the methods given so far in this chapter, the presence of the e^{-1000x} term would require a stepsize $h \ll 1/1000$ for the method to be stable (the reason for this is explained below). This is so even

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 163 of 181

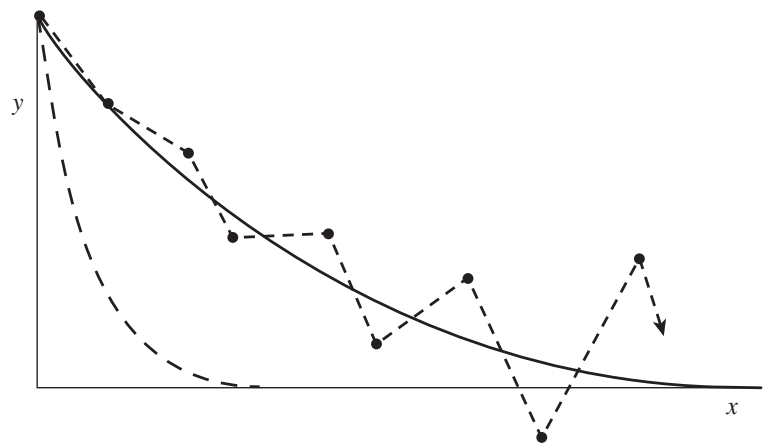

Figure 16.6.1. Example of an instability encountered in integrating a stiff equation (schematic). Here it is supposed that the equation has two solutions, shown as solid and dashed lines. Although the initial conditions are such as to give the solid solution, the stability of the integration (shown as the unstable dotted sequence of segments) is determined by the more rapidly varying dashed solution, even after that solution has effectively died away to zero. Implicit integration methods are the cure.

though the e^{-1000x} term is completely negligible in determining the values of u and v as soon as one is away from the origin (see Figure 16.6.1).

This is the generic disease of stiff equations: we are required to follow the variation in the solution on the shortest length scale to maintain stability of the integration, even though accuracy requirements allow a much larger stepsize.

To see how we might cure this problem, consider the single equation

$$y' = -cy \tag{16.6.5}$$

where $c > 0$ is a constant. The explicit (or *forward*) Euler scheme for integrating this equation with stepsize h is

$$y_{n+1} = y_n + hy'_n = (1 - ch)y_n \tag{16.6.6}$$

The method is called explicit because the new value y_{n+1} is given explicitly in terms of the old value y_n. Clearly the method is unstable if $h > 2/c$, for then $|y_n| \to \infty$ as $n \to \infty$.

The simplest cure is to resort to *implicit* differencing, where the right-hand side is evaluated at the *new* y location. In this case, we get the *backward Euler* scheme:

$$y_{n+1} = y_n + hy'_{n+1} \tag{16.6.7}$$

or

$$y_{n+1} = \frac{y_n}{1 + ch} \tag{16.6.8}$$

The method is absolutely stable: even as $h \to \infty$, $y_{n+1} \to 0$, which is in fact the correct solution of the differential equation. If we think of x as representing time, then the implicit method converges to the true equilibrium solution (i.e., the solution at late times) for large stepsizes. This nice feature of implicit methods holds only for linear systems, but even in the general case implicit methods give better stability.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 164 of 181

Of course, we give up *accuracy* in following the evolution towards equilibrium if we use large stepsizes, but we maintain *stability*.

These considerations can easily be generalized to sets of linear equations with constant coefficients:

$$\mathbf{y}' = -\mathbf{C} \cdot \mathbf{y} \tag{16.6.9}$$

where $\mathbf{C}$ is a positive definite matrix. Explicit differencing gives

$$\mathbf{y}_{n+1} = (\mathbf{1} - \mathbf{C}h) \cdot \mathbf{y}_n \tag{16.6.10}$$

Now a matrix $\mathbf{A}^n$ tends to zero as $n \to \infty$ only if the largest eigenvalue of $\mathbf{A}$ has magnitude less than unity. Thus $\mathbf{y}_n$ is bounded as $n \to \infty$ only if the largest eigenvalue of $\mathbf{1} - \mathbf{C}h$ is less than 1, or in other words

$$h < \frac{2}{\lambda_{\max}} \tag{16.6.11}$$

where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{C}$.

On the other hand, implicit differencing gives

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{y}'_{n+1} \tag{16.6.12}$$

or

$$\mathbf{y}_{n+1} = (\mathbf{1} + \mathbf{C}h)^{-1} \cdot \mathbf{y}_n \tag{16.6.13}$$

If the eigenvalues of $\mathbf{C}$ are λ, then the eigenvalues of $(\mathbf{1} + \mathbf{C}h)^{-1}$ are $(1 + \lambda h)^{-1}$, which has magnitude less than one for all h. (Recall that all the eigenvalues of a positive definite matrix are nonnegative.) Thus the method is stable for all stepsizes h. The penalty we pay for this stability is that we are required to invert a matrix at each step.

Not all equations are linear with constant coefficients, unfortunately! For the system

$$\mathbf{y}' = \mathbf{f}(\mathbf{y}) \tag{16.6.14}$$

implicit differencing gives

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(\mathbf{y}_{n+1}) \tag{16.6.15}$$

In general this is some nasty set of nonlinear equations that has to be solved iteratively at each step. Suppose we try linearizing the equations, as in Newton's method:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\mathbf{f}(\mathbf{y}_n) + \left.\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right|_{\mathbf{y}_n} \cdot (\mathbf{y}_{n+1} - \mathbf{y}_n)\right] \tag{16.6.16}$$

Here $\partial\mathbf{f}/\partial\mathbf{y}$ is the matrix of the partial derivatives of the right-hand side (the Jacobian matrix). Rearrange equation (16.6.16) into the form

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot \mathbf{f}(\mathbf{y}_n) \tag{16.6.17}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 165 of 181

If h is not too big, only one iteration of Newton's method may be accurate enough to solve equation (16.6.15) using equation (16.6.17). In other words, at each step we have to invert the matrix

$$\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}} \tag{16.6.18}$$

to find $\mathbf{y}_{n+1}$. Solving implicit methods by linearization is called a "semi-implicit" method, so equation (16.6.17) is the *semi-implicit Euler method*. It is not guaranteed to be stable, but it usually is, because the behavior is locally similar to the case of a constant matrix $\mathbf{C}$ described above.

So far we have dealt only with implicit methods that are first-order accurate. While these are very robust, most problems will benefit from higher-order methods. There are three important classes of higher-order methods for stiff systems:

- Generalizations of the Runge-Kutta method, of which the most useful are the Rosenbrock methods. The first practical implementation of these ideas was by Kaps and Rentrop, and so these methods are also called Kaps-Rentrop methods.
- Generalizations of the Bulirsch-Stoer method, in particular a semi-implicit extrapolation method due to Bader and Deuflhard.
- Predictor-corrector methods, most of which are descendants of Gear's backward differentiation method.

We shall give implementations of the first two methods. Note that systems where the right-hand side depends explicitly on x, $\mathbf{f}(\mathbf{y}, x)$, can be handled by adding x to the list of dependent variables so that the system to be solved is

$$\begin{pmatrix} \mathbf{y} \\ x \end{pmatrix}' = \begin{pmatrix} \mathbf{f} \\ 1 \end{pmatrix} \tag{16.6.19}$$

In both the routines to be given in this section, we have explicitly carried out this replacement for you, so the routines can handle right-hand sides of the form $\mathbf{f}(\mathbf{y}, x)$ without any special effort on your part.

We now mention an important point: *It is absolutely crucial to scale your variables properly when integrating stiff problems with automatic stepsize adjustment.* As in our nonstiff routines, you will be asked to supply a vector $\mathbf{y}_{\rm scal}$ with which the error is to be scaled. For example, to get constant fractional errors, simply set $\mathbf{y}_{\rm scal} = |\mathbf{y}|$. You can get constant absolute errors relative to some maximum values by setting $\mathbf{y}_{\rm scal}$ equal to those maximum values. In stiff problems, there are often strongly decreasing pieces of the solution which you are not particularly interested in following once they are small. You can control the relative error above some threshold $\mathbf{C}$ and the absolute error below the threshold by setting

$$\mathbf{y}_{\rm scal} = \max(\mathbf{C}, |\mathbf{y}|) \tag{16.6.20}$$

If you are using appropriate nondimensional units, then each component of $\mathbf{C}$ should be of order unity. If you are not sure what values to take for $\mathbf{C}$, simply try setting each component equal to unity. *We strongly advocate the choice (16.6.20) for stiff problems.*

One final warning: Solving stiff problems can sometimes lead to catastrophic precision loss. Be alert for situations where double precision is necessary.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 166 of 181

Rosenbrock Methods

These methods have the advantage of being relatively simple to understand and implement. For moderate accuracies ($\epsilon \lesssim 10^{-4} - 10^{-5}$ in the error criterion) and moderate-sized systems ($N \lesssim 10$), they are competitive with the more complicated algorithms. For more stringent parameters, Rosenbrock methods remain reliable; they merely become less efficient than competitors like the semi-implicit extrapolation method (see below).

A Rosenbrock method seeks a solution of the form

$$\mathbf{y}(x_0 + h) = \mathbf{y}_0 + \sum_{i=1}^{s} c_i \mathbf{k}_i \tag{16.6.21}$$

where the corrections $\mathbf{k}_i$ are found by solving s linear equations that generalize the structure in (16.6.17):

$$(\mathbf{1} - \gamma h \mathbf{f}') \cdot \mathbf{k}_i = h\mathbf{f}\left(\mathbf{y}_0 + \sum_{j=1}^{i-1} \alpha_{ij}\mathbf{k}_j\right) + h\mathbf{f}' \cdot \sum_{j=1}^{i-1} \gamma_{ij}\mathbf{k}_j, \qquad i = 1, \ldots, s \tag{16.6.22}$$

Here we denote the Jacobian matrix by $\mathbf{f}'$. The coefficients γ, c_i, α_{ij}, and γ_{ij} are fixed constants independent of the problem. If $\gamma = \gamma_{ij} = 0$, this is simply a Runge-Kutta scheme. Equations (16.6.22) can be solved successively for $\mathbf{k}_1, \mathbf{k}_2, \ldots$.

Crucial to the success of a stiff integration scheme is an automatic stepsize adjustment algorithm. Kaps and Rentrop [2] discovered an *embedded* or Runge-Kutta-Fehlberg method as described in §16.2: Two estimates of the form (16.6.21) are computed, the "real" one $\mathbf{y}$ and a lower-order estimate $\widehat{\mathbf{y}}$ with different coefficients $\hat{c}_i, i = 1, \ldots, \hat{s}$, where $\hat{s} < s$ but the $\mathbf{k}_i$ are the same. The difference between $\mathbf{y}$ and $\widehat{\mathbf{y}}$ leads to an estimate of the local truncation error, which can then be used for stepsize control. Kaps and Rentrop showed that the smallest value of s for which embedding is possible is $s = 4$, $\hat{s} = 3$, leading to a fourth-order method.

To minimize the matrix-vector multiplications on the right-hand side of (16.6.22), we rewrite the equations in terms of quantities

$$\mathbf{g}_i = \sum_{j=1}^{i-1} \gamma_{ij}\mathbf{k}_j + \gamma\mathbf{k}_i \tag{16.6.23}$$

The equations then take the form

$$\begin{aligned}
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_1 &= \mathbf{f}(\mathbf{y}_0) \\
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_2 &= \mathbf{f}(\mathbf{y}_0 + a_{21}\mathbf{g}_1) + c_{21}\mathbf{g}_1/h \\
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_3 &= \mathbf{f}(\mathbf{y}_0 + a_{31}\mathbf{g}_1 + a_{32}\mathbf{g}_2) + (c_{31}\mathbf{g}_1 + c_{32}\mathbf{g}_2)/h \\
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_4 &= \mathbf{f}(\mathbf{y}_0 + a_{41}\mathbf{g}_1 + a_{42}\mathbf{g}_2 + a_{43}\mathbf{g}_3) + (c_{41}\mathbf{g}_1 + c_{42}\mathbf{g}_2 + c_{43}\mathbf{g}_3)/h
\end{aligned} \tag{16.6.24}$$

In our implementation `stiff` of the Kaps-Rentrop algorithm, we have carried out the replacement (16.6.19) explicitly in equations (16.6.24), so you need not concern yourself about it. Simply provide a routine (called `derivs` in `stiff`) that returns $\mathbf{f}$ (called `dydx`) as a function of x and $\mathbf{y}$. Also supply a routine `jacobn` that returns $\mathbf{f}'$ (`dfdy`) and $\partial\mathbf{f}/\partial x$ (`dfdx`) as functions of x and $\mathbf{y}$. If x does not occur explicitly on the right-hand side, then `dfdx` will be zero. Usually the Jacobian matrix will be available to you by analytic differentiation of the right-hand side $\mathbf{f}$. If not, your routine will have to compute it by numerical differencing with appropriate increments $\Delta\mathbf{y}$.

Kaps and Rentrop gave two different sets of parameters, which have slightly different stability properties. Several other sets have been proposed. Our default choice is that of Shampine [3], but we also give you one of the Kaps-Rentrop sets as an option. Some proposed parameter sets require function evaluations outside the domain of integration; we prefer to avoid that complication.

The calling sequence of `stiff` is exactly the same as the nonstiff routines given earlier in this chapter. It is thus "plug-compatible" with them in the general ODE integrating routine

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 167 of 181

odeint. This compatibility requires, unfortunately, one slight anomaly: While the user-supplied routine derivs is a dummy argument (which can therefore have any actual name), the other user-supplied routine is *not* an argument and must be named (exactly) jacobn.

stiff begins by saving the initial values, in case the step has to be repeated because the error tolerance is exceeded. The linear equations (16.6.24) are solved by first computing the LU decomposition of the matrix $\mathbf{1}/\gamma h - \mathbf{f}'$ using the routine ludcmp. Then the four $\mathbf{g}_i$ are found by back-substitution of the four different right-hand sides using lubksb. Note that each step of the integration requires one call to jacobn and three calls to derivs (one call to get dydx before calling stiff, and two calls inside stiff). The reason only three calls are needed and not four is that the parameters have been chosen so that the last two calls in equation (16.6.24) are done with the same arguments. Counting the evaluation of the Jacobian matrix as roughly equivalent to N evaluations of the right-hand side $\mathbf{f}$, we see that the Kaps-Rentrop scheme involves about $N + 3$ function evaluations per step. Note that if N is large and the Jacobian matrix is sparse, you should replace the LU decomposition by a suitable sparse matrix procedure.

Stepsize control depends on the fact that

$$\begin{aligned} \mathbf{y}_{\text{exact}} &= \mathbf{y} + O(h^5) \\ \mathbf{y}_{\text{exact}} &= \widehat{\mathbf{y}} + O(h^4) \end{aligned} \tag{16.6.25}$$

Thus

$$|\mathbf{y} - \widehat{\mathbf{y}}| = O(h^4) \tag{16.6.26}$$

Referring back to the steps leading from equation (16.2.4) to equation (16.2.10), we see that the new stepsize should be chosen as in equation (16.2.10) but with the exponents 1/4 and 1/5 replaced by 1/3 and 1/4, respectively. Also, experience shows that it is wise to prevent too large a stepsize change in one step, otherwise we will probably have to undo the large change in the next step. We adopt 0.5 and 1.5 as the maximum allowed decrease and increase of h in one step.

```
#include <math.h>
#include "nrutil.h"
#define SAFETY 0.9
#define GROW 1.5
#define PGROW -0.25
#define SHRNK 0.5
#define PSHRNK (-1.0/3.0)
#define ERRCON 0.1296
#define MAXTRY 40
```

Here NMAX is the maximum value of n; GROW and SHRNK are the largest and smallest factors by which stepsize can change in one step; ERRCON equals (GROW/SAFETY) raised to the power (1/PGROW) and handles the case when errmax $\simeq 0$.

```
#define GAM (1.0/2.0)
#define A21 2.0
#define A31 (48.0/25.0)
#define A32 (6.0/25.0)
#define C21 -8.0
#define C31 (372.0/25.0)
#define C32 (12.0/5.0)
#define C41 (-112.0/125.0)
#define C42 (-54.0/125.0)
#define C43 (-2.0/5.0)
#define B1 (19.0/9.0)
#define B2 (1.0/2.0)
#define B3 (25.0/108.0)
#define B4 (125.0/108.0)
#define E1 (17.0/54.0)
#define E2 (7.0/36.0)
#define E3 0.0
#define E4 (125.0/108.0)
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 168 of 181

```
#define C1X (1.0/2.0)
#define C2X (-3.0/2.0)
#define C3X (121.0/50.0)
#define C4X (29.0/250.0)
#define A2X 1.0
#define A3X (3.0/5.0)

void stiff(float y[], float dydx[], int n, float *x, float htry, float eps,
    float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []))
```

Fourth-order Rosenbrock step for integrating stiff o.d.e.'s, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector `y[1..n]` and its derivative `dydx[1..n]` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal[1..n]` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is a user-supplied routine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied routine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`.

```
{
    void jacobn(float x, float y[], float dfdx[], float **dfdy, int n);
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,j,jtry,*indx;
    float d,errmax,h,xsav,**a,*dfdx,**dfdy,*dysav,*err;
    float *g1,*g2,*g3,*g4,*ysav;

    indx=ivector(1,n);
    a=matrix(1,n,1,n);
    dfdx=vector(1,n);
    dfdy=matrix(1,n,1,n);
    dysav=vector(1,n);
    err=vector(1,n);
    g1=vector(1,n);
    g2=vector(1,n);
    g3=vector(1,n);
    g4=vector(1,n);
    ysav=vector(1,n);
    xsav=(*x);                      Save initial values.
    for (i=1;i<=n;i++) {
        ysav[i]=y[i];
        dysav[i]=dydx[i];
    }
    jacobn(xsav,ysav,dfdx,dfdy,n);
    The user must supply this routine to return the n-by-n matrix dfdy and the vector dfdx.
    h=htry;                         Set stepsize to the initial trial value.
    for (jtry=1;jtry<=MAXTRY;jtry++) {
        for (i=1;i<=n;i++) {        Set up the matrix 1 - γhf'.
            for (j=1;j<=n;j++) a[i][j] = -dfdy[i][j];
            a[i][i] += 1.0/(GAM*h);
        }
        ludcmp(a,n,indx,&d);        LU decomposition of the matrix.
        for (i=1;i<=n;i++)          Set up right-hand side for g1.
            g1[i]=dysav[i]+h*C1X*dfdx[i];
        lubksb(a,n,indx,g1);        Solve for g1.
        for (i=1;i<=n;i++)          Compute intermediate values of y and x.
            y[i]=ysav[i]+A21*g1[i];
        *x=xsav+A2X*h;
        (*derivs)(*x,y,dydx);       Compute dydx at the intermediate values.
        for (i=1;i<=n;i++)          Set up right-hand side for g2.
            g2[i]=dydx[i]+h*C2X*dfdx[i]+C21*g1[i]/h;
        lubksb(a,n,indx,g2);        Solve for g2.
        for (i=1;i<=n;i++)          Compute intermediate values of y and x.
            y[i]=ysav[i]+A31*g1[i]+A32*g2[i];
```

```
        *x=xsav+A3X*h;
        (*derivs)(*x,y,dydx);        Compute dydx at the intermediate values.
        for (i=1;i<=n;i++)           Set up right-hand side for g3.
            g3[i]=dydx[i]+h*C3X*dfdx[i]+(C31*g1[i]+C32*g2[i])/h;
        lubksb(a,n,indx,g3);         Solve for g3.
        for (i=1;i<=n;i++)           Set up right-hand side for g4.
            g4[i]=dydx[i]+h*C4X*dfdx[i]+(C41*g1[i]+C42*g2[i]+C43*g3[i])/h;
        lubksb(a,n,indx,g4);         Solve for g4.
        for (i=1;i<=n;i++) {         Get fourth-order estimate of y and error estimate.
            y[i]=ysav[i]+B1*g1[i]+B2*g2[i]+B3*g3[i]+B4*g4[i];
            err[i]=E1*g1[i]+E2*g2[i]+E3*g3[i]+E4*g4[i];
        }
        *x=xsav+h;
        if (*x == xsav) nrerror("stepsize not significant in stiff");
        errmax=0.0;                  Evaluate accuracy.
        for (i=1;i<=n;i++) errmax=FMAX(errmax,fabs(err[i]/yscal[i]));
        errmax /= eps;               Scale relative to required tolerance.
        if (errmax <= 1.0) {         Step succeeded. Compute size of next step and re-
            *hdid=h;                     turn.
            *hnext=(errmax > ERRCON ? SAFETY*h*pow(errmax,PGROW) : GROW*h);
            free_vector(ysav,1,n);
            free_vector(g4,1,n);
            free_vector(g3,1,n);
            free_vector(g2,1,n);
            free_vector(g1,1,n);
            free_vector(err,1,n);
            free_vector(dysav,1,n);
            free_matrix(dfdy,1,n,1,n);
            free_vector(dfdx,1,n);
            free_matrix(a,1,n,1,n);
            free_ivector(indx,1,n);
            return;
        } else {                     Truncation error too large, reduce stepsize.
            *hnext=SAFETY*h*pow(errmax,PSHRNK);
            h=(h >= 0.0 ? FMAX(*hnext,SHRNK*h) : FMIN(*hnext,SHRNK*h));
        }
    }                                Go back and re-try step.
    nrerror("exceeded MAXTRY in stiff");
}
```

Here are the Kaps-Rentrop parameters, which can be substituted for those of Shampine simply by replacing the `#define` statements:

```
#define GAM 0.231
#define A21 2.0
#define A31 4.52470820736
#define A32 4.16352878860
#define C21 -5.07167533877
#define C31 6.02015272865
#define C32 0.159750684673
#define C41 -1.856343618677
#define C42 -8.50538085819
#define C43 -2.08407513602
#define B1 3.95750374663
#define B2 4.62489238836
#define B3 0.617477263873
#define B4 1.282612945268
#define E1 -2.30215540292
#define E2 -3.07363448539
#define E3 0.873280801802
#define E4 1.282612945268
#define C1X GAM
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 170 of 181

```
#define C2X -0.396296677520e-01
#define C3X 0.550778939579
#define C4X -0.553509845700e-01
#define A2X 0.462
#define A3X 0.880208333333
```

As an example of how `stiff` is used, one can solve the system

$$\begin{aligned} y_1' &= -.013y_1 - 1000y_1y_3 \\ y_2' &= -2500y_2y_3 \\ y_3' &= -.013y_1 - 1000y_1y_3 - 2500y_2y_3 \end{aligned} \tag{16.6.27}$$

with initial conditions

$$y_1(0) = 1, \qquad y_2(0) = 1, \qquad y_3(0) = 0 \tag{16.6.28}$$

(This is test problem D4 in [4].) We integrate the system up to $x = 50$ with an initial stepsize of $h = 2.9 \times 10^{-4}$ using `odeint`. The components of $\mathbf{C}$ in (16.6.20) are all set to unity. The routines `derivs` and `jacobn` for this problem are given below. Even though the ratio of largest to smallest decay constants for this problem is around 10^6, `stiff` succeeds in integrating this set in only 29 steps with $\epsilon = 10^{-4}$. By contrast, the Runge-Kutta routine `rkqs` requires 51,012 steps!

```
void jacobn(float x, float y[], float dfdx[], float **dfdy, int n)
{
    int i;

    for (i=1;i<=n;i++) dfdx[i]=0.0;
    dfdy[1][1] = -0.013-1000.0*y[3];
    dfdy[1][2]=0.0;
    dfdy[1][3] = -1000.0*y[1];
    dfdy[2][1]=0.0;
    dfdy[2][2] = -2500.0*y[3];
    dfdy[2][3] = -2500.0*y[2];
    dfdy[3][1] = -0.013-1000.0*y[3];
    dfdy[3][2] = -2500.0*y[3];
    dfdy[3][3] = -1000.0*y[1]-2500.0*y[2];
}

void derivs(float x, float y[], float dydx[])
{
    dydx[1] = -0.013*y[1]-1000.0*y[1]*y[3];
    dydx[2] = -2500.0*y[2]*y[3];
    dydx[3] = -0.013*y[1]-1000.0*y[1]*y[3]-2500.0*y[2]*y[3];
}
```

Semi-implicit Extrapolation Method

The Bulirsch-Stoer method, which discretizes the differential equation using the modified midpoint rule, does not work for stiff problems. Bader and Deuflhard [5] discovered a semi-implicit discretization that works very well and that lends itself to extrapolation exactly as in the original Bulirsch-Stoer method.

The starting point is an implicit form of the midpoint rule:

$$\mathbf{y}_{n+1} - \mathbf{y}_{n-1} = 2h\mathbf{f}\left(\frac{\mathbf{y}_{n+1} + \mathbf{y}_{n-1}}{2}\right) \tag{16.6.29}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 171 of 181

Convert this equation into semi-implicit form by linearizing the right-hand side about $\mathbf{f}(\mathbf{y}_n)$. The result is the *semi-implicit midpoint rule*:

$$\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \cdot \mathbf{y}_{n+1} = \left[\mathbf{1} + h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \cdot \mathbf{y}_{n-1} + 2h\left[\mathbf{f}(\mathbf{y}_n) - \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \cdot \mathbf{y}_n\right] \tag{16.6.30}$$

It is used with a special first step, the semi-implicit Euler step (16.6.17), and a special "smoothing" last step in which the last $\mathbf{y}_n$ is replaced by

$$\overline{\mathbf{y}}_n \equiv \tfrac{1}{2}(\mathbf{y}_{n+1} + \mathbf{y}_{n-1}) \tag{16.6.31}$$

Bader and Deuflhard showed that the error series for this method once again involves only even powers of h.

For practical implementation, it is better to rewrite the equations using $\Delta_k \equiv \mathbf{y}_{k+1} - \mathbf{y}_k$. With $h = H/m$, start by calculating

$$\begin{aligned} \Delta_0 &= \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot h\mathbf{f}(\mathbf{y}_0) \\ \mathbf{y}_1 &= \mathbf{y}_0 + \Delta_0 \end{aligned} \tag{16.6.32}$$

Then for $k = 1, \ldots, m-1$, set

$$\begin{aligned} \Delta_k &= \Delta_{k-1} + 2\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot [h\mathbf{f}(\mathbf{y}_k) - \Delta_{k-1}] \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + \Delta_k \end{aligned} \tag{16.6.33}$$

Finally compute

$$\begin{aligned} \Delta_m &= \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot [h\mathbf{f}(\mathbf{y}_m) - \Delta_{m-1}] \\ \overline{\mathbf{y}}_m &= \mathbf{y}_m + \Delta_m \end{aligned} \tag{16.6.34}$$

It is easy to incorporate the replacement (16.6.19) in the above formulas. The additional terms in the Jacobian that come from $\partial\mathbf{f}/\partial x$ all cancel out of the semi-implicit midpoint rule (16.6.30). In the special first step (16.6.17), and in the corresponding equation (16.6.32), the term $h\mathbf{f}$ becomes $h\mathbf{f} + h^2\partial\mathbf{f}/\partial x$. The remaining equations are all unchanged.

This algorithm is implemented in the routine `simpr`:

```
#include "nrutil.h"

void simpr(float y[], float dydx[], float dfdx[], float **dfdy, int n,
    float xs, float htot, int nstep, float yout[],
    void (*derivs)(float, float [], float []))
```

Performs one step of semi-implicit midpoint rule. Input are the dependent variable `y[1..n]`, its derivative `dydx[1..n]`, the derivative of the right-hand side with respect to x, `dfdx[1..n]`, and the Jacobian `dfdy[1..n][1..n]` at `xs`. Also input are `htot`, the total step to be taken, and `nstep`, the number of substeps to be used. The output is returned as `yout[1..n]`. `derivs` is the user-supplied routine that calculates `dydx`.

```
{
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,j,nn,*indx;
    float d,h,x,**a,*del,*ytemp;

    indx=ivector(1,n);
    a=matrix(1,n,1,n);
    del=vector(1,n);
    ytemp=vector(1,n);
    h=htot/nstep;                       Stepsize this trip.
    for (i=1;i<=n;i++) {                Set up the matrix 1 - hf'.
        for (j=1;j<=n;j++) a[i][j] = -h*dfdy[i][j];
```

```
        ++a[i][i];
    }
    ludcmp(a,n,indx,&d);                LU decomposition of the matrix.
    for (i=1;i<=n;i++)                  Set up right-hand side for first step. Use yout
        yout[i]=h*(dydx[i]+h*dfdx[i]);      for temporary storage.
    lubksb(a,n,indx,yout);
    for (i=1;i<=n;i++)                  First step.
        ytemp[i]=y[i]+(del[i]=yout[i]);
    x=xs+h;
    (*derivs)(x,ytemp,yout);            Use yout for temporary storage of derivatives.
    for (nn=2;nn<=nstep;nn++) {         General step.
        for (i=1;i<=n;i++)              Set up right-hand side for general step.
            yout[i]=h*yout[i]-del[i];
        lubksb(a,n,indx,yout);
        for (i=1;i<=n;i++)
            ytemp[i] += (del[i] += 2.0*yout[i]);
        x += h;
        (*derivs)(x,ytemp,yout);
    }
    for (i=1;i<=n;i++)                  Set up right-hand side for last step.
        yout[i]=h*yout[i]-del[i];
    lubksb(a,n,indx,yout);
    for (i=1;i<=n;i++)                  Take last step.
        yout[i] += ytemp[i];
    free_vector(ytemp,1,n);
    free_vector(del,1,n);
    free_matrix(a,1,n,1,n);
    free_ivector(indx,1,n);
}
```

The routine `simpr` is intended to be used in a routine `stifbs` that is almost exactly the same as `bsstep`. The only differences are:

- The stepsize sequence is

$$n = 2, 6, 10, 14, 22, 34, 50, \ldots, \tag{16.6.35}$$

  where each member differs from its predecessor by the smallest multiple of 4 that makes the ratio of successive terms be $\leq \frac{5}{7}$. The parameter `KMAXX` is taken to be 7.
- The work per unit step now includes the cost of Jacobian evaluations as well as function evaluations. We count one Jacobian evaluation as equivalent to N function evaluations, where N is the number of equations.
- Once again the user-supplied routine `derivs` is a dummy argument and so can have any name. However, to maintain "plug-compatibility" with `rkqs`, `bsstep` and `stiff`, the routine `jacobn` is not an argument and *must* have exactly this name. It is called once per step to return $\mathbf{f}'$ (`dfdy`) and $\partial\mathbf{f}/\partial x$ (`dfdx`) as functions of x and $\mathbf{y}$.

Here is the routine, with comments pointing out only the differences from `bsstep`:

```
#include <math.h>
#include "nrutil.h"
#define KMAXX 7
#define IMAXX (KMAXX+1)
#define SAFE1 0.25
#define SAFE2 0.7
#define REDMAX 1.0e-5
#define REDMIN 0.7
#define TINY 1.0e-30
#define SCALMX 0.1

float **d,*x;

void stifbs(float y[], float dydx[], int nv, float *xx, float htry, float eps,
    float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []))
```

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 173 of 181

Semi-implicit extrapolation step for integrating stiff o.d.e.'s, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector `y[1..nv]` and its derivative `dydx[1..nv]` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal[1..nv]` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is a user-supplied routine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied routine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`. Be sure to set `htry` on successive steps to the value of `hnext` returned from the previous step, as is the case if the routine is called by `odeint`.

```
{
    void jacobn(float x, float y[], float dfdx[], float **dfdy, int n);
    void simpr(float y[], float dydx[], float dfdx[], float **dfdy,
        int n, float xs, float htot, int nstep, float yout[],
        void (*derivs)(float, float [], float []));
    void pzextr(int iest, float xest, float yest[], float yz[], float dy[],
        int nv);
    int i,iq,k,kk,km;
    static int first=1,kmax,kopt,nvold = -1;
    static float epsold = -1.0,xnew;
    float eps1,errmax,fact,h,red,scale,work,wrkmin,xest;
    float *dfdx,**dfdy,*err,*yerr,*ysav,*yseq;
    static float a[IMAXX+1];
    static float alf[KMAXX+1][KMAXX+1];
    static int nseq[IMAXX+1]={0,2,6,10,14,22,34,50,70};      Sequence is different from
    int reduct,exitflag=0;                                        bsstep.

    d=matrix(1,nv,1,KMAXX);
    dfdx=vector(1,nv);
    dfdy=matrix(1,nv,1,nv);
    err=vector(1,KMAXX);
    x=vector(1,KMAXX);
    yerr=vector(1,nv);
    ysav=vector(1,nv);
    yseq=vector(1,nv);
    if(eps != epsold || nv != nvold) {         Reinitialize also if nv has changed.
        *hnext = xnew = -1.0e29;
        eps1=SAFE1*eps;
        a[1]=nseq[1]+1;
        for (k=1;k<=KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
        for (iq=2;iq<=KMAXX;iq++) {
            for (k=1;k<iq;k++)
                alf[k][iq]=pow(eps1,((a[k+1]-a[iq+1])/
                    ((a[iq+1]-a[1]+1.0)*(2*k+1))));
        }
        epsold=eps;
        nvold=nv;                              Save nv.
        a[1] += nv;                            Add cost of Jacobian evaluations to work
        for (k=1;k<=KMAXX;k++) a[k+1]=a[k]+nseq[k+1];        coefficients.
        for (kopt=2;kopt<KMAXX;kopt++)
            if (a[kopt+1] > a[kopt]*alf[kopt-1][kopt]) break;
        kmax=kopt;
    }
    h=htry;
    for (i=1;i<=nv;i++) ysav[i]=y[i];
    jacobn(*xx,y,dfdx,dfdy,nv);                Evaluate Jacobian.
    if (*xx != xnew || h != (*hnext)) {
        first=1;
        kopt=kmax;
    }
    reduct=0;
    for (;;) {
        for (k=1;k<=kmax;k++) {
```

```
        xnew=(*xx)+h;
        if (xnew == (*xx)) nrerror("step size underflow in stifbs");
        simpr(ysav,dydx,dfdx,dfdy,nv,*xx,h,nseq[k],yseq,derivs);
        Semi-implicit midpoint rule.
        xest=SQR(h/nseq[k]);              The rest of the routine is identical to
        pzextr(k,xest,yseq,y,yerr,nv);       bsstep.
        if (k != 1) {
            errmax=TINY;
            for (i=1;i<=nv;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
            errmax /= eps;
            km=k-1;
            err[km]=pow(errmax/SAFE1,1.0/(2*km+1));
        }
        if (k != 1 && (k >= kopt-1 || first)) {
            if (errmax < 1.0) {
                exitflag=1;
                break;
            }
            if (k == kmax || k == kopt+1) {
                red=SAFE2/err[km];
                break;
            }
            else if (k == kopt && alf[kopt-1][kopt] < err[km]) {
                    red=1.0/err[km];
                    break;
                }
            else if (kopt == kmax && alf[km][kmax-1] < err[km]) {
                    red=alf[km][kmax-1]*SAFE2/err[km];
                    break;
                }
            else if (alf[km][kopt] < err[km]) {
                red=alf[km][kopt-1]/err[km];
                break;
            }
        }
    }
    if (exitflag) break;
    red=FMIN(red,REDMIN);
    red=FMAX(red,REDMAX);
    h *= red;
    reduct=1;
}
*xx=xnew;
*hdid=h;
first=0;
wrkmin=1.0e35;
for (kk=1;kk<=km;kk++) {
    fact=FMAX(err[kk],SCALMX);
    work=fact*a[kk+1];
    if (work < wrkmin) {
        scale=fact;
        wrkmin=work;
        kopt=kk+1;
    }
}
*hnext=h/scale;
if (kopt >= k && kopt != kmax && !reduct) {
    fact=FMAX(scale/alf[kopt-1][kopt],SCALMX);
    if (a[kopt+1]*fact <= wrkmin) {
        *hnext=h/fact;
        kopt++;
    }
}
free_vector(yseq,1,nv);
```

```
    free_vector(ysav,1,nv);
    free_vector(yerr,1,nv);
    free_vector(x,1,KMAXX);
    free_vector(err,1,KMAXX);
    free_matrix(dfdy,1,nv,1,nv);
    free_vector(dfdx,1,nv);
    free_matrix(d,1,nv,1,KMAXX);
}
```

The routine `stifbs` is an excellent routine for all stiff problems, competitive with the best Gear-type routines. `stiff` is comparable in execution time for moderate N and $\epsilon \lesssim 10^{-4}$. By the time $\epsilon \sim 10^{-8}$, `stifbs` is roughly an order of magnitude faster. There are further improvements that could be applied to `stifbs` to make it even more robust. For example, very occasionally `ludcmp` in `simpr` will encounter a singular matrix. You could arrange for the stepsize to be reduced, say by a factor of the current `nseq[k]`. There are also certain stability restrictions on the stepsize that come into play on some problems. For a discussion of how to implement these automatically, see [6].

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Kaps, P., and Rentrop, P. 1979, *Numerische Mathematik*, vol. 33, pp. 55–68. [2]

Shampine, L.F. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 93–113. [3]

Enright, W.H., and Pryce, J.D. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 1–27. [4]

Bader, G., and Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 373–398. [5]

Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 399–422.

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535.

Deuflhard, P. 1987, "Uniqueness Theorems for Stiff ODE Initial Value Problems," *Preprint SC-87-3* (Berlin: Konrad Zuse Zentrum für Informationstechnik). [6]

Enright, W.H., Hull, T.E., and Lindberg, B. 1975, *BIT*, vol. 15, pp. 10–48.

Wanner, G. 1988, in *Numerical Analysis 1987*, Pitman Research Notes in Mathematics, vol. 170, D.F. Griffiths and G.A. Watson, eds. (Harlow, Essex, U.K.: Longman Scientific and Technical).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag).

16.7 Multistep, Multivalue, and Predictor-Corrector Methods

The terms multistep and multivalue describe two different ways of implementing essentially the same integration technique for ODEs. Predictor-corrector is a particular subcategrory of these methods — in fact, the most widely used. Accordingly, the name predictor-corrector is often loosely used to denote all these methods.

We suspect that predictor-corrector integrators have had their day, and that they are no longer the method of choice for most problems in ODEs. For high-precision applications, or applications where evaluations of the right-hand sides are expensive, Bulirsch-Stoer dominates. For convenience, or for low precision, adaptive-stepsize Runge-Kutta dominates. Predictor-corrector methods have been, we think, squeezed

out in the middle. There is possibly only one exceptional case: high-precision solution of very smooth equations with very complicated right-hand sides, as we will describe later.

Nevertheless, these methods have had a long historical run. Textbooks are full of information on them, and there are a lot of standard ODE programs around that are based on predictor-corrector methods. Many capable researchers have a lot of experience with predictor-corrector routines, and they see no reason to make a precipitous change of habit. It is not a bad idea for you to be familiar with the principles involved, and even with the sorts of bookkeeping details that are the bane of these methods. Otherwise there will be a big surprise in store when you first have to fix a problem in a predictor-corrector routine.

Let us first consider the multistep approach. Think about how integrating an ODE is different from finding the integral of a function: For a function, the integrand has a known dependence on the independent variable x, and can be evaluated at will. For an ODE, the "integrand" is the right-hand side, which depends both on x and on the dependent variables y. Thus to advance the solution of $y' = f(x, y)$ from x_n to x, we have

$$y(x) = y_n + \int_{x_n}^{x} f(x', y)\, dx' \tag{16.7.1}$$

In a single-step method like Runge-Kutta or Bulirsch-Stoer, the value y_{n+1} at x_{n+1} depends only on y_n. In a multistep method, we approximate $f(x, y)$ by a polynomial passing through *several* previous points $x_n, x_{n-1}, \ldots$ and possibly also through x_{n+1}. The result of evaluating the integral (16.7.1) at $x = x_{n+1}$ is then of the form

$$y_{n+1} = y_n + h(\beta_0 y'_{n+1} + \beta_1 y'_n + \beta_2 y'_{n-1} + \beta_3 y'_{n-2} + \cdots) \tag{16.7.2}$$

where y'_n denotes $f(x_n, y_n)$, and so on. If $\beta_0 = 0$, the method is explicit; otherwise it is implicit. The order of the method depends on how many previous steps we use to get each new value of y.

Consider how we might solve an implicit formula of the form (16.7.2) for y_{n+1}. Two methods suggest themselves: *functional iteration* and *Newton's method*. In functional iteration, we take some initial guess for y_{n+1}, insert it into the right-hand side of (16.7.2) to get an updated value of y_{n+1}, insert this updated value back into the right-hand side, and continue iterating. But how are we to get an initial guess for y_{n+1}? Easy! Just use some *explicit* formula of the same form as (16.7.2). This is called the *predictor step*. In the predictor step we are essentially *extrapolating* the polynomial fit to the derivative from the previous points to the new point x_{n+1} and then doing the integral (16.7.1) in a Simpson-like manner from x_n to x_{n+1}. The subsequent Simpson-like integration, using the prediction step's value of y_{n+1} to *interpolate* the derivative, is called the *corrector step*. The difference between the predicted and corrected function values supplies information on the local truncation error that can be used to control accuracy and to adjust stepsize.

If one corrector step is good, aren't many better? Why not use each corrector as an improved predictor and iterate to convergence on each step? Answer: Even if you had a *perfect* predictor, the step would still be accurate only to the finite order of the corrector. This incurable error term is on the same order as that which your iteration is supposed to cure, so you are at best changing only the coefficient in front

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 177 of 181

of the error term by a fractional amount. So dubious an improvement is certainly not worth the effort. Your extra effort would be better spent in taking a smaller stepsize.

As described so far, you might think it desirable or necessary to predict several intervals ahead at each step, then to use all these intervals, with various weights, in a Simpson-like corrector step. That is not a good idea. Extrapolation is the least stable part of the procedure, and it is desirable to minimize its effect. Therefore, the integration steps of a predictor-corrector method are overlapping, each one involving several stepsize intervals h, but extending just one such interval farther than the previous ones. Only that one extended interval is extrapolated by each predictor step.

The most popular predictor-corrector methods are probably the Adams-Bashforth-Moulton schemes, which have good stability properties. The Adams-Bashforth part is the predictor. For example, the third-order case is

$$\text{predictor:} \qquad y_{n+1} = y_n + \frac{h}{12}(23y'_n - 16y'_{n-1} + 5y'_{n-2}) + O(h^4) \tag{16.7.3}$$

Here information at the current point x_n, together with the two previous points x_{n-1} and x_{n-2} (assumed equally spaced), is used to predict the value y_{n+1} at the next point, x_{n+1}. The Adams-Moulton part is the corrector. The third-order case is

$$\text{corrector:} \qquad y_{n+1} = y_n + \frac{h}{12}(5y'_{n+1} + 8y'_n - y'_{n-1}) + O(h^4) \tag{16.7.4}$$

Without the trial value of y_{n+1} from the predictor step to insert on the right-hand side, the corrector would be a nasty implicit equation for y_{n+1}.

There are actually three separate processes occurring in a predictor-corrector method: the predictor step, which we call P, the evaluation of the derivative y'_{n+1} from the latest value of y, which we call E, and the corrector step, which we call C. In this notation, iterating m times with the corrector (a practice we inveighed against earlier) would be written P(EC)^m. One also has the choice of finishing with a C or an E step. The lore is that a final E is superior, so the strategy usually recommended is PECE.

Notice that a PC method with a fixed number of iterations (say, one) is an explicit method! When we fix the number of iterations in advance, then the final value of y_{n+1} can be written as some complicated function of known quantities. Thus fixed iteration PC methods lose the strong stability properties of implicit methods and *should only be used for nonstiff problems*.

For stiff problems we *must* use an implicit method if we want to avoid having tiny stepsizes. (Not all implicit methods are good for stiff problems, but fortunately some good ones such as the Gear formulas are known.) We then appear to have two choices for solving the implicit equations: functional iteration to convergence, or Newton iteration. However, it turns out that for stiff problems functional iteration will not even converge unless we use tiny stepsizes, no matter how close our prediction is! Thus Newton iteration is usually an essential part of a multistep stiff solver. For convergence, Newton's method doesn't particularly care what the stepsize is, as long as the prediction is accurate enough.

Multistep methods, as we have described them so far, suffer from two serious difficulties when one tries to implement them:

- Since the formulas require results from equally spaced steps, adjusting the stepsize is difficult.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 178 of 181

- Starting and stopping present problems. For starting, we need the initial values plus several previous steps to prime the pump. Stopping is a problem because equal steps are unlikely to land directly on the desired termination point.

Older implementations of PC methods have various cumbersome ways of dealing with these problems. For example, they might use Runge-Kutta to start and stop. Changing the stepsize requires considerable bookkeeping to do some kind of interpolation procedure. Fortunately both these drawbacks disappear with the multivalue approach.

For multivalue methods the basic data available to the integrator are the first few terms of the Taylor series expansion of the solution at the current point x_n. The aim is to advance the solution and obtain the expansion coefficients at the next point x_{n+1}. This is in contrast to multistep methods, where the data are the values of the solution at $x_n, x_{n-1}, \ldots$. We'll illustrate the idea by considering a four-value method, for which the basic data are

$$\mathbf{y}_n \equiv \begin{pmatrix} y_n \\ hy'_n \\ (h^2/2)y''_n \\ (h^3/6)y'''_n \end{pmatrix} \tag{16.7.5}$$

It is also conventional to scale the derivatives with the powers of $h = x_{n+1} - x_n$ as shown. Note that here we use the vector notation $\mathbf{y}$ to denote the solution and its first few derivatives at a point, not the fact that we are solving a system of equations with many components y.

In terms of the data in (16.7.5), we can approximate the value of the solution y at some point x:

$$y(x) = y_n + (x - x_n)y'_n + \frac{(x - x_n)^2}{2}y''_n + \frac{(x - x_n)^3}{6}y'''_n \tag{16.7.6}$$

Set $x = x_{n+1}$ in equation (16.7.6) to get an approximation to y_{n+1}. Differentiate equation (16.7.6) and set $x = x_{n+1}$ to get an approximation to y'_{n+1}, and similarly for y''_{n+1} and y'''_{n+1}. Call the resulting approximation $\widetilde{\mathbf{y}}_{n+1}$, where the tilde is a reminder that all we have done so far is a polynomial extrapolation of the solution and its derivatives; we have not yet used the differential equation. You can easily verify that

$$\widetilde{\mathbf{y}}_{n+1} = \mathbf{B} \cdot \mathbf{y}_n \tag{16.7.7}$$

where the matrix $\mathbf{B}$ is

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{16.7.8}$$

We now write the actual approximation to $\mathbf{y}_{n+1}$ that we will use by adding a correction to $\widetilde{\mathbf{y}}_{n+1}$:

$$\mathbf{y}_{n+1} = \widetilde{\mathbf{y}}_{n+1} + \alpha\mathbf{r} \tag{16.7.9}$$

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 179 of 181

Here **r** will be a fixed vector of numbers, in the same way that **B** is a fixed matrix. We fix α by requiring that the differential equation

$$y'_{n+1} = f(x_{n+1}, y_{n+1}) \tag{16.7.10}$$

be satisfied. The second of the equations in (16.7.9) is

$$hy'_{n+1} = h\widetilde{y}'_{n+1} + \alpha r_2 \tag{16.7.11}$$

and this will be consistent with (16.7.10) provided

$$r_2 = 1, \qquad \alpha = hf(x_{n+1}, y_{n+1}) - h\widetilde{y}'_{n+1} \tag{16.7.12}$$

The values of r_1, r_3, and r_4 are free for the inventor of a given four-value method to choose. Different choices give different orders of method (i.e., through what order in h the final expression 16.7.9 actually approximates the solution), and different stability properties.

An interesting result, not obvious from our presentation, is that multivalue and multistep methods are entirely equivalent. In other words, the value y_{n+1} given by a multivalue method with given **B** and **r** is exactly the same value given by some multistep method with given β's in equation (16.7.2). For example, it turns out that the Adams-Bashforth formula (16.7.3) corresponds to a four-value method with $r_1 = 0$, $r_3 = 3/4$, and $r_4 = 1/6$. The method is explicit because $r_1 = 0$. The Adams-Moulton method (16.7.4) corresponds to the implicit four-value method with $r_1 = 5/12$, $r_3 = 3/4$, and $r_4 = 1/6$. Implicit multivalue methods are solved the same way as implicit multistep methods: either by a predictor-corrector approach using an explicit method for the predictor, or by Newton iteration for stiff systems.

Why go to all the trouble of introducing a whole new method that turns out to be equivalent to a method you already knew? The reason is that multivalue methods allow an easy solution to the two difficulties we mentioned above in actually implementing multistep methods.

Consider first the question of stepsize adjustment. To change stepsize from h to h' at some point x_n, simply multiply the components of $\mathbf{y}_n$ in (16.7.5) by the appropriate powers of h'/h, and you are ready to continue to $x_n + h'$.

Multivalue methods also allow a relatively easy change in the *order* of the method: Simply change **r**. The usual strategy for this is first to determine the new stepsize with the current order from the error estimate. Then check what stepsize would be predicted using an order one greater and one smaller than the current order. Choose the order that allows you to take the biggest next step. Being able to change order also allows an easy solution to the starting problem: Simply start with a first-order method and let the order automatically increase to the appropriate level.

For low accuracy requirements, a Runge-Kutta routine like `rkqs` is almost always the most efficient choice. For high accuracy, `bsstep` is both robust and efficient. For very smooth functions, a variable-order PC method can invoke very high orders. If the right-hand side of the equation is relatively complicated, so that the expense of evaluating it outweighs the bookkeeping expense, then the best PC packages can outperform Bulirsch-Stoer on such problems. As you can imagine, however, such a variable-stepsize, variable-order method is not trivial to program. If

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 180 of 181

you suspect that your problem is suitable for this treatment, we recommend use of a canned PC package. For further details consult Gear [1] or Shampine and Gordon [2].

Our prediction, nevertheless, is that, as extrapolation methods like Bulirsch-Stoer continue to gain sophistication, they will eventually beat out PC methods in all applications. We are willing, however, to be corrected.

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Shampine, L.F., and Gordon, M.K. 1975, *Computer Solution of Ordinary Differential Equations. The Initial Value Problem.* (San Francisco: W.H Freeman). [2]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 5.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 8.

Hamming, R.W. 1962, *Numerical Methods for Engineers and Scientists*; reprinted 1986 (New York: Dover), Chapters 14–15.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 7.

NVIDIA Ex1011
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606
Page 181 of 181