

Rubin H. Landau, Manuel J. Páez,  
and Cristian C. Bordeianu

WILEY-VCH

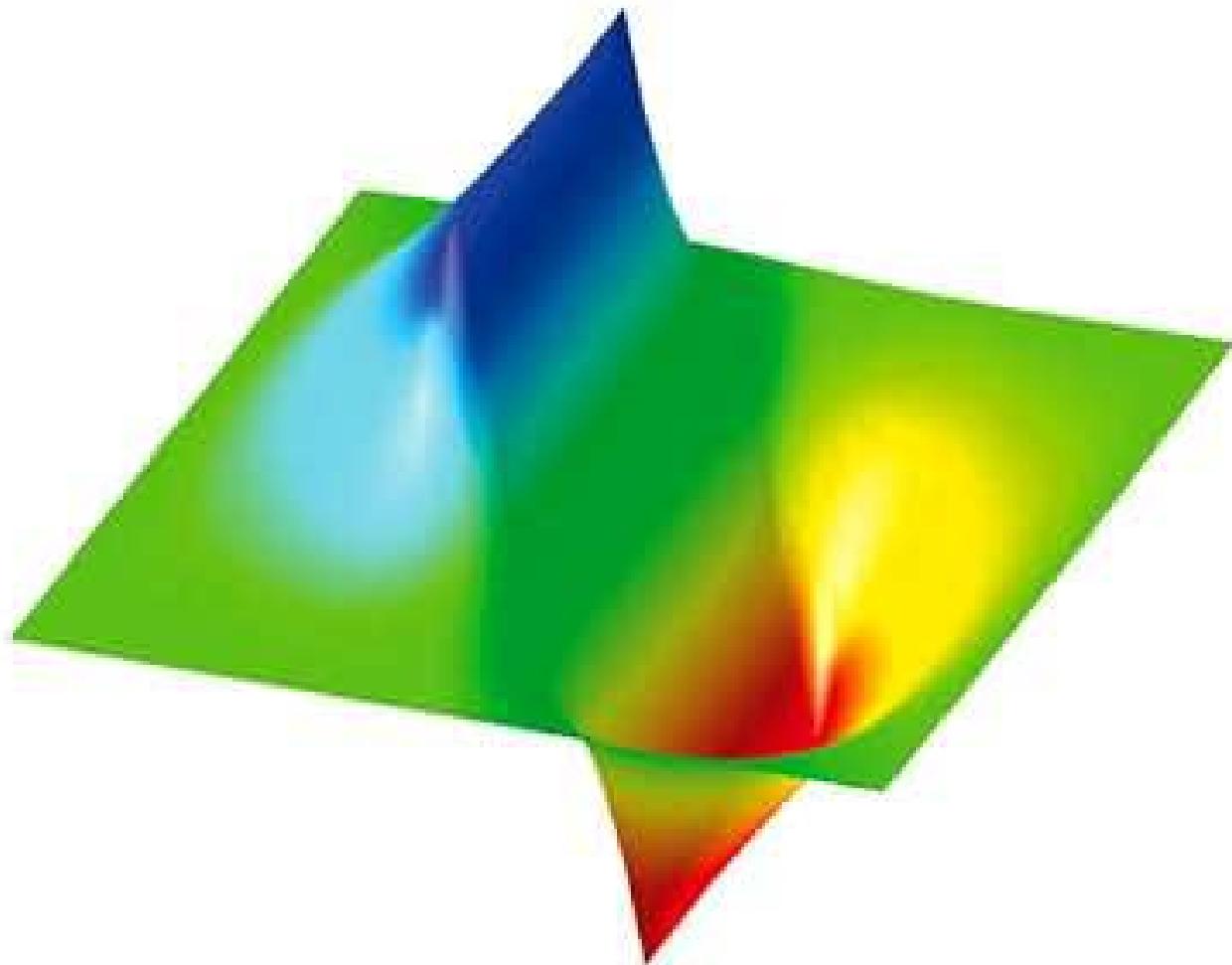
# Computational Physics

Problem Solving with Computers

Second, Revised and Enlarged Edition



included



*Rubin H. Landau, Manuel José Páez,  
and Cristian C. Bordeianu*

**Computational Physics**

## **1807–2007 Knowledge for Generations**

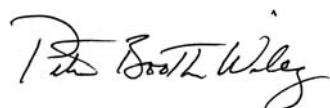
Each generation has its unique needs and aspirations. When Charles Wiley first opened his small printing shop in lower Manhattan in 1807, it was a generation of boundless potential searching for an identity. And we were there, helping to define a new American literary tradition. Over half a century later, in the midst of the Second Industrial Revolution, it was a generation focused on building the future. Once again, we were there, supplying the critical scientific, technical, and engineering knowledge that helped frame the world. Throughout the 20th Century, and into the new millennium, nations began to reach out beyond their own borders and a new international community was born. Wiley was there, expanding its operations around the world to enable a global exchange of ideas, opinions, and know-how.

For 200 years, Wiley has been an integral part of each generation's journey, enabling the flow of information and understanding necessary to meet their needs and fulfill their aspirations. Today, bold new technologies are changing the way we live and learn. Wiley will be there, providing you the must-have knowledge you need to imagine new worlds, new possibilities, and new opportunities.

Generations come and go, but you can always count on Wiley to provide you the knowledge you need, when and where you need it!



William J. Pesce  
President and Chief Executive Officer



Peter Booth Wiley  
Chairman of the Board

*Rubin H. Landau, Manuel J. Páez,  
and Cristian C. Bordeianu*

# **Computational Physics**

## **Problem Solving with Computers**

2nd, Revised and Enlarged Edition



WILEY-VCH Verlag GmbH & Co. KGaA

**The Authors****Prof. Rubin H. Landau**

Oregon State University  
Department of Physics  
Oregon, USA  
[rubin@science.oregonstate.edu](mailto:rubin@science.oregonstate.edu)

**Prof. Manuel J. Páez**

Universidad de Antioquia  
Departamento Física  
Kolumbien  
[mpaez@pegasus.udea.edu.co](mailto:mpaez@pegasus.udea.edu.co)

**Prof. Cristian C. Bordeianu**

University of Bucharest  
Physics Dept.  
Bucharest, Romania  
[cristian.bordeianu@brahms.fizica.unibuc.ro](mailto:cristian.bordeianu@brahms.fizica.unibuc.ro)

All books published by Wiley-VCH are carefully produced. Nevertheless, authors, editors, and publisher do not warrant the information contained in these books, including this book, to be free of errors. Readers are advised to keep in mind that statements, data, illustrations, procedural details or other items may inadvertently be inaccurate.

**Library of Congress Card No.:**  
applied for**British Library Cataloguing-in-Publication Data:**  
A catalogue record for this book is available from the British Library.

**Bibliographic information published by the Deutsche Nationalbibliothek**  
Die Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <http://dnb.d-nb.de>

© 2007 WILEY-VCH Verlag GmbH & Co KGaA,  
Weinheim

All rights reserved (including those of translation into other languages). No part of this book may be reproduced in any form – by photostriking, microfilm, or any other means – nor transmitted or translated into a machine language without written permission from the publishers. Registered names, trademarks, etc. used in this book, even when not specifically marked as such, are not to be considered unprotected by law.

**Composition:** Steingraeber Satztechnik GmbH,  
Ladenburg

**Printing:** Strauss GmbH, Mörlenbach

**Bookbinding:** Litges & Dopf GmbH, Heppenheim

**Wiley Bicentennial Logo:** Richard J. Pacifico

Printed in the Federal Republic of Germany  
Printed on acid-free paper

**ISBN:** 978-3-527-40626-5

*In memory of the parents*  
*Bertha Israel Landau, Philip Landau, Sinclitica Bordeianu*

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Computational Physics and Computational Science	1
1.2	How to Use this Book	3
<b>2</b>	<b>Computing Software Basics</b>	<b>7</b>
2.1	Making Computers Obey	7
2.2	Computer Languages	7
2.3	Programming Warmup	9
2.3.1	Java-Scanner Implementation	10
2.3.2	C Implementation	11
2.3.3	Fortran Implementation	12
2.4	Shells, Editors, and Programs	12
2.5	Limited Range and Precision of Numbers	13
2.6	Number Representation	13
2.7	IEEE Floating Point Numbers	14
2.8	Over/Underflows Exercise	20
2.9	Machine Precision	21
2.10	Determine Your Machine Precision	23
2.11	Structured Program Design	24
2.12	Summing Series	26
2.13	Numeric Summation	26
2.14	Good and Bad Pseudocode	27
2.15	Assessment	27
<b>3</b>	<b>Errors and Uncertainties in Computations</b>	<b>29</b>
3.1	Living with Errors	29
3.2	Types of Errors	29
3.3	Model for Disaster: Subtractive Cancellation	31
3.4	Subtractive Cancellation Exercises	32
3.5	Model for Roundoff Error Accumulation	34

3.6	Errors in Spherical Bessel Functions (Problem)	35
3.7	Numeric Recursion Relations (Method)	35
3.8	Implementation and Assessment: Recursion Relations	37
3.9	Experimental Error Determination	39
3.10	Errors in Algorithms	39
3.11	Minimizing the Error	41
3.12	Error Assessment	42
<b>4</b>	<b>Object-Oriented Programming: Kinematics</b>	45
4.1	Problem: Superposition of Motions	45
4.2	Theory: Object-Oriented Programming	45
4.2.1	OOP Fundamentals	46
4.3	Theory: Newton's Laws, Equation of Motion	46
4.4	OOP Method: Class Structure	47
4.5	Implementation: Uniform 1D Motion, unim1d.cpp	48
4.5.1	Uniform Motion in 1D, Class Um1D	49
4.5.2	Implementation: Uniform Motion in 2D, Child Um2D, unimot2d.cpp	50
4.5.3	Class Um2D: Uniform Motion in 2D	51
4.5.4	Implementation: Projectile Motion, Child Accm2D, accm2d.cpp	53
4.5.5	Accelerated Motion in Two Directions	54
4.6	Assessment: Exploration, shms.cpp	56
<b>5</b>	<b>Integration</b>	59
5.1	Problem: Integrating a Spectrum	59
5.2	Quadrature as Box Counting (Math)	59
5.3	Algorithm: Trapezoid Rule	61
5.4	Algorithm: Simpson's Rule	63
5.5	Integration Error	65
5.6	Algorithm: Gaussian Quadrature	66
5.6.1	Mapping Integration Points	68
5.6.2	Gauss Implementation	69
5.7	Empirical Error Estimate (Assessment)	71
5.8	Experimentation	72
5.9	Higher Order Rules	72
<b>6</b>	<b>Differentiation</b>	75
6.1	Problem 1: Numerical Limits	75
6.2	Method: Numeric	75
6.3	Forward Difference	75
6.4	Central Difference	76
6.5	Extrapolated Difference	77

6.6	Error Analysis	78
6.7	Error Analysis (Implementation and Assessment)	79
6.8	Second Derivatives	80
6.8.1	Second Derivative Assessment	80
<b>7</b>	<b>Trial and Error Searching</b>	<b>81</b>
7.1	Quantum States in Square Well	81
7.2	Trial-and-Error Root Finding via Bisection Algorithm	83
7.2.1	Bisection Algorithm Implementation	84
7.3	Newton–Raphson Algorithm	84
7.3.1	Newton–Raphson with Backtracking	86
7.3.2	Newton–Raphson Implementation	87
<b>8</b>	<b>Matrix Computing and N-D Newton Raphson</b>	<b>89</b>
8.1	Two Masses on a String	90
8.1.1	Statics	91
8.1.2	Multidimensional Newton–Raphson Searching	92
8.2	Classes of Matrix Problems	95
8.2.1	Practical Aspects of Matrix Computing	96
8.2.2	Implementation: Scientific Libraries, WWW	100
8.2.3	Exercises for Testing Matrix Calls	106
8.2.4	Matrix Solution of Problem	108
8.2.5	Explorations	108
<b>9</b>	<b>Data Fitting</b>	<b>111</b>
9.1	Fitting Experimental Spectrum	111
9.1.1	Lagrange Interpolation	112
9.1.2	Lagrange Implementation and Assessment	114
9.1.3	Explore Extrapolation	116
9.1.4	Cubic Splines	116
9.1.5	Spline Fit of Cross Section	118
9.2	Fitting Exponential Decay	120
9.2.1	Theory to Fit	120
9.3	Theory: Probability and Statistics	121
9.4	Least-Squares Fitting	124
9.4.1	Goodness of Fit	126
9.4.2	Least-Squares Fits Implementation	126
9.4.3	Exponential Decay Fit Assessment	128
9.4.4	Exercise: Fitting Heat Flow	129
9.4.5	Nonlinear Fit of Breit–Wigner to Cross Section	130
9.5	Appendix: Calling LAPACK from C	132
9.5.1	Calling LAPACK Fortran from C	134

9.5.2	Compiling C Programs with Fortran Calls	134
<b>10</b>	<b>Deterministic Randomness</b>	137
10.1	Random Sequences	137
10.1.1	Random-Number Generation	138
10.1.2	Implementation: Random Sequence	140
10.1.3	Assessing Randomness and Uniformity	141
<b>11</b>	<b>Monte Carlo Applications</b>	145
11.1	A Random Walk	145
11.1.1	Simulation	145
11.1.2	Implementation: Random Walk	147
11.2	Radioactive Decay	148
11.2.1	Discrete Decay	148
11.2.2	Continuous Decay	150
11.2.3	Simulation	150
11.3	Implementation and Visualization	151
11.4	Integration by Stone Throwing	152
11.5	Integration by Rejection	153
11.5.1	Implementation	154
11.5.2	Integration by Mean Value	154
11.6	High-Dimensional Integration	155
11.6.1	Multidimensional Monte Carlo	156
11.6.2	Error in N-D Integration	156
11.6.3	Implementation: 10D Monte Carlo Integration	157
11.7	Integrating Rapidly Varying Functions	157
11.7.1	Variance Reduction ⊙ (Method)	157
11.7.2	Importance Sampling	158
11.7.3	Implementation: Nonuniform Randomness	158
11.7.4	von Neumann Rejection	162
11.7.5	Nonuniform Assessment	163
<b>12</b>	<b>Thermodynamic Simulations: Ising Model</b>	165
12.1	Statistical Mechanics	165
12.2	An Ising Chain (Model)	166
12.2.1	Analytic Solutions	169
12.3	The Metropolis Algorithm	169
12.3.1	Implementation	173
12.3.2	Equilibration	173
12.3.3	Thermodynamic Properties	175
12.3.4	Beyond Nearest Neighbors and 1D	177

<b>13</b>	<b>Computer Hardware Basics: Memory and CPU</b>	179
13.1	High-Performance Computers	179
13.1.1	Memory Hierarchy	180
13.2	The Central Processing Unit	184
13.2.1	CPU Design: RISC	185
13.2.2	Vector Processor	186
<b>14</b>	<b>High-Performance Computing: Profiling and Tuning</b>	189
14.1	Rules for Optimization	189
14.1.1	Programming for Virtual Memory	190
14.1.2	Optimizing Programs; Java vs. Fortran/C	190
14.1.3	Good, Bad Virtual Memory Use	192
14.1.4	Experimental Effects of Hardware on Performance	193
14.1.5	Java versus Fortran/C	195
14.2	Programming for Data Cache	203
14.2.1	Exercise 1: Cache Misses	204
14.2.2	Exercise 2: Cache Flow	204
14.2.3	Exercise 3: Large Matrix Multiplication	205
<b>15</b>	<b>Differential Equation Applications</b>	207
15.1	UNIT I. <i>Free Nonlinear Oscillations</i>	207
15.2	Nonlinear Oscillator	208
15.3	Math: Types of Differential Equations	209
15.4	Dynamical Form for ODEs	212
15.5	ODE Algorithms	213
15.5.1	Euler's Rule	215
15.5.2	Runge–Kutta Algorithm	215
15.5.3	Assessment: rk2 v. rk4 v. rk45	221
15.6	Solution for Nonlinear Oscillations	223
15.6.1	Precision Assessment: Energy Conservation	224
15.7	Extensions: Nonlinear Resonances, Beats and Friction	225
15.7.1	Friction: Model and Implementation	225
15.7.2	Resonances and Beats: Model and Implementation	226
15.8	Implementation: Inclusion of Time-Dependent Force	226
15.9	UNIT II. <i>Balls, not Planets, Fall Out of the Sky</i>	228
15.10	Theory: Projectile Motion with Drag	228
15.10.1	Simultaneous Second Order ODEs	229
15.10.2	Assessment	230
15.11	Exploration: Planetary Motion	231
15.11.1	Implementation: Planetary Motion	232
<b>16</b>	<b>Quantum Eigenvalues via ODE Matching</b>	235

16.1	Theory: The Quantum Eigenvalue Problem	236
16.1.1	Model: Nucleon in a Box	236
16.1.2	Algorithm: Eigenvalues via ODE Solver + Search	238
16.1.3	Implementation: ODE Eigenvalues Solver	242
16.1.4	Explorations	243
<b>17</b>	<b>Fourier Analysis of Linear and Nonlinear Signals</b>	<b>245</b>
17.1	Harmonics of Nonlinear Oscillations	245
17.2	Fourier Analysis	246
17.2.1	Example 1: Sawtooth Function	248
17.2.2	Example 2: Half-Wave Function	249
17.3	Summation of Fourier Series(Exercise)	250
17.4	Fourier Transforms	250
17.5	Discrete Fourier Transform Algorithm (DFT)	252
17.6	Aliasing and Antialiasing $\odot$	257
17.7	DFT for Fourier Series	259
17.8	Assessments	260
17.9	DFT of Nonperiodic Functions (Exploration)	261
17.10	Model Independent Data Analysis $\odot$	262
17.11	Assessment	264
<b>18</b>	<b>Unusual Dynamics of Nonlinear Systems</b>	<b>267</b>
18.1	The Logistic Map	267
18.2	Properties of Nonlinear Maps	269
18.2.1	Fixed Points	269
18.2.2	Period Doubling, Attractors	270
18.3	Explicit Mapping Implementation	271
18.4	Bifurcation Diagram	272
18.4.1	Implementation	273
18.4.2	Visualization Algorithm: Binning	274
18.5	Random Numbers via Logistic Map	275
18.6	Feigenbaum Constants	276
18.7	Other Maps	276
<b>19</b>	<b>Differential Chaos in Phase Space</b>	<b>277</b>
19.1	Problem: A Pendulum Becomes Chaotic (Differential Chaos)	277
19.2	Equation of Chaotic Pendulum	278
19.2.1	Oscillations of a Free Pendulum	279
19.2.2	Pendulum's "Solution" as Elliptic Integrals	280
19.2.3	Implementation and Test: Free Pendulum	280
19.3	Visualization: Phase-Space Orbits	282
19.3.1	Chaos in Phase Space	285

19.3.2	Assessment in Phase Space	286
19.4	Assessment: Fourier Analysis of Chaos	288
19.5	Exploration: Bifurcations in Chaotic Pendulum	290
19.6	Exploration: Another Type of Phase-Space Plot	291
19.7	Further Explorations	291
<b>20</b>	<b>Fractals</b>	<b>293</b>
20.1	Fractional Dimension	293
20.2	The Sierpiński Gasket	294
20.2.1	Implementation	295
20.2.2	Assessing Fractal Dimension	295
20.3	Beautiful Plants	297
20.3.1	Self-Affine Connection	297
20.3.2	Barnsley's Fern (fern.c)	298
20.3.3	Self-Affinity in Trees (tree.c)	300
20.4	Ballistic Deposition	301
20.4.1	Random Deposition Algorithm (film.c)	301
20.5	Length of British Coastline	303
20.5.1	Coastline as Fractal	303
20.5.2	Box Counting Algorithm	304
20.5.3	Coastline Implementation	305
20.6	Problem 5: Correlated Growth, Forests, and Films	306
20.6.1	Correlated Ballistic Deposition Algorithm (column.c)	307
20.6.2	Globular Cluster	308
20.6.3	Diffusion-Limited Aggregation Algorithm (dla.c)	308
20.6.4	Fractal Analysis of DLA Graph	310
20.7	Problem 7: Fractals in Bifurcation Graph	311
<b>21</b>	<b>Parallel Computing</b>	<b>313</b>
21.1	Parallel Semantics	314
21.1.1	Granularity	315
21.2	Distributed Memory Programming	316
21.3	Parallel Performance	317
21.3.1	Communication Overhead	319
<b>22</b>	<b>Parallel Computing with MPI</b>	<b>321</b>
22.1	Running on a Beowulf	322
22.1.1	An Alternative: BCCD = Your Cluster on a CD	326
22.2	Running MPI	326
22.2.1	MPI under a Queuing System	327
22.2.2	Your First MPI Program	329
22.2.3	MPHello.c Explained	330

22.2.4	Send/Receive Messages	332
22.2.5	Receive More Messages	333
22.2.6	Broadcast Messages: MPIpi.c	334
22.2.7	Exercise	336
22.3	Parallel Tuning: TuneMPI.c	340
22.4	A String Vibrating in Parallel	342
22.4.1	MPIstring.c Exercise	345
22.5	Deadlock	346
22.5.1	Nonblocking Communication	347
22.5.2	Collective Communication	347
22.6	Supplementary Exercises	348
22.7	List of MPI Commands	349
<b>23</b>	<b>Electrostatics Potentials via Finite Differences (PDEs)</b>	<b>351</b>
23.1	PDE Generalities	351
23.2	Electrostatic Potentials	353
23.2.1	Laplace's Elliptic PDE	353
23.3	Fourier Series Solution of PDE	354
23.3.1	Shortcomings of Polynomial Expansions	356
23.4	Solution: Finite Difference Method	357
23.4.1	Relaxation and Over-Relaxation	359
23.4.2	Lattice PDE Implementation	361
23.5	Assessment via Surface Plot	362
23.6	Three Alternate Capacitor Problems	363
23.7	Implementation and Assessment	365
23.8	Other Geometries and Boundary Conditions	368
<b>24</b>	<b>Heat Flow</b>	<b>369</b>
24.1	The Parabolic Heat Equation	369
24.2	Solution: Analytic Expansion	370
24.3	Solution: Finite Time Stepping (Leap Frog)	371
24.4	von Neumann Stability Assessment	373
24.4.1	Implementation	374
24.5	Assessment and Visualization	376
<b>25</b>	<b>PDE Waves on Strings and Membranes</b>	<b>379</b>
25.1	The Hyperbolic Wave Equation	379
25.1.1	Solution via Normal Mode Expansion	381
25.1.2	Algorithm: Time Stepping (Leapfrog)	382
25.1.3	Implementation	386
25.1.4	Assessment and Exploration	386
25.1.5	Including Friction (Extension)	388

25.1.6	Variable Tension and Density	390
25.2	Realistic 1D Wave Exercises	391
25.3	Vibrating Membrane (2D Waves)	392
25.4	Analytical Solution	394
25.5	Numerical Solution for 2D Waves	396
<b>26</b>	<b>Solitons; KdV and Sine-Gordon</b>	<b>399</b>
26.1	Chain of Coupled Pendulums (Theory)	399
26.2	Wave Dispersion	400
26.2.1	Continuum Limit, the SGE	402
26.3	Analytic SGE Solution	403
26.4	Numeric Solution: 2D SGE Solitons	403
26.5	2D Soliton Implementation	406
26.6	Visualization	408
26.7	Shallow Water (KdV) Solitons $\odot$	409
26.8	Theory: The Korteweg–de Vries Equation	410
26.8.1	Analytic Solution: KdV Solitons	411
26.8.2	Algorithm: KdV Soliton Solution	412
26.8.3	Implementation: KdV Solitons	413
26.8.4	Exploration: Two KdV Solitons Crossing	415
26.8.5	Phase-Space Behavior	415
<b>27</b>	<b>Quantum Wave Packets <math>\odot</math></b>	<b>417</b>
27.1	Time-Dependent Schrödinger Equation (Theory)	417
27.1.1	Finite Difference Solution	419
27.1.2	Implementation	419
27.1.3	Visualization and Animation	422
27.2	Wave Packets Confined to Other Wells (Exploration)	422
27.2.1	Algorithm for 2D Schrödinger Equation	423
<b>28</b>	<b>Quantum Paths for Functional Integration</b>	<b>427</b>
28.1	Feynman’s Space–Time Propagation	427
28.1.1	Bound-State Wave Function	431
28.1.2	Lattice Path Integration (Algorithm)	432
28.1.3	Implementation	439
28.1.4	Assessment and Exploration	441
<b>29</b>	<b>Quantum Bound States via Integral Equations</b>	<b>443</b>
29.1	Momentum–Space Schrödinger Equation	444
29.1.1	Integral to Linear Equations	445
29.1.2	Delta-Shell Potential (Model)	447
29.1.3	Implementation	448

29.1.4	Wave Function	449
<b>30</b>	<b>Quantum Scattering via Integral Equations</b>	<b>451</b>
30.1	Lippmann–Schwinger Equation	451
30.1.1	Singular Integrals	452
30.1.2	Numerical Principal Values	453
30.1.3	Reducing Integral to Matrix Equations	454
30.1.4	Solution via Inversion, Elimination	455
30.1.5	Solving <i>ie</i> Integral Equations ⊖	456
30.1.6	Delta-Shell Potential Implementation	456
30.1.7	Scattering Wave Function	458
<b>A</b>	<b>PtPlot: 2D Graphs within Java</b>	<b>461</b>
<b>B</b>	<b>Glossary</b>	<b>467</b>
<b>C</b>	<b>Fortran 95 Codes</b>	<b>479</b>
<b>D</b>	<b>Fortran 77 Codes</b>	<b>513</b>
<b>E</b>	<b>C Language Codes</b>	<b>547</b>
<b>References</b> 583		
<b>Index</b> 587		

## Preface to the Second Edition

During the 10 years that have passed since the publication of the first edition, many students and teachers have read it and rewarded us with their gratitude (and suggestions). Our cumulative experiences have permitted us to uncover which parts of the text could be made clearer and which parts could use new materials. In this second edition we stay true to the general structure of the first edition that has proven successful to many people. However, most every chapter has been rewritten, and new materials have been added in many places, particularly: probability and statistics, trial-and-error search techniques, IEEE floating point representation, optimizing and tuning, the Message Passing Interface (MPI) for parallel programming, the Java Matrix library (JAMA), the solution of simultaneous nonlinear equations, cubic spline techniques, the eigenvalue problem, and the solutions of ordinary and partial differential equations. Probably the most obvious change when compared to the first edition is our use of Java as our language of choice in the text. Even though it might not be evident from all the hype about Java's prowess at Web computing, Java is actually a good language for teaching computational physics: it demands proper syntax, produces useful error messages, is consistent and intelligent in handling precision, goes a long way toward being computer-system independent, has Sun Microsystems providing free program-development environments<sup>1</sup> (so students can compile at home too), and runs fast enough on modern computers for nonresearch purposes. Recognizing that one language cannot satisfy everyone, the CD included with the text contains many of the same programs in C, Fortran 77 and Fortran 9X; we recommend that the reader become familiar with more than one compiled language, and this may be a good way to do that.

## Acknowledgments for the Second Edition

Acknowledgments for the second edition go to Justin Elser, Phil Carter and Donna Hertel (MIP), Juan Vanegas (PDEs and visualization), Zlatko Dimcovic

<sup>1</sup> SUN JAVA DEVELOPER'S SITE, [http://java.sun.com./](http://java.sun.com/)

(improved I/O codes), Guenter Schneider (draft reading), and David Roundy (code formatting). We gratefully acknowledge support of the National Science Foundation through their CCLI program, as well as through their EPIC and NPACI grants, and of Oregon State University's College of Science. Thanks also goes to Anja Tschoertner of Wiley-VCH for her good-natured assistance in the difficult transfer of editorship from New York to Berlin.

Rubin H. Landau

Corvallis, January 2007

## Preface to the First Edition

*Applying computer technology is simply finding  
the right wrench to pound in the correct screw.*

Anonymous

This is not the book I thought I would be writing. When, about a decade ago, I initiated the discussions that led to our Computational Physics course, I thought we would teach mainly physics in it. The Computer Science Department, I thought, would teach the students what they needed to know about computers, the Mathematics Department would teach them what they needed to know about numerical methods and statistics, and I would teach them what I knew about applying that knowledge to solve physics problems using computers. That is how I thought it would be. But, by and large, I have found that the students taking our Computational Physics course do not carry the subject matter from these other disciplines with them, and so a lot of what I have put into this book is material that, in a more perfect world, would be taught and written by experts in other fields.

While that is why I feel this is not the book I thought I would be writing, I believe it is probably for the better. On the one hand, having a basic research physicist tell students they need to know "this" in computer science and "that" in mathematics, gets the message across that "this stuff really matters." On the other hand, it is useful to have the physics, mathematics, and computer science concepts conveyed in the language of a natural scientist and within the context of solving a problem scientifically.

The official and detailed aims of this book are given in Chap. 2, *Computational Science Basics*, and the subjects covered are listed in the Table of Contents. This book differs from a standard text in its underlying philosophy:

*Hear and wonder,  
see and follow,  
do and understand,*

a philosophy also followed in pioneering books such as Thompson, Koonin, and Gould and Tobochnik. As applied in our book, students learn by solving an exceptionally wide class of computational physics problems. When I teach from it, I continually emphasize that it is the student's job to solve each problem, which includes understanding the results the computer gives. In the process, the students are excited by applying scientific, mathematical, and computational techniques that are often new to them (at least in combination).

As a consequence of having to interact with the materials from a number of viewpoints, and often on their own, the materials become part of the personal experiences of the students. To illustrate, I have heard students comment that "I never understood what was dynamic in thermodynamics until after this simulation," and "I would never have imagined that there could be such a difference between upward and downward recursion," as well as "Is that what a random walk *really* looks like?" or "I don't believe a pendulum can really move like that!" In my experience, a teacher just does not hear students express such insight and interest in course material in lecture courses. The students, in turn, are often stimulated to learn more about these subjects or to understand them at greater depth when encountered elsewhere.

There is a price to pay for my unusually broad approach: the students must work hard and cannot master material in great depth. The workload is lightened somewhat by providing "bare bones" programs, and the level is deepened somewhat by having references readily available. In addition, there are appendixes listing the C and Fortran programs and the sections which reference them, as well as source code on the CD and the World Wide Web (the "Web"). The names of the programs are also included in the titles of the Implementation sections. By eliminating time-consuming theoretical background (more properly taught in other places), and by deemphasizing timidity-inducing discussions of error in every single procedure, the students get many challenging projects "to work" for them and enjoy the stimulation of experiencing the material. In the process, the students gain pride and self-confidence immediately, and this makes for a fun course for everyone.

A sample of the problems actually solved during each of two 10-week quarters is given in an appendix. I require the students to write up a mini-lab report for each problem solved containing

Equations solved	Numerical method	Code listing
Visualization	Discussion	Critique

The emphasis is to make the report an executive summary of the type given to a boss or manager; just tell enough to get across that you know what you are talking about, and be certain to convey what you did and your evaluation of it. Recently, some students have written their reports as hypertext documents for the Web. This medium appears ideal for computational projects; the projects

are always in a centralized place for the students and faculty to observe, the original code is there to run or modify, and the visualizations are striking in three-dimensional (3D) color or animation.

An unusual aspect of this book is that, in one form or another, and to an ever-increasing degree, it is enhanced by materials made available on the Web. This is part of a research project which explores ways to better incorporate high-performance computing techniques into science. Even though we cannot recommend the Web as a pleasant way to read a book, we do recommend that students try out some multimedia tutorials and obtain corrections or updates to the programs from the Web. Your comments and corrections regarding the book and its Web enhancements are welcomed.

There is more than enough material to use this book for a full year course (I used it for a two-quarter course and had to pick and choose). It is possible to teach a 10-week class, in which case I would advise moving rapidly through the basic materials so that the students can experience some applications. Chapters 2 and 3, *Computing Software Basics and Errors and Uncertainties in Computation*, in Part I are essential background material. Although placed in the middle of the book, students who are not familiar with computer hardware may benefit from reading Chap. 13, *Computing Hardware Basics: Memory and CPU*, before progressing too far into the materials. Those chapters and sections marked with the  $\odot$  symbol may be treated as optional.

I have found that one or two lectures a week in which the instructor emphasizes the big points and makes it clear what the students "have" to do (and when it is due), appear necessary to keep the course moving at a good pace. Regardless of how the course is taught, most of the learning occurs when the students sit down and experience the physics with their computers, referring to the book or Web enhancements for assistance.

## Acknowledgments (First Edition)

This book was developed with a decade's worth of students in the Computational Physics course at Oregon State University. I am deeply indebted to them for their good-willed cooperation and enthusiasm, and codes. Some materials developed by Henri Jansen have ingrained themselves into this book, and his contributions are gratefully acknowledged. Hans Kowallik has contributed as a student in the course, as a researcher preparing Web tutorials based on these materials, and as a teacher developing materials, figures, and codes; his contributions kept the project on time and improved its quality.

I have also received helpful discussions, valuable materials, and invaluable friendship and encouragement from Paul Fink, Melanie Johnson, Al Stetz, Jon Maestri, Tom Marchioro II, Cherri Pancake, Pat Canan, Shashi Phatak, Paul Hillard, and Al Wasserman. The support of the UCES project and their

award for the Web implementations of our projects, is sincerely acknowledged. In many ways, this book was made possible by the U.S. National Science Foundation (through laboratory development grants and through the NACSE Metacenter Regional Alliance), the U.S. Department of Energy, and Oregon State University. Thanks also goes to the people at Wiley-Interscience, in particular Greg Franklin, John Falcone, Lisa Van Horn, Rosalyn Farkas, and Amy Hendrickson. My final formatting of this book in LaTeX was done at the San Diego Supercomputer Center and the Institute for Nonlinear Science, both at UCSD. Finally, I am especially grateful to my wife Jan, whose reliable support, and encouragement (and proofreading) is lovingly accepted.

*Rubin H. Landau*

Corvallis, 1996

# 1

## Introduction

*Beginnings are hard.*

Chaim Potok

### 1.1

#### Computational Physics and Computational Science

This book adopts the view that computational physics (CP) is a subfield of computational science. This means that CP is a multidisciplinary subject that combines aspects of physics, applied mathematics, and computer science (Fig. 1.1), with the aim of solving realistic physics problems. Other computational sciences replace the physics with biology, chemistry, engineering, etc. At present, grand challenge problems facing computational science include:

Climate Prediction	Materials Science	Structural Biology
Superconductivity	Semiconductor Design	Drug Design
Human Genome	Quantum Chromodynamics	Turbulence
Speech and Vision	Relativistic Astrophysics	Vehicle Dynamics
Nuclear Fusion	Combustion System	Oil and Gas Recovery
Ocean Science	Vehicle Signature (Mil.)	Undersea Surveillance

Although related, computational science is not computer science. Computer science studies computing for its own intrinsic interest and develops the hardware and software tools that computational scientists use. Likewise, applied mathematics develops and studies the mathematics of the algorithms that computational scientists use. Although we too find mathematics and computer science interesting for its own sake, our focus is on solving physical problems and in understanding the computing tools that we use in order to be sure that we have solved the problems correctly.

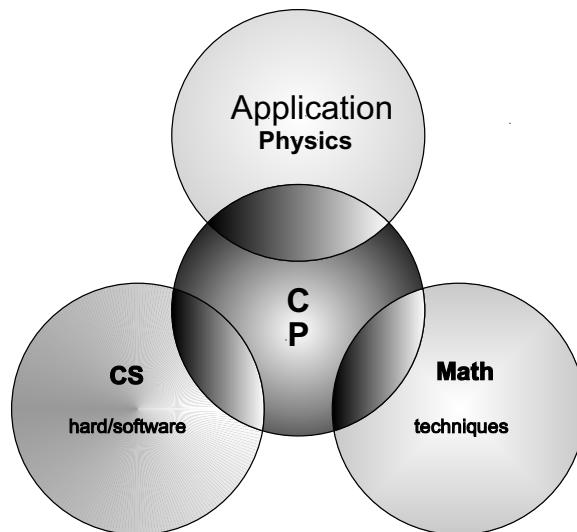
As CP has matured, we have come to realize that it is more than the overlap of physics, computer science, and mathematics (Fig. 1.1). It is also a bridge among them (the central region in the figure) containing core elements of its own, such as computational tools and methods. In our view, CP's common-

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5

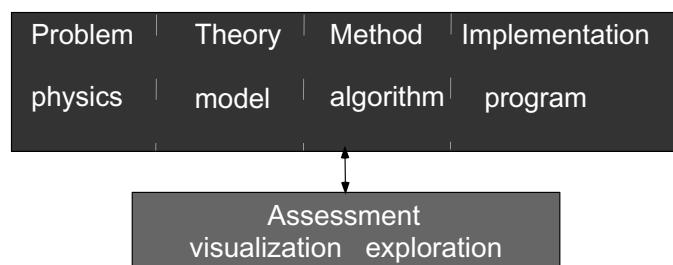


**Fig. 1.1** A representation of the multidisciplinary nature of computational physics not only as the overlap of physics, applied mathematics, and computer science, but also as a bridge among the disciplines.

ality of tools and similar problem-solving mindset [1] draw it closer to the other computational sciences and away from the subspecialization found in so much of physics.

In order to emphasize our computational science focus, we present most of the subjects in this book in the form of a problem to solve, with the components that constitute the solution separated according to the scientific problem-solving paradigm (Fig. 1.2). Clearly, this paradigm is more appropriate to dealing with physical problems than to developing basic skills, yet even in the former case we often follow it for the purpose of uniformity and organization.

Traditionally, physics follows either an experimental or a theoretical approach to discover scientific truth. Being able to transform a theory into an algorithm requires significant theoretical insight, detailed physical and mathematical understanding, and mastery of the art of programming. The actual debugging, testing, and organization of scientific programs have analogies to experiment, with the numerical simulations of nature essentially virtual experiments. The synthesis of numbers into generalizations, predictions, and conclusions requires the insight and intuition common to both experimental and theoretical science. In fact, the use of computation and simulation has now become so prevalent and essential a part of the scientific process that many people have suggested that the scientific paradigm can be extended to include simulation as an additional dimension.



**Fig. 1.2** The problem solving paradigm followed in this book.

We would like the reader of this book to learn how to use computing to help explore the physical world at depths greater than otherwise possible. This may be by solving problems whose difficulty and complexity place them beyond analytic solution or human endurance, or by using the computer as a virtual laboratory that simulates a physical system. Computing is not a substitute for analytic or mathematical skills, but rather a lever for those skills in order for the reader to do something new and different.

## 1.2

### How to Use this Book

The operational aim of this book is to survey the multidisciplinary subjects that constitute what we think of as undergraduate computational physics. For this to be a learning experience, the reader must *do* the highlighted problem at the beginning of each chapter or unit. This means that the reader should study the text, write, debug and run a program, visualize the results, and then express in words what has been done and what can be concluded. In contrast to other books which may emphasize the theory behind CP, we want the reader to understand the algorithm and the program as well as the physics, and to consider the former two as much a part of the scientific process as the latter one.

Although it is possible to use a *problem-solving environment* such as *Maple*, *Mathematica*, or *Matlab* to solve many of these problems without having to write any programs, we recommend that the user “get their hands dirty” by writing programs in a compiled language. We believe it is an important part of the learning experience for you to get close to the hardware and software workings of the computer and to understand the algorithms being used. While indubitably valuable for the well-trained professional, we think that the problem-solving environments hide too much of the details from some-

one trying to learn CP.<sup>1</sup> In addition, while the analytic solutions provided by these packages are often perfect for textbook problems, they are hardly ever useful for realistic problems; or must be evaluated numerically to plot a graph.

While carrying through all the steps in our projects no doubt entails some work, it should be exciting and fun to see the math and physics come alive on the screen before your eyes. Often, as a consequence of having to interact with the materials from a number of viewpoints, and on your own, the materials will become part of your personal experiences and end up being more meaningful than possible via any other way of learning.

Programming is a valuable skill for all scientists, but it is also incredibly demanding and time consuming. In order to lighten the workload somewhat we provide “bare bones” programs in the text and on the CD. These can be used as models or helpers for the reader’s own programs (the best learning experience if time permits), or tested and modified to solve the problem at hand. (More complete versions of the programs are available to instructors by contacting the authors or publisher.) The CD also includes animations and interactive applets based on the text materials, as well as sample data files and visualization scripts.

The programs in the text are given in the Java programming language. The CD contains corresponding versions in C and Fortran; regardless of the language, the algorithm, the math, the logic, and the physics remain the same. Our choice of Java may surprise some readers who view it mainly for Web computing. Actually, Java is quite good for CP education because it demands proper syntax, produces useful error messages, is essentially universal (the same on all computers), and has free program development environments available from Sun so that learners can work on any computer they want. Of particular importance for scientific computing is that Java is consistent and intelligent in its handling of precision, in contrast to C. Although Java is slower than C and Fortran (something we document in Chap. 13), we found this irrelevant for educational projects with the present-day fast computers. Because of the similarity in structure, it is easy and wise to transform a Java program into C for research projects. Finally, although Java is an object-oriented programming (OOP) language (something we discuss at length in Chap. 4), in order to keep the programs closer to the math and readable to a wider audience, we do not emphasize the object orientation in our sample programs, but do encourage you to use OOP techniques.

One of the most rewarding uses of computers is *visualizing* the results of calculations with 2D and 3D plots, with color, and with animation. This assists the debugging process, hastens the development of physical and mathemati-

<sup>1</sup> This is not meant to be a criticism of problem-solving environments; in fact our introductory scientific computing book [2] covers both Maple and Mathematica.

cal intuition, and increases the enjoyment of the work. It is essential that you learn to use visualization tools as soon as possible, and in [2] and [3] we describe a number of free visualization tools that we use and recommend. We include many figures showing visualizations in this edition (unfortunately just in grey scale), with additional animations, applets, and color images on the CD. Furthermore, there is an online version of a course using this book under development, and those interested should contact RHL.

The projects and problems incorporated into the text are an essential part of the learning experience. Some are exercises to ensure that you have your programs functioning properly, while others ask you to think creatively and do self-directed exploration and analysis. In working on these projects, we suggest that you try to develop an attitude of independence and organization, but also that you learn to communicate with others for assistance when you get stuck debugging.

## 2

# Computing Software Basics

### 2.1

#### Making Computers Obey (Problem 1)

You write your own program, wanting to have the computer work something out for you. Your **problem** is that you are beginning to get annoyed because the computer repeatedly refuses to give you the correct answers.

### 2.2

#### Computer Languages (Theory)

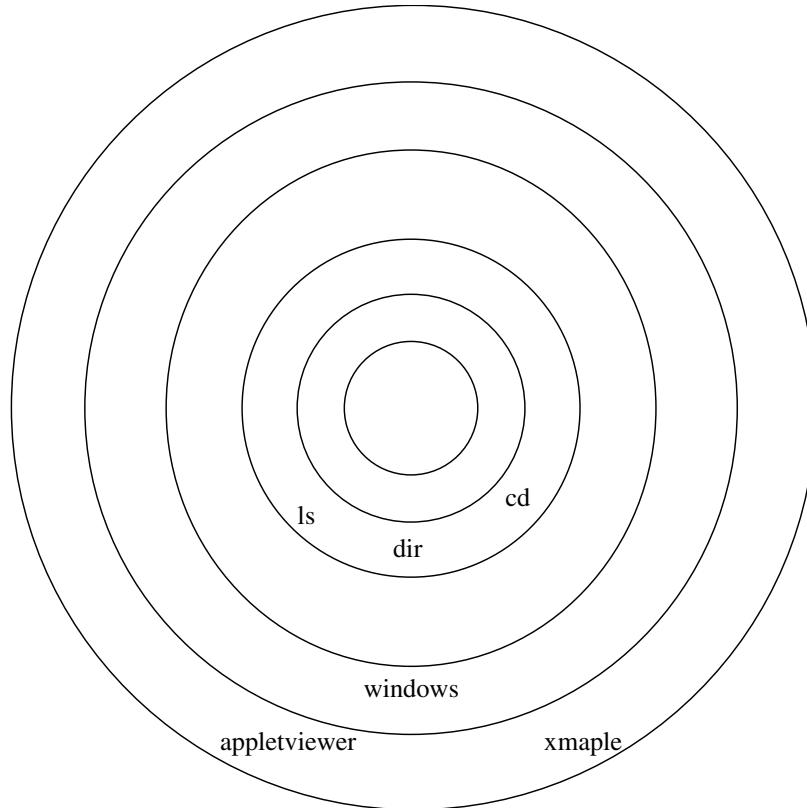
As anthropomorphic as your view of your computer may be, it is good to keep in mind that computers always do exactly as told. This means that you must tell them exactly and everything they have to do. Of course, the programs you run may be so complicated and have so many logical paths that you may not have the endurance to figure out just what you have told the computer to do in detail, but it is always possible in principle. So the real **problem** addressed here is how to give you enough understanding so that you feel well enough in control, no matter how illusionary, to figure out what the computer is doing.<sup>1</sup>

Before you tell the computer to obey your orders, you need to understand that life is not simple for computers. The instructions they understand are in a *basic machine language*<sup>2</sup> that tells the hardware to do things like move a number stored in one memory location to another location, or to do some simple, binary arithmetic. Hardly any computational scientist really talks to a computer in a language it can understand. When writing and running programs,

<sup>1</sup> To keep language from getting in the way of communication, there is a glossary at the end of this book containing many common computer terms. We suggest that you use it as needed.

<sup>2</sup> The “BASIC” (Beginner’s All-purpose Symbolic Instruction Code) programming language should not be confused with basic machine language.

we usually communicate to the computer through *shells*, in *high-level languages* (Java, Fortran, C), or through *problem-solving environments* (Maple, Mathematica, and Matlab). Eventually these commands or programs all get translated to a basic machine language to which the hardware responds.



**Fig. 2.1** A schematic view of a computer's kernel and shells.

A *shell* is a *command-line interpreter*, that is, a set of small programs run by a computer which respond to the commands (the names of the programs) that you key in. Usually you open a special window to access the shell, and this window is called “a shell” as well. It is helpful to think of these shells as the outer layers of the computer’s *operating system* (Fig. 2.1). While every operating system has some type of shell, usually each system has its own set of commands that constitutes its shell. It is the job of the shell to run programs, compilers, and utilities that do things such as copy (`cp`) or delete (`del`, `rm`) files. There can be different types of shells on a single computer, or multiple copies of the same shell running at the same time for different users.

Whereas the shell forms the outermost layer of the operating system, the nucleus of the operating system is called, appropriately, the *kernel*. The user

seldom interacts directly with the kernel, except possibly when installing programs or when building the operating system from scratch.

Operating systems (OS) have names such as *Unix*, *Linux*, *DOS*, *MacOS*, and *MS Windows*. The *operating system* is a group of programs used by the computer to communicate with users and devices, to store and read data, and to execute programs. Under Unix and Linux, the OS tells the computer what to do in an elementary way, while Windows includes various graphical elements as part of the operating system (this increases speed at the cost of complexity). The OS views you, other devices, and programs as input data for it to process; in many ways, it is the indispensable office manager. While all this may seem complicated, the purpose of the OS is to let the computer do the nitty-gritty work so that you may think higher level thoughts and communicate with the computer in something closer to your normal, everyday language.

When you submit a program to your computer in a high-level language, the computer uses a *compiler* to process it. The compiler is another program that treats your program as a foreign language and uses a built-in dictionary and set of rules to translate it into basic machine language. As you can probably imagine, the final set of instructions is quite detailed and long, and the compiler may make several passes through your program to decipher your logic and to translate it into a fast code. The translated statements form an *object code*, and when *linked* together with other needed subprograms, form a *load module*. A load module is a complete set of machine language instructions that can be *loaded* into the computer's memory and read, understood, and followed by the computer.

Languages such as *Fortran* and *C* use programs called *compilers* that read your entire program and then translate it into basic machine instructions. Languages such as *BASIC* and *Maple* translate each line of your program as it is entered. Compiled languages usually lead to more efficient programs and permit the use of vast subprogram libraries. Interpreted languages give a more immediate response to the user and thereby appear "friendlier." The Java language is a mix of the two. When you first compile your program, it interprets it into an intermediate and universal *byte code*, but then when you run your program, it recompiles it into a machine-specific compiled code.

## 2.3

### Programming Warmup

Before we get on to serious work, we want to ensure that you have your local computer working right for you. Assume that calculators have not yet been invented and that you need a program to calculate the area of a circle. Rather than using any specific language, we will discuss how to write that program

in *pseudocode* that can be converted to your favorite language later. The first program tells the computer:<sup>3</sup>

Calculate area of circle	// Do this computer!
--------------------------	----------------------

This program cannot really work because it does not tell the computer which circle to consider and what to do with the area. A better program would be

read radius	// Input
calculate area of circle	// Numerics
print area	// Output

The instruction `calculate area of circle` has no meaning in most computer languages, so we need to specify an *algorithm*, that is, a set of rules for the computer to follow:

read radius	// Input
PI = 3.141593	// Set constant
area = PI * r * r	// Algorithm
print area	// Output

This is a better program, and so let us see how to implement it.

### 2.3.1

#### **Java-Scanner Implementation**

In the past, input and output were not as well developed with Java as with C or Fortran. Yet with Java 1.5 and later, there are the `scanf` and `printf` methods, which, as you can see below, are similar to those in the C program. A Java version of our area program is found on the CD under the name `Area_Scanner.java`, and listed in Listing 2.1. There you will see that we show you what is possible with these new methods by reading from the keyboard, as well as from a file, and by outputing to both screen and file. Beware, unless you first create the file `Name.dat`, the program will take exception with you because it cannot find the file.

<sup>3</sup> Comments placed in the field to the right are for you and *not* for the computer to act upon.

**Listing 2.1:** The code `AreaScanner.java` uses the recent Java 1.5 Scanner class for input. Note how we input first from the keyboard, then from a file, and how different methods are used to convert the input string to doubles or integers.

```
// AreaScanner: examples of use of Scanner and printf (JDK 1.5)

import java.io.*;                                // Standard I/O classes
import java.util.*;                             // scanner class

public class Area_Scanner {
    public static final double PI = 3.141593;           // Constants

    public static void main(String[] argv) throws
        IOException, FileNotFoundException {

        double r, A;                                     // Connect Scanner to std in
        Scanner sc1 = new Scanner(System.in);             System.out.println("Key in your name & r on 1 or more lines");
                                                       // Read String, read double
        String name = sc1.next();                         r = sc1.nextDouble();
        System.out.printf("Hi "+name);
        System.out.printf("\n radius = "+r);
        System.out.printf("\n\n Enter new name and r in Name.dat\n"); // Input file
                                                       // Open file
        Scanner sc2 = new Scanner(new File("Name.dat"));   System.out.printf("Hi %s\n",sc2.next());      // Read, print ln 1
        r = sc2.nextDouble();                            // Read ln 2
        System.out.printf("r = %5.1f\n",r);              // Print ln 2

        A = PI * r * r;
        System.out.printf("Done, look in A.dat\n");       // Print to screen
                                                       // Open output file
        PrintWriter q = new PrintWriter
            (new FileOutputStream("A.dat"), true);
        q.printf("r = %5.1f\n" ,r);                      // File output
        q.printf("A = %8.3f\n" ,A);
        System.out.printf("r = %5.1f\n" , r);             // Screen output
        System.out.printf("A = %8.3f\n" , A);             // Integer input
                                                       // Read int
        System.out.printf("\n\n Now key in your age as an integer\n");
        int age = sc1.nextInt();                         System.out.printf(age+"years old, you don't look it!" );
        sc1.close(); sc2.close();                        // Close inputs
    }
}
```

### 2.3.2

#### C Implementation

A C version of our area program is found on the CD as `area.c`, and listed below:

```
// area.c: Calculate area of a circle (comments for reader)

#include <stdio.h> // Standard I/O headers
#define pi 3.14159265359 // Define constant

int main() { // Declare main
    double r, A; // Double-precision variables
    printf("Enter the radius of a circle \n"); // Request input
    scanf("%lf", &r); // Read from standard input
    A = r * r * pi; // Calculate area
    printf("radius r= %f, area A = %f\n", r, A); // Print results
} // End main
```

### 2.3.3

#### Fortran Implementation

A Fortran version of `area` is found on the CD as `area.f90`, and listed below:

```
! area.f90: Area of a circle , sample program

Program circle_area ! Begin main program
Implicit None ! Declare all variables
Real(8) :: radius, circum, area ! Declare Reals
Real(8) :: PI = 3.14159265358979323846 ! Declare, assign Real
Integer :: model_n = 1 ! Declare, assign Ints
Print *, 'Enter a radius:' ! Talk to user
Read *, radius ! Read into radius
circum = 2.0 * PI * radius ! Calc circumference
area = radius * radius * PI ! Calc area
Print *, 'Program number =', model_n ! Print program number
Print *, 'Radius =', radius ! Print radius
Print *, 'Circumference =', circum ! Print circumference
Print *, 'Area =', area ! Print area
End Program circle_area ! End main prog
```

Notice that the variable and program names are meaningful and similar to standard nomenclature, that there are plenty of comments, and that the input and output are self-explanatory.

### 2.4

#### Shells, Editors, and Programs (Implementation)

1. To gain some experience with your computer system, use an editor to key into a file one of the programs `area.c`, `areas.f90`, or `Area.java`. Then save your file to disk by saving it in your home (personal) directory (we advise having a separate subdirectory for each week).
2. Compile and execute the appropriate version of `Area`.

3. Check that the program is running correctly by running a number of trial cases. Good input data are  $r = 1$ , because then  $A = \pi$  and  $r = 10$ .
4. Experiment with your program. To illustrate, see what happens if you leave off decimal points in the assignment statement for  $r$ , if you assign  $r$  equal to a blank, or if you assign a letter to it. Remember, it is unlikely for you to break anything by exploring.
5. Revise `AreaScanner.java` so that it takes input from a file name that you have made up, then writes in a different format to another file you have created, and then reads from the latter file.
6. See what happens when the data type given to `printf` does not match the type of data in the file (e.g., data are `doubles`, but read in as `ints`).
7. Revise your version of `AreaScanner` so that it uses a main method (which does the input and output) and a separate method for the calculation. Check that the answers do not change when methods are used.

## 2.5

### Limited Range and Precision of Numbers (Problem 2)

Computers may be powerful, but they are finite. A **problem** in computer design is how to represent an arbitrary number using a finite amount of memory space, and then how to deal with the limitations that representation.

## 2.6

### Number Representation (Theory)

As a consequence of computer memories being based on the magnetic or electronic realization of a spin pointing up or down, the most elementary units of computer memory are the two *bits* (binary integers) 0 and 1. This means that all numbers are stored in memory in *binary* form, that is, as long strings of zeros and ones. As a consequence,  $N$  bits can store integers in the range  $[0, 2^N]$ , yet because the sign of the integer is represented by the first bit (a zero bit for positive numbers); the actual range decreases to  $[0, 2^{N-1}]$ .

Long strings of zeros and ones are fine for computers, but are awkward for people. Consequently, binary strings are converted to *octal*, *decimal*, or *hexadecimal* numbers before results are communicated to people. Octal and hexadecimal numbers are nice because the conversion loses no precision, but not all that nice because our decimal rules of arithmetic do not work for them. Converting to decimal numbers makes the numbers easier for us to work with, but unless the number is a power of 2, it leads to a decrease in precision.

A description of a particular computer system will normally state the *word length*, that is, the number of bits used to store a number. The length is often expressed in *bytes* with

$$1 \text{ byte} \stackrel{\text{def}}{=} 8 \text{ bits}$$

Memory and storage sizes are measured in bytes, kilobytes, megabytes, gigabytes, and terabytes. Some care is needed here for those who chose to compute sizes in detail because not everyone means the same thing by these units. As an example,

$$1 \text{ K} \stackrel{\text{def}}{=} 1 \text{ kB} = 2^{10} \text{ bytes} = 1024 \text{ bytes}$$

This is often (and confusingly) compensated for when memory size is stated in Ks, for example,

$$512 \text{ K} = 2^9 \text{ bytes} = 524,288 \text{ bytes} \times \frac{1 \text{ K}}{1024 \text{ bytes}}$$

Conveniently, 1 byte is also the amount of memory needed to store a single character, such as the letter "a" or "b." This adds up to a typical typed page requiring  $\sim 3$  KB of storage.

The memory chips in some of the older personal computers used 8-bit words. This meant that the maximum integer was  $2^7 = 128$  (7 because one bit is used for the sign). Trying to store a number larger than possible (*overflow*) was common on these machines, sometimes accompanied by an informative error message and sometimes not. At present most scientific computers use 64 bits for an integer, which means that the maximum integer is  $2^{63} \simeq 10^{19}$ . While at first this may seem to be a large range for numbers, it really is not when compared to the range of sizes encountered in the physical world. As a case in point, the ratio of the size of the universe to the size of a proton is approximately  $10^{41}$ .

## 2.7

### IEEE Floating Point Numbers

Real numbers are represented on computers in either *fixed-point* or *floating-point* notation. Fixed-point notation uses  $N$  bits to represent a number  $I$  as

$$I_{\text{fix}} = \text{sign} \times (\alpha_n 2^n + \alpha_{n-1} 2^{n-1} + \cdots + \alpha_0 2^0 + \cdots + \alpha_{-m} 2^{-m}) \quad (2.1)$$

where  $n + m = N - 2$ . That is, one bit is used to store the sign, with the remaining  $N - 1$  bits used to store the  $\alpha_i$  values. The particular values for

$N$ ,  $m$ , and  $n$  are machine dependent. Integers are typically 4 bytes (32 bits) in length and in the range

$$-2147,483,648 \leq 4\text{B integer} \leq 2147,483,647.$$

The advantage of the representation (2.1) is that you can count on all fixed-point numbers to have the same absolute error of  $2^{-m-1}$  (the term left off the right-hand end of (2.1)). The corresponding disadvantage is that *small* numbers (those for which the first string of  $\alpha$  values are zeros) have large *relative* errors. Because in the real world relative errors tend to be more important than absolute ones, integers are used mainly for counting purposes and in special applications (like banking).

Most scientific computations use double-precision floating-point numbers. The floating-point representation of numbers on computers is a binary version of what is commonly known as “scientific” or “engineering” notation. For example, the speed of light  $c = +2.99792458 \times 10^{+8}$  m/s in scientific notation. In engineering notation it has the forms  $+0.299792458 \times 10^{+9}$ ,  $0.299795498 \times 10^9$ ,  $0.299795498 \times 10^9$ , or  $0.299795498 \times 10^9$  m/s. In any of the cases, the number out front is called the *mantissa* and contains nine *significant figures*. The power to which 10 is raised is called the *exponent*, with our explicit + signs included as a reminder that the exponent and mantissa may be negative.

Floating-point numbers are stored on the computer as a concatenation (juxtaposition) of the sign bit, the exponent, and the mantissa. Because only a finite number of bits are stored, the set of floating-point numbers that the computer can store exactly, *machine numbers*, is much smaller than the set of real numbers. In particular, there is a maximum and minimum to machine numbers. If you exceed the maximum, an error condition known as *overflow* occurs; if you fall below the minimum, an error condition known as *underflow* occurs.

The actual relation between what is stored in memory and the value of a floating-point number is somewhat indirect, with there being a number of special cases and relations used over the years [4]. In fact, in the past each computer operating system and each computer language would define its own standards for floating-point numbers. Different standards meant that the same program running correctly on different computers could give different results. While the results usually were only slightly different, the user could never be sure if the lack of reproducibility of a test case was due to the particular computer being used, or to an error in the program’s implementation.

In 1987, the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) adopted the IEEE 754 standard for floating-point arithmetic. When the standard is followed, you can expect the primitive data types to have the precision and ranges given in Tab. 2.1. In addition, when computers and software adhere to this standard, and most

**Tab. 2.1** The IEEE 754 standard for Java's primitive data types.

Name	Type	Bits	Bytes	Range
boolean	Logical	1	1/8	true or false
char	String	16	2	'\u0000' $\leftrightarrow$ '\uFFFF' (ISO Unicode characters)
byte	Integer	8	1	-128 $\leftrightarrow$ +127
short	Integer	16	2	-32,768 $\leftrightarrow$ +32,767
int	Integer	32	4	-2,147,483,648 $\leftrightarrow$ +2,147,483,647
long	Integer	64	8	-9,223,372,036,854,775,808 $\leftrightarrow$ 9,223,372,036,854,775,807
float	Floating	32	4	$\pm 1.401298 \times 10^{-45} \leftrightarrow \pm 3.402923 \times 10^{+38}$
double	Floating	64	8	$\pm 4.94065645841246544 \times 10^{-324} \leftrightarrow \pm 1.7976931348623157 \times 10^{+308}$

now do, you are guaranteed that your program will produce identical results on different computers. (However, because the IEEE standard may not produce the most efficient code or the highest accuracy for a particular computer, sometimes you may have to invoke compiler options to demand that the IEEE standard is strictly followed for your test cases; after that you may want to run with whatever gives the greatest speed and precision.)

There are actually a number of components in the IEEE standard, and different computer or chip manufacturers may adhere to only some of them. Normally a floating-point number  $x$  is stored as

$$x_{\text{float}} = (-1)^s \times 1.f \times 2^{e - \text{bias}} \quad (2.2)$$

that is, with separate entities for the sign  $s$ , the fractional part of the mantissa  $f$ , and the exponential field  $e$ . All parts are stored in binary form and occupy adjacent segments of a single 32-bit word for singles, or two adjacent 32-bit words for doubles. The sign  $s$  is stored as a single bit, with  $s = 0$  or  $1$  for positive or negative signs. Eight bits are used to store the exponent  $e$ , which means that  $e$  can be in the range  $0 \leq e \leq 255$ . The endpoints  $e = 0$  and  $e = 255$  are special cases (Tab. 2.2). *Normal numbers* have  $0 < e < 255$ , and with them, the convention is to assume that the mantissa's first bit is a 1, and so only the fractional part  $f$  after the *binary point* is stored. The representations for *subnormal numbers* and the special cases are given in Tab. 2.2.

The IEEE representations ensure that all normal floating-point numbers have the same relative precision. Because the first bit is assumed to be 1, it does not have to be stored, and computer designers need only recall that there is a *phantom bit* there to obtain an extra bit of precision. During the processing of numbers in a calculation, the first bit of an intermediate result may become zero, but this will be corrected before the final number is stored. In summary,

for normal cases, the actual mantissa ( $1.f$  in binary notation) contains an implied 1 preceding the binary point.

**Tab. 2.2** Representation scheme for IEEE Singles.

Number name	Values of s, e, and f	Value of single
Normal	$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$
Subnormal	$e = 0, f \neq 0$	$(-1)^s \times 2^{-126} \times 0.f$
Signed Zero ( $\pm 0$ )	$e = 0, f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, e = 255, f = 0$	$+\text{INF}$
$-\infty$	$s = 1, e = 255, f = 0$	$-\text{INF}$
Not a number	$s = u, e = 255, f \neq 0$	$\text{NaN}$

Finally, in order to guarantee that the stored biased exponent  $e$  is always positive, a fixed number, called the *bias*, is added to the actual exponent  $p$  before it is stored as the biased exponent  $e$ . The actual exponent, which may be negative, is

$$p = e - \text{bias} \quad (2.3)$$

Note that the values  $\pm\text{INF}$  and  $\text{NaN}$  are not numbers in the mathematical sense, that is, objects that can be manipulated or used in calculations to take limits and such. Rather, they are signal to the computer and to you that something has gone wrong and that the calculation should probably stop until you get things right. In contrast, the value  $-0$  can be used in a calculation with no harm. Some languages may set unassigned variable to  $-0$  as a hint that they have yet to be assigned, though it is best not to count on that!

### Example: IEEE Singles Representations

To be more specific about the actual storage of floating-point numbers, we need to examine the two basic floating-point formats: *singles* and *doubles*. “Singles” or *floats* is shorthand for *single precision floating-point numbers*, and “doubles” is shorthand for *double precision floating-point numbers*. Singles occupy 32 bits overall, with 1 for the sign, 8 bits for the exponent, and 23 bits for the fractional mantissa (which gives 24-bit precision when the phantom bit is included). Doubles occupy 64 bits overall, with one for the sign, 10 for the exponent, and 53 for the fractional mantissa (for 54-bit precision). This means that the exponents and mantissas for doubles are not simply double those of floats, as we see in Tab. 2.1. In addition, the IEEE standard also permits *extended precision* that goes beyond doubles, but this is all complicated enough without going into that right now.

To see this scheme in action, look at the 32-bit float represented by (2.2):

	$s$	$e$	$f$	
Bit position:	31	30	23	22 0

The sign bit  $s$  is in bit position 31, the biased exponent  $e$  is in bits 30–23, and the fractional part of the mantissa  $f$  is in bits 22–0. Since eight bits are used to store the exponent  $e$  in (2.2) and  $2^8 = 256$ ,  $e$  has a range

$$0 \leq e \leq 255$$

with  $e = 0$  or 255 as special cases. With the  $bias = 127_{10}$ , the actual exponent

$$p = e_{10} - 127$$

and the full exponent  $p$  for singles has the range

$$-126 \leq p \leq 127$$

as indicated in Tab. 2.1.

The mantissa  $f$  for singles is stored as the 23 bits in positions 22–0. For **normal numbers**, that is, numbers with  $0 < e < 255$ ,  $f$  is the fractional part of the mantissa, and therefore the actual number represented by the 32 bits is

$$\text{normal floating-point number} = (-1)^s \times 1.f \times 2^{e-127}$$

*Subnormal numbers* have  $e = 0$ ,  $f \neq 0$ . For these,  $f$  is the entire mantissa, so the actual number represented by these 32 bits is

$$\text{subnormal numbers} = (-1)^s \times 0.f \times 2^{e-126} \quad (2.4)$$

The 23 bits  $m_{22} - m_0$ , which are used to store the mantissa of normal singles, correspond to the representation

$$\text{mantissa} = 1.f = 1 + m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \dots + m_0 \times 2^{-23} \quad (2.5)$$

with  $0.f$  used for subnormal numbers. The special  $e = 0$  representations used to store  $\pm 0, \pm \infty$  are given in Tab. 2.2.

To see how this works in practice, the largest positive normal floating-point number possible for a 32-bit machine has the maximum value for  $e$  (254), and the maximum value for  $f$ :

$$\begin{aligned} X_{\max} &= 01111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 \\ &= (0)(1111\ 1111)(1111\ 1111\ 1111\ 1111\ 1111\ 1111) \end{aligned} \quad (2.6)$$

where we have grouped the bits for clarity. After putting all the pieces together, we obtain the value shown in Tab. 2.1:

$$\begin{aligned} s &= 0 \quad e = 1111\ 1110 = 254, \quad p = e - 127 = 127, \\ f &= 1.1111\ 1111\ 1111\ 1111\ 1111 = 1 + 0.5 + 0.25 + \dots \simeq 2, \\ \Rightarrow (-1)^s \times 1.f \times 2^{p=e-127} &\simeq 2 \times 2^{127} \simeq 3.4 \times 10^{38}. \end{aligned} \quad (2.7)$$

Likewise, the smallest positive floating-point number possible is subnormal ( $e = 0$ ) with a single significant bit in the mantissa:

$$0\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 001.$$

This corresponds to

$$\begin{aligned} s &= 0 \quad e = 0 \quad p = e - 126 = -126 \\ f &= 0.0000\ 0000\ 0000\ 0000\ 0001 = 2^{-23} \\ \Rightarrow (-1)^s \times 0.f \times 2^{p=e-126} &= 2^{-149} \simeq 1.4 \times 10^{-45} \end{aligned} \quad (2.8)$$

In summary, single-precision (32 bit or 4-byte) numbers have 6–7 decimal places of significance and magnitudes in the range

$$1.4 \times 10^{-45} \leq \text{single precision} \leq 3.4 \times 10^{38}.$$

Specific values are given in Tab. 2.1, and the special cases  $\pm 0$  and  $\pm \infty$  are given in Tab. 2.3:

**Tab. 2.3** The IEEE single precision standard for special cases.

Number	s	e	f
+0	0	0000 0000	0000 0000 00000 00000 00000 000
-0	1	0000 0000	0000 0000 00000 00000 00000 000
$+\infty$	0	1111 1111	0000 0000 00000 00000 00000 000
$-\infty$	1	1111 1111	0000 0000 00000 00000 00000 000

Doubles are stored as two 32-bit words, for a total of 64 bits (8B) overall. The sign occupies one bit, the biased exponent  $e$  11 bits, and the fractional mantissa 52 bits:

	s	e	f	f (cont)
Bit position:	63	62	52	31 0

As we see here, the fields are stored contiguously, with part of the mantissa  $f$  stored in separate 32-bit words. The order of these words, and whether the

second word with  $f$  is the most-, or least-significant part of the mantissa, is machine dependent. For doubles, the bias is quite a bit larger than for singles,

$$bias = 111111111_2 = 1023_{10}$$

so the actual exponent  $p = e - 1023$ .

**Tab. 2.4** Representation scheme for IEEE doubles.

Number name	Values of $s, e$ , and $f$	Value of double
Normal	$0 \leq e \leq 2047$	$(-1)^s \times 2^{e-1023} \times 1.f$
Subnormal	$e = 0, f \neq 0$	$(-1)^s \times 2^{-1022} \times 0.f$
Signed zero	$e = 0, f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, e = 2047, f = 0$	$+INF$
$-\infty$	$s = 1, e = 2047, f = 0$	$-INF$
Not a number	$s = u, e = 2047, f \neq 0$	$NAN$

The bit patterns for doubles are given in Tab. 2.4, with the range and precision given in Tab. 2.1. To summarize, if you write a program with doubles, then 64 bits (8 bytes) will be used to store your floating-point numbers. You will have approximately 16 decimal places of precision (1 part in  $2^{52}$ ) and magnitudes in the range

$$4.9 \times 10^{-324} \leq \text{double precision} \leq 1.8 \times 10^{308} \quad (2.9)$$

If a single-precision number  $x$  is larger than  $2^{128}$ , a fault condition known as *overflow* occurs. If  $x$  is smaller than  $2^{-128}$ , an *underflow* occurs. For overflows, the resulting number  $x_c$  may end up being a machine-dependent pattern, *NAN* (not a number), or unpredictable. For underflows, the resulting number  $x_c$  is usually set to zero, although this can usually be changed via a compiler option. (Having the computer automatically convert underflows to zero is usually a good path to follow; converting overflows to zero may be the path to disaster.) Because the only difference between the representations of positive and negative numbers on the computer is the sign bit of one for negative numbers, the same considerations hold for negative numbers.

In our experience, *serious scientific calculations almost always require 64-bit (double precision) floats*. And if you need double precision in one part of your calculation, you probably need it all over, and that also means double-precision library routines for methods and functions.

## 2.8

### Over/Underflows Exercise

1. Consider the 32-bit, single-precision, floating-point number

	<i>s</i>	<i>e</i>	<i>f</i>
Bit position:	31 0	30 0000 1110	23 1010 0000 0000 0000 0000 0000 000

- (a) What are the (binary) values for the sign *s*, the biased exponent *e*, and the fractional mantissa *f*. (*Hint*:  $e_{10} = 14$ .)
- (b) Determine decimal values for the biased exponent *e* and the true exponent *p*.
- (c) Show that the mantissa of *A* equals 1.625 000.
- (d) Determine the full value of *A*.
2. Write a program to test for the **underflow** and **overflow** limits (within a factor of 2) of your computer system and of your computer language. A sample pseudocode is

```
under = 1. over = 1.
begin do N times
    under = under/2.
    over = over * 2.
    write out: loop number, under, over
end do
```

You may need to increase *N* if your initial choice does not lead to underflow and overflow. Notice that if you want to be more precise regarding the limits of your computer, you may want to multiply and divide by a number smaller than 2.

- (a) Check where under- and overflow occur for single-precision floating-point numbers (floats).
- (b) Check where under- and overflow occur for double-precision floating-point numbers (doubles).
- (c) Check where under- and overflow occur for integers. Note, there is no exponent stored for integers, so the smallest integer corresponds to the most negative one. To determine the largest and smallest integers, you need to observe your program's output as you explicitly pass through the limits. You accomplish this by continually adding and subtracting 1. (Integer arithmetic uses *two's compliment* arithmetic, so you should expect some surprises.)

## 2.9

### Machine Precision (Model)

A major concern of computational scientists is that the floating-point representation used to store numbers is of limited precision. In general for a 32-bit word machine, *single-precision numbers usually are good to 6–7 decimal places*

while doubles are good to 15–16 places. (Java and symbolic manipulation programs let you store numbers with increased precision; that is, they increase the word size to the length needed to obtain the requisite precision.) To see how limited precision affects calculations, consider the simple computer addition of two single-precision words:

$$7 + 1.0 \times 10^{-7} = ?$$

The computer fetches these numbers from memory and stores the bit patterns

$$7 = 0\ 10000010\ 1110\ 0000\ 0000\ 0000\ 0000\ 000, \quad (2.10)$$

$$10^{-7} = 0\ 01100000\ 1101\ 0110\ 1011\ 1111\ 1001\ 010 \quad (2.11)$$

in *working registers* (pieces of fast-responding memory). Because the exponents are different, it would be incorrect to add the mantissas, and so the exponent of the smaller number is made larger while progressively decreasing the mantissa by *shifting bits* to the right (inserting zeros), until both numbers have the same exponent:

$$\begin{aligned} 10^{-7} &= 0\ 01100001\ 0110\ 1011\ 0101\ 1111\ 1100101 (0) \\ &= 0\ 01100010\ 0011\ 0101\ 1010\ 1111\ 1110010 (10) \\ &\dots \\ &= 0\ 10000010\ 0000\ 0000\ 0000\ 0000\ 000 (0001101\dots0) \\ \Rightarrow \quad &7 + 1.0 \times 10^{-7} = 7 \end{aligned} \quad (2.12)$$

Because there is no more room left to store the last digits, they are lost, and after all this hard work, the addition just gives 7 as the answer. In other words, because a 32-bit computer only stores 6–7 decimal places, it effectively ignores any changes beyond the sixth decimal place.

The preceding loss of precision is categorized by defining the *machine precision*  $\epsilon_m$  as the maximum positive number that, on the computer, can be added to the number stored as 1 without changing that stored 1:

$$1_c + \epsilon_m \stackrel{\text{def}}{=} 1_c \quad (2.14)$$

where the subscript  $c$  is a reminder that this is computer representation of 1. Consequently, an arbitrary number  $x$  can be thought of as related to its the floating-point representation  $x_c$  by

$$x_c = x(1 \pm \epsilon) \quad |\epsilon| \leq \epsilon_m$$

where the actual value for  $\epsilon$  is not known. In other words, although some numbers are represented exactly (powers of 2), we should assume that all

single-precision numbers contain an error in their sixth decimal place, and that all doubles have an error in their 15th place. And as is always the case with errors, there is no way to know ahead of time what the error is, for if we could, then we would get rid of the error! Consequently, the arguments we put forth regarding errors are always approximate, and that is the best we can do.

## 2.10

### Determine Your Machine Precision

Write a program to determine the machine precision  $\epsilon_m$  of your computer system (within a factor of 2 or better). A sample pseudocode is

```
eps = 1. begin do N times
    eps = eps/2.
    one = 1. + eps
end do // Make smaller
        // Write loop number, one, eps
```

A Java implementation is given in Listing 2.2.

**Listing 2.2:** The code `Limits.java` available on the CD which determines machine precision within a factor of 2. Note how we skip a line at the beginning of each class or method, and how we align the closing brace vertically with its appropriate keyword (in **bold**).

```
// Limits.java: Determines machine precision

public class Limits {
    public static void main(String[] args) {
        final int N = 60;
        int i;
        double eps = 1., onePlusEps;

        for ( i = 0; i < N; i = i + 1) {
            eps = eps/2.;
            onePlusEps = 1. + eps;
            System.out.println("onePlusEps = " +onePlusEps+, eps = "+eps);
        }
    }
}
```

1. Check the precision for single-precision floating-point numbers.
2. Check the precision for double-precision floating-point numbers.

To print out a number in decimal format, the computer must make a conversion from its internal binary format. Not only does this take time, but unless the stored number is a power of 2, there is a concordant loss of precision. So if

you want a truly precise indication of the stored numbers, you need to avoid conversion to decimals and, instead, print them out in octal or hexadecimal format.

## 2.11

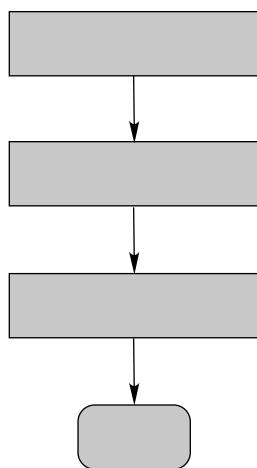
### Structured Program Design

Programming is a written art that blends elements of science, mathematics, and computer science into a set of instructions to permit a computer to accomplish a scientific goal. Now that we are getting into the program-writing business, it is to your benefit to understand not only the detailed grammar of a computer language, but also the overall structures that you should be building into your programs. As with other arts, we suggest that until you know better, you should follow some simple rules. Good programs should

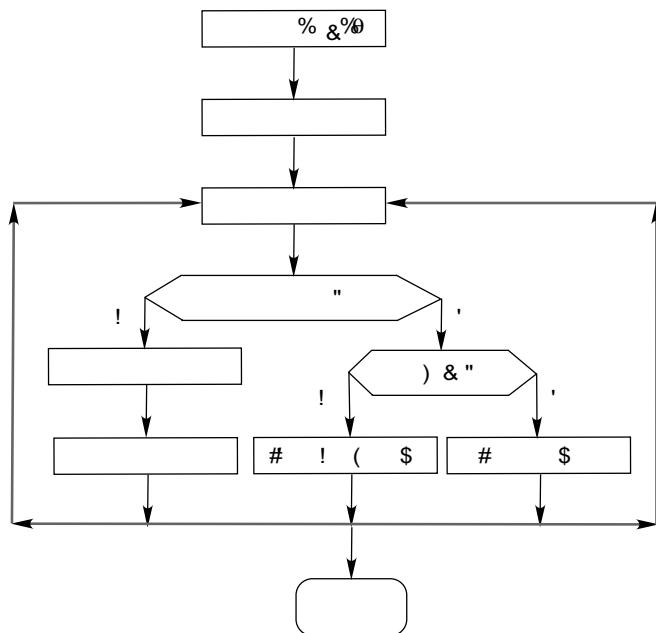
- give the correct answers;
- be clear and easy to read, with the action of each part easy to analyze;
- document themselves for the sake of readers and the programmer;
- be easy to use;
- be easy to modify and robust enough to keep giving the right answers;
- be passed on to others to use and develop further.

The main attraction of object-oriented programming (Chap. 4) is that it enforces these rules, and others, automatically. An elementary way to make your programs clearer is to *structure* them. You may have already noticed that sometimes we show our coding examples with indentation, skipped lines, and braces placed strategically. This is done to provide visual clues to the function of the different program parts (the “structures” in structured programming). Regardless of the compiler ignoring these visual clues, human readers are aided in understanding and modifying the program by having it arranged in a manner that not only looks good, but also makes the different parts of the program manifest to the eye. Even though the page limitation of a printed book keeps us from inserting as many blank lines and spaces as we would prefer, we recommend that you do as we say and not as we do!

In Figs. 2.2 and 2.3 we present *flowcharts* that illustrate a possible program to compute projectile motion. A flowchart is not meant to be a detailed description of a program, but instead is a graphical aide to help visualize its logical flow. As such, it is independent of computer language and is useful for developing and understanding the basic structure of a program. We recommend



**Fig. 2.2** A flowchart illustrating the basic components of a program to compute projectile motion. When writing a program, first map out the basic components, then decide upon the structures, and finally fill in the details (shown in Fig. 2.3).



**Fig. 2.3** A flowchart illustrating some of the details of a program to compute projectile motion. The basic components are shown in Fig. 2.2.

that you draw a flowchart or (second best) write some *pseudocode* every time you write a program. Pseudocode is like a text version of a flowchart in that

it also leaves out details and instead focuses on the logic and structures; to illustrate:

```

Store g, Vo, and theta Calculate R and T
Begin time loop
    Print out ''not yet fired'' if t < 0
    Print out ''grounded'' if t > T
    Calculate, print x(t) and y(t)
    Print out error message if x > R, y > H
End time loop
End program

```

## 2.12

### Summing Series (Problem 3)

A classic numerical problem is the summation of a series to evaluate a function. As an instant, consider the power series for the exponential of a negative argument:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \dots \quad (\text{exact})$$

Your **problem** is to use this series to calculate  $e^{-x}$  for  $x = 0.1, 1, 10, 100$ , and  $1000$ , with an absolute error in each case of less than one part in  $10^8$ . While an infinite series is exact in a mathematical sense, to use it as an algorithm we must stop summing at some point,

$$e^{-x} \approx \sum_{n=0}^N \frac{(-x)^n}{n!} \quad (\text{algorithm}) \quad (2.15)$$

But how do we decide when to stop summing? (Do not even think of saying "When the answer agrees with the table or with the built-in library function.")

## 2.13

### Numeric Summation (Method)

Regardless of the algorithm (2.15) indicating that we should calculate  $(-x)^n$  and then divide it by  $n!$ , this is not a good way to compute. On the one hand,  $n!$  or  $(-x)^n$  can get very large and cause overflows, even though the quotient of the two may not. On the other hand, powers and factorials are very expensive (time consuming) to evaluate on the computer. For these reasons, a better approach is to realize that each term in series (2.15) is just the previous one

times  $(-x)/n$ , and so to make just a single multiplication to obtain the new term:

$$\frac{(-x)^n}{n!} = \frac{(-x)}{n} \frac{(-x)^{n-1}}{(n-1)!} \Rightarrow \text{nth term} = \frac{(-x)}{n} \times (n-1)\text{th term.}$$

While we really want to ensure a definite accuracy for  $e^{-x}$ , that is not so easy to do. What is easy to do is to assume that the error in the summation is approximately the last term summed (this assumes no roundoff error). To obtain an absolute error of one part in  $10^8$ , we stop the calculation when

$$\left| \frac{\text{Nth term}}{\text{sum}} \right| < 10^{-8} \quad (2.16)$$

where *term* is the last term kept in the series (2.15), and *sum* is the accumulated sum of all terms. In general, you are free to pick any tolerance level you desire, although if it is too close to, to smaller than, machine precision, your calculation may not be able to attain it.

## 2.14

### Good and Bad Pseudocode

A pseudocode for performing the summation is

```
term = 1, // Initialize sum = 1, eps = 10^(-8)
do
    term = -term * x/i // New term in terms of old
    sum = sum + term // Add in term
    while abs(term/sum) > eps // Break iteration
end do
```

Write a program that implements this pseudocode for the indicated  $x$  values. Present your results as a table with the heading

x	imax	sum	$ \text{sum} - \exp(-x)  / \exp(-x)$

where  $\exp(-x)$  is the value obtained from the built-in exponential function. The last column here is the relative error in your computation. Modify your code that sums the series in a “good way” (no factorials) to one that calculates the sum in a “bad way” (explicit factorials).

## 2.15

### Assessment

1. Produce a table as above.

2. Start with a tolerance of  $10^{-8}$  as in (2.16).
3. Show that for sufficiently small values of  $x$  your algorithm converges (the changes are smaller than your tolerance level), and that it converges to the correct answer.
4. Do you obtain the number of decimal places of precision expected?
5. Show that there is a range of somewhat large values of  $x$  for which the algorithm converges, but that it converges to the wrong answer.
6. Show that as you keep increasing  $x$  you will reach a regime where the algorithm does not even converge.
7. Repeat the calculation using the “bad” version of the algorithm (the one that calculates factorials), and compare the answers.
8. Set your tolerance level to a number close to machine precision, and see how this affects your conclusions.

### 3

## Errors and Uncertainties in Computations

*Whether you are careful or not, errors and uncertainties are always a part of computation. Some errors are the ones that humans inevitably make, but some are introduced by the computer. Computer errors arise because of the limited precision with which computers store numbers, or because programs can get things wrong. Although it stifles creativity to keep thinking “error” when approaching a computation, it certainly is a waste of time to work with results that are meaningless (“garbage”) because of errors. In this chapter we examine some of the errors and uncertainties introduced by the computer.*

### 3.1

#### Living with Errors (Problem)

Let us say you have a program of significant complexity. To gauge why errors may be a concern for you, imagine a program with the logical flow:

$$\text{start} \rightarrow U_1 \rightarrow U_2 \rightarrow \dots \rightarrow U_n \rightarrow \text{end} \quad (3.1)$$

where each unit  $U$  might be a step. If each unit has probability  $p$  of being correct, then the joint probability  $P$  of the whole program being correct is  $P = p^n$ . Let us say we have a large program with  $n = 1000$  steps, and that the probability of each step being correct is  $p = 0.9993$ . This means that you end up with  $P = \frac{1}{2}$ , that is, a final answer that is as likely wrong as right (not a good way to do science). The **problem** is that, as a scientist, you want a result that is correct—or at least in which the uncertainty is small.

### 3.2

#### Types of Errors (Theory)

Four general types of errors exist to plague your computations:

**Blunders or bad theory:** Typographical errors entered with your program or data, running the wrong program or having a fault in your reasoning

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5

(theory), using the wrong data file, and so on. (If your blunder count starts increasing, it may be time to go home or take a break.)

**Random errors:** Errors caused by events such as fluctuation in electronics due to power surges, cosmic rays, or someone pulling a plug. These may be rare but you have no control over them and their likelihood increases with running time; while you may have confidence in a 20-second calculation, a week-long calculation may have to be run several times to check reproducibility.

**Approximation errors:** Errors arising from simplifying the mathematics so that a problem can be solved or approximated on the computer. They include the replacement of infinite series by finite sums, infinitesimal intervals by finite ones, and variable functions by constants. For example,

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} \simeq \sum_{n=0}^N \frac{(-x)^n}{n!} = e^{-x} + \mathcal{E}(x, N) \quad (3.2)$$

where  $\mathcal{E}(x, N)$  is the approximation error. Because the approximation error arises from the application of the mathematics, it is also called *algorithmic error*. The approximation error clearly decreases as  $N$  increases, and vanishes in the  $N \rightarrow \infty$  limit. Specifically for (3.2), because the scale for  $N$  is set by the value of  $x$ , the small approximation error requires  $N \gg x$ . So if  $x$  and  $N$  are close in value, the approximation error will be large.

**Roundoff errors:** Imprecisions arising from the finite number of digits used to store floating-point numbers. These “errors” are analogous to the uncertainty in the measurement of a physical quantity encountered in an elementary physics laboratory. The overall error arising from using a finite number of digits to represent numbers accumulates as the computer handles more numbers, that is, as the number of steps in a computation increases. In fact, the roundoff error causes some algorithms to become *unstable* with a rapid increase in error for certain parameters. In some cases, the roundoff error may become the major component in your answer, leading to what computer experts call *garbage*. For example, if your computer kept four decimal, then it would store  $1/3$  as 0.3333 and  $2/3$  as 0.6667, where the computer has “rounded off” the last digit in  $2/3$ . Accordingly, if we ask the computer to as simple a calculation as  $2(1/3) - 2/3$ , it produces

$$2\left(\frac{1}{3}\right) - \frac{2}{3} = 0.6666 - 0.6667 = -0.0001 \neq 0 \quad (3.3)$$

So even though the result is small, it is not 0, and if we repeat this type of calculation millions of times, the answer might not even be small.

When considering the precision of calculations, it is good to recall those discussions of *significant figures* and scientific notation given in your early physics

or engineering classes. For computational purposes, let us consider how the computer may store the floating-point number:

$$a = 11223344556677889900 = 1.12233445566778899 \times 10^{19} \quad (3.4)$$

Because the exponent is stored separately and is a small number, we can assume that it will be stored in full precision. The mantissa may not be stored completely, depending on the word length of the computer and whether we declare the word to be stored in single or double precision. In double precision (or `REAL*8` on a 32-bit machine or `double`), the mantissa of  $a$  will be stored in two words, the *most significant part* representing the decimal 1.12233, and the *least significant part* 44556677. The digits beyond 7 may be lost. As we see below, when we perform calculations with words of fixed length, it is inevitable that errors get introduced into the least significant parts of the words.

### 3.3

#### Model for Disaster: Subtractive Cancellation

An operation performed on a computer usually only approximates the analytic answer because the numbers are stored only approximately. To demonstrate the effect of this type of uncertainty, let us call  $x_c$  the computer representation of the exact number  $x$ . The two are related by

$$x_c \simeq x(1 + \epsilon_x) \quad (3.5)$$

where  $\epsilon_x$  is the relative error in  $x_c$ , which we expect to be of a similar magnitude to the machine precision  $\epsilon_m$ . If we apply this notation to the simple subtraction  $a = b - c$ , we obtain

$$\begin{aligned} a = b - c &\Rightarrow a_c \simeq b_c - c_c \simeq b(1 + \epsilon_b) - c(1 + \epsilon_c) \\ &\Rightarrow \frac{a_c}{a} \simeq 1 + \epsilon_b \frac{b}{a} - \frac{c}{a} \epsilon_c \end{aligned} \quad (3.6)$$

We see from (3.6) that the resulting error in  $a$  is essentially a weighted average of the errors in  $b$  and  $c$ , with no assurance that the terms will cancel. Of special importance here is to observe that the error in the answer  $a$  increases when we subtract two nearly equal numbers ( $b \approx c$ ), because then we are subtracting off the most significant parts of both numbers and leaving the error-prone least significant parts:

*If you subtract two large numbers and end up with a small one, there will be less significance in the small one.*

In terms of our error analysis, if the answer from your subtraction  $a$  is small, it must mean  $b \simeq c$  and so

$$\frac{a_c}{a} \stackrel{\text{def}}{=} 1 + \epsilon_a \simeq 1 + \frac{b}{a}(\epsilon_b - \epsilon_c) \simeq 1 + \frac{b}{a}\max(|\epsilon_b|, |\epsilon_c|) \quad (3.7)$$

This shows that even if the relative errors in  $b$  and  $c$  may cancel somewhat, they are multiplied by the large number  $b/a$ , which can significantly magnify the error. Because we cannot assume any sign for the errors, we must assume the worst (the “max” in (3.7)).

We have already seen an example of subtractive cancellation in the power series summation  $e^{-x} \simeq 1 - x + x^2/2 + \dots$  in Chap. 3. For large  $x$ , the early terms in the series are large, but because the final answer must be very small, these large terms must subtract each other away to give the small answer. Consequently, a better approach for calculating  $e^x$  eliminates subtractive cancellation by calculating  $e^x$ , and then setting  $e^{-x} = 1/e^x$ . As you should check for yourself, this improves the algorithm, but does not make it good because there is still a roundoff error.

### 3.4

#### Subtractive Cancellation Exercises

1. Remember back in high school when you learned that the quadratic equation

$$ax^2 + bx + c = 0 \quad (3.8)$$

has an analytic solution that can be written as either

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \text{or} \quad x'_{1,2} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}} \quad (3.9)$$

Inspection of (3.9) indicates that subtractive cancellation (and consequently an increase in error) arises when  $b^2 \gg 4ac$  because then the square root and its preceding term nearly cancel for one of the roots.

- (a) Write a program that calculates all four solutions for arbitrary values of  $a$ ,  $b$ , and  $c$ .
  - (b) Investigate how errors in your computed answers become large as the subtractive cancellation increases, and relate this to the known machine precision. (*Hint:* A good test case employs  $a = 1$ ,  $b = 1$ ,  $c = 10^{-n}$ ,  $n = 1, 2, 3, \dots$ )
  - (c) Extend your program so that it indicates the most precise solutions.
2. As we have seen, subtractive cancellation occurs when summing a series with alternating signs. As another example, consider the finite sum:

$$S_N^{(1)} = \sum_{n=1}^{2N} (-1)^n \frac{n}{n+1} \quad (3.10)$$

If you sum the even and odd values of  $n$  separately, you get two sums

$$S_N^{(2)} = - \sum_{n=1}^N \frac{2n-1}{2n} + \sum_{n=1}^N \frac{2n}{2n+1} \quad (3.11)$$

All terms are positive in this form with just a single subtraction at the end of the calculation. Even this one subtraction and its resulting cancellation can be avoided by combining the series analytically:

$$S_N^{(3)} = \sum_{n=1}^N \frac{1}{2n(2n+1)} \quad (3.12)$$

While all three summations  $S^{(1)}$ ,  $S^{(2)}$ , and  $S^{(3)}$  are mathematically equal, this may not be true numerically.

- (a) Write a single-precision program that calculates  $S^{(1)}$ ,  $S^{(2)}$ , and  $S^{(3)}$ .
  - (b) Assume  $S^{(3)}$  to be the exact answer. Make a log-log plot of the relative error versus number of terms, that is, of  $\log_{10}|(S_N^{(1)} - S_N^{(3)})/S_N^{(3)}|$ , versus  $\log_{10}(N)$ . Start with  $N = 1$  and work up to  $N = 1,000,000$ . (Recall,  $\log_{10} x = \ln x / \ln 10$ .)
  - (c) See whether straight-line behavior for the error occurs in some region of your plot. This indicates that the error is proportional to some power of  $N$ .
3. In spite of the power of your trusty computer, calculating the sum of even a simple series may require some thought and care. Consider the two series

$$S^{(\text{up})} = \sum_{n=1}^N \frac{1}{n}, \quad S^{(\text{down})} = \sum_{n=N}^1 \frac{1}{n}$$

Both series are finite as long as  $N$  is finite, and when summed analytically, it does not matter if you sum the series upward or downward. Nonetheless, because of the roundoff error, the numerical  $S^{(\text{up})} \neq S^{(\text{down})}$ .

- (a) Write a program to calculate  $S^{(\text{up})}$  and  $S^{(\text{down})}$  as functions of  $N$ .
- (b) Make a log-log plot of  $(S^{(\text{up})} - S^{(\text{down})})/(|S^{(\text{up})}| + |S^{(\text{down})}|)$  divided by the sum versus  $N$ .
- (c) Observe the linear regime on your graph and explain why the downward sum is more precise.

### 3.5

#### Model for Roundoff Error Accumulation

Let us start by seeing how error arises in a single multiplication of the computer representation of two numbers:

$$\begin{aligned} a = b \times c &\Rightarrow a_c = b_c \times c_c = b(1 + \epsilon_b) \times c(1 + \epsilon_c) \\ &\Rightarrow \frac{a_c}{a} = (1 + \epsilon_b)(1 + \epsilon_c) \simeq 1 + \epsilon_b + \epsilon_c \end{aligned} \quad (3.13)$$

where we ignore very small  $\epsilon^2$  terms.<sup>1</sup> This is just the basic rule of error propagation from elementary physics or engineering laboratory: you add the uncertainties in each quantity involved in an analysis in order to determine the overall uncertainty. As before, the safest assumption is that there is no cancellation of error, that is, that the errors add in absolute value.

There is a useful model for approximating how the roundoff error accumulates in a calculation involving a large number of steps. We view the error in each step as a literal “step” in a *random walk*, that is, a walk for which each step is in a random direction. As we derive and simulate in Chap. 11 the total distance covered in  $N$  steps of length  $r$ , is, on average,

$$R \approx \sqrt{N} r \quad (3.14)$$

By analogy, the total relative error  $\epsilon_{\text{ro}}$  arising after  $N$  calculational steps each with the machine precision error  $\epsilon_m$ , is, on average,

$$\epsilon_{\text{ro}} \approx \sqrt{N} \epsilon_m \quad (3.15)$$

If the roundoff errors in a particular algorithm do not accumulate in a random manner, then a detailed analysis is needed to predict the dependence of the error on the number of steps  $N$ . In some cases there may be no cancellation and the error may increase like  $N\epsilon_m$ . Even worse, in some recursive algorithms, where the production of errors is coherent, such as the upward recursion for Bessel functions, the error increases like  $N!$ .

Our discussion of errors has an important implication for a student to keep in mind before being impressed by a calculation requiring hours of supercomputer time. A fast computer may complete  $10^{10}$  floating-point operations per second. This means a program running for 3 h performs approximately  $10^{14}$  operations. Therefore, if the roundoff error accumulates randomly, after 3 h we expect a relative error of  $10^7 \epsilon_m$ . For the error to be smaller than the answer, we need  $\epsilon_m < 10^{-7}$ , which means double-precision calculations.

<sup>1</sup> We thank B. Gollsneider for informing us of an error in an earlier version of this equation.

### 3.6

#### Errors in Spherical Bessel Functions (Problem)

Accumulating roundoff errors often limits the ability of a program to perform accurate calculations. Your **problem** is to compute the spherical Bessel and Neumann functions  $j_l(x)$  and  $n_l(x)$ . These are, respectively, the regular (non-singular at the origin) and irregular solutions of the differential equation

$$x^2 f''(x) + 2x f'(x) + \left[ x^2 - l(l+1) \right] f(x) = 0 \quad (3.16)$$

and are related to the Bessel function of the first kind by  $j_l(x) = \sqrt{\pi/2x} J_{n+1/2}(x)$ . Spherical Bessel functions occur in many physical problems, for example; the  $j_l$ 's are part of the partial wave expansion of a plane wave into spherical waves,

$$e^{ik \cdot r} = \sum_{l=0}^{\infty} i^l (2l+1) j_l(kr) P_l(\cos \theta) \quad (3.17)$$

where  $\theta$  is the angle between  $\mathbf{k}$  and  $\mathbf{r}$ . Figure 3.1 shows what the first few  $j_l$ 's look like, and Tab. 3.1 gives some explicit values. For the first two  $l$  values, explicit forms are

$$j_0(x) = +\frac{\sin x}{x} \quad j_1(x) = +\frac{\sin x}{x^2} - \frac{\cos x}{x} \quad (3.18)$$

$$n_0(x) = -\frac{\cos x}{x} \quad n_1(x) = -\frac{\cos x}{x^2} - \frac{\sin x}{x} \quad (3.19)$$

### 3.7

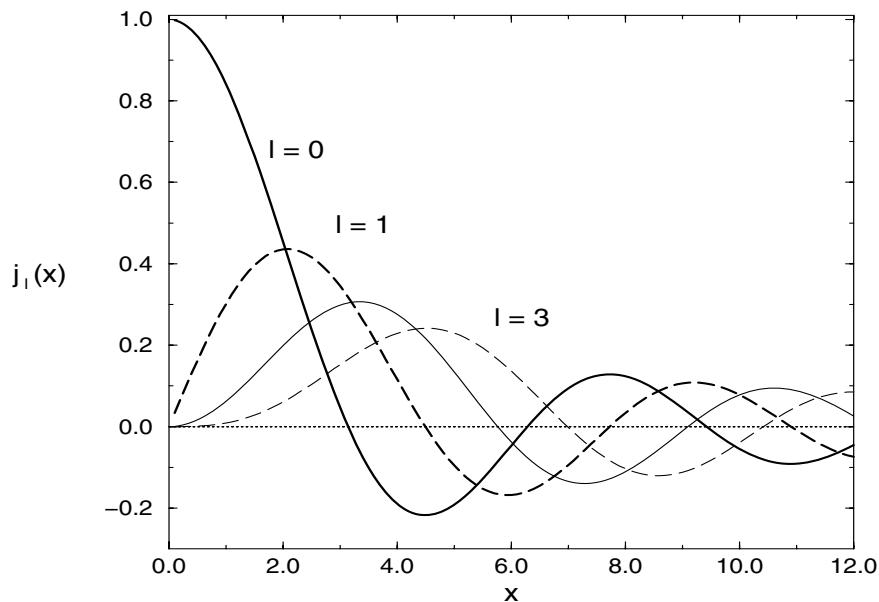
#### Numeric Recursion Relations (Method)

The classic way to calculate  $j_l(x)$  would be by summing its power series for small values of  $x/l$ , and its asymptotic expansion for large values. The approach we adopt here is quicker and has the advantage of generating the spherical Bessel functions for *all*  $l$  values at one time (for fixed  $x$ ). It is based on the *recursion relations*:

$$j_{l+1}(x) = \frac{2l+1}{x} j_l(x) - j_{l-1}(x) \quad (\text{up}) \quad (3.20)$$

$$j_{l-1}(x) = \frac{2l+1}{x} j_l(x) - j_{l+1}(x) \quad (\text{down}) \quad (3.21)$$

Equations (3.20) and (3.21) are the same relation, one written for upward recurrence from small to large  $l$ , and the other for downward recurrence from large to small  $l$ . With just a few additions and multiplications, a recurrence relation permits a rapid and simple computation of the entire set of  $j_l$ 's for fixed  $x$  and all  $l$ .



**Fig. 3.1** The first four spherical Bessel functions  $j_l(x)$  as functions of  $x$ . Notice how for small  $x$ , the values for increasing  $l$  become progressively smaller.

To recur upward for fixed  $x$ , we start with the known forms for  $j_0$  and  $j_1$ , (3.18), and use (3.20). As you yourself will see, this upward recurrence usually starts working pretty well but then fails. The reason for the failure can be seen from the plots of  $j_l(x)$  and  $n_l(x)$  versus  $x$  (Fig. 3.1). If we start at  $x \approx 2$  and  $l = 0$ , then we see that as we recur  $j_l$  up to larger  $l$  values with (3.20), we are essentially taking the difference of two “large” numbers to produce a “small” one. This process suffers from subtractive and always reduces the precision. As we continue recurring, we take the difference of two “small” numbers with large errors and produce a smaller number with yet larger relative error. After a while, we are left with only roundoff error (garbage).

To be more specific, let us call  $j_l^{(c)}$  the numerical value we compute as an approximation for  $j_l(x)$ . Even if we start with pure  $j_l$ , after a short while the computer’s lack of precision effectively mixes in a bit of  $n_l(x)$ :

$$j_l^{(c)} = j_l(x) + \epsilon n_l(x) \quad (3.22)$$

This is inevitable because both  $j_l$  and  $n_l$  satisfy the same differential equation, and on that account, the same recurrence relation. The admixture of  $n_l$  becomes a problem if the numerical value of  $n_l(x)$  is much larger than that of  $j_l(x)$ . Then even a minuscule amount of a very large number may be large.

**Tab. 3.1** Approximate values for spherical Bessel functions of orders 3, 5, and 8 (from Maple).

$x$	$j_3(x)$	$j_5(x)$	$j_8(x)$
0.1	$+9.518519719 \times 10^{-6}$	$+9.616310231 \times 10^{-10}$	$+2.901200102 \times 10^{-16}$
1	$+9.006581118 \times 10^{-3}$	$+9.256115862 \times 10^{-05}$	$+2.826498802 \times 10^{-08}$
10	$-3.949584498 \times 10^{-1}$	$-5.553451162 \times 10^{-01}$	$+1.255780236 \times 10^{+00}$

In contrast, if we use the upward recurrence relation (3.20) to produce the spherical Neumann function  $n_l$ , there is no problem. In that case we are combining small numbers to produce larger ones (Fig. 3.1), a process that does not contain subtractive cancellation, and so we are always working with the most significant parts of the numbers.

The simple solution to this problem (*Miller's device*) is to use (3.21) for downward recursion starting at a large value of  $l$ . This avoids subtractive cancellation by taking small values of  $j_{l+1}(x)$  and  $j_l(x)$  and producing a larger  $j_{l-1}(x)$  by addition. While the error may still behave like a Neumann function, the actual magnitude of the error will *decrease* quickly as we move downward to smaller  $l$  values. In fact, if we start iterating downward with arbitrary values for  $j_{L+1}^{(c)}$  and  $j_L^{(c)}$ , and after a short while we arrive at the correct  $l$  dependence for this value of  $x$ . Although the numerical value of  $j_0^{(c)}$  so obtained will not be correct because it depends upon the arbitrary values assumed for  $j_{L+1}^{(c)}$  and  $j_L^{(c)}$ , we know from the analytic form (3.18) what the value of  $j_0(x)$  should be. In addition, because the recurrence relation is a linear relation between the  $j_l$ 's, we need only normalize all the computed values via

$$j_l^{\text{normalized}}(x) = j_l^{\text{compute}}(x) \times \frac{j_0^{\text{analytic}}(x)}{j_0^{\text{compute}}(x)} \quad (3.23)$$

Accordingly, after you have finished the downward recurrence, you normalize all  $j_l^{(c)}$  values.

### 3.8

#### Implementation and Assessment: Recursion Relations

A program implementing recurrence relations is most easily written using subscripts. If you need to polish up your skills with subscripts, you may want to study our program `Bessel.java` in Listing 3.1 before writing your own.

**Listing 3.1:** The code `Bessel.java` that determines spherical Bessel functions by downward recursion (you should modify this to also work by upward recursion). Note that the comments (beginning with //) are on the right and are ignored by the compiler.

```
// Bessel.java: Spherical Bessels via up and down recursion

import java.io.*;

public class bessel {
    // Global class variables
    public static double xmax = 40., xmin = 0.25, step = 0.1;
    public static int order = 10, start = 50;

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {

        double x;

        PrintWriter w =
            new PrintWriter(new FileOutputStream("Bessel.dat"), true);
            // Step thru x values
        for (x = xmin; x <= xmax; x += step)
            w.println(" " +x+" "+down(x, order, start));
        System.out.println("data stored in Bessel.dat");
    } // End main

    public static double down (double x, int n, int m) { // Recur down
        double scale;
        double j [] = new double[start + 2];
        int k;
                    // Start with anything
        j[m + 1] = j[m] = 1.;
        for (k = m; k>0 ; k--) j[k-1] = ((2.*k+1.)/x)*j[k] - j[k+1];
                    // Scale solution to known j[0]
        scale = (Math.sin(x)/x)/j[0];
        return j[n] * scale;
    }
}
```

1. Write a program that uses both upward and downward recursion to calculate  $j_l(x)$  for the first 25  $l$  values for  $x = 0.1, 1, 10$ .
2. Tune your program so that at least one method gives “good” values (“good” means a relative error  $\simeq 10^{-10}$ ). See Tab. 3.1 for some sample values.
3. Show the convergence and stability of your results.
4. Compare the upward and downward recursion methods, printing out  $l$ ,  $j_l^{(\text{up})}$ ,  $j_l^{(\text{down})}$ , and the relative difference  $|j_l^{(\text{up})} - j_l^{(\text{down})}| / |j_l^{(\text{up})}| + |j_l^{(\text{down})}|$ .
5. The errors in computation depend on  $x$ , and for certain values of  $x$ , both up and down recursions give similar answers. Explain the reason for this and what it tells you about your program.

**3.9****Experimental Error Determination (Problem)**

Numerical algorithms play a vital role in computational physics. You start with a physical theory or mathematical model, you use algorithms to convert the mathematics into a calculational scheme, and, finally, you convert your scheme into a computer program. Your **problem** is to take a general algorithm, and decide

1. Does it converge?
2. How precise are the converged results?
3. How expensive (time consuming) is it?

**3.10****Errors in Algorithms (Model)**

On first thought you may think “What a dumb problem! All algorithms converge if enough terms are used, and if you want more precision then just use more terms.” Well, some algorithms may be asymptotic expansions that just approximate a function in certain regions of parameter space, and only converge up to a point. Yet even if a uniformly convergent power series is used as the algorithm, including more terms will decrease the algorithmic error but increase the roundoff errors. And because the roundoff errors eventually diverge to infinity, the best we can hope for is a “best” approximation. “Good” algorithms are good because they yield an acceptable approximation in a small number of steps, thereby not giving the roundoff error time to grow large.

Let us assume that an algorithm takes  $N$  steps to get a good answer. As a rule of thumb, the approximation (algorithmic) error decreases rapidly, often as the inverse power of the number of terms used:

$$\epsilon_{\text{aprx}} \simeq \frac{\alpha}{N^\beta} \quad (3.24)$$

Here  $\alpha$  and  $\beta$  are empirical constants that would change for different algorithms, and may be only approximately constant as  $N \rightarrow \infty$ . The fact that the error must fall off for large  $N$  is just a statement that the algorithm converges. In contrast to this algorithmic error, roundoff errors tend to grow slowly and somewhat randomly with  $N$ . If the roundoff errors in the individual steps of the algorithm are not correlated, then we know from our previous discussion that we can model the accumulation of error as a random walk with step size equal to the machine precision  $\epsilon_m$ ,

$$\epsilon_{\text{ro}} \simeq \sqrt{N}\epsilon_m \quad (3.25)$$

Indeed this is the slow growth with  $N$  we expect from the roundoff error. The total error in a computation would be the sum of the two:

$$\epsilon_{\text{tot}} = \epsilon_{\text{aprx}} + \epsilon_{\text{ro}} \simeq \frac{\alpha}{N^\beta} + \sqrt{N}\epsilon_m \quad (3.26)$$

We expect the first term to be the larger of the two for small  $N$ , but ultimately to be overcome by the slowly growing second term. As an example, in Fig. 5.4 we present a log–log plot of the relative error in numerical integration using the Simpson integration rule (Chap. 5).

We use the  $\log_{10}$  of the relative error because its negative tells us the number of decimal places of precision obtained.<sup>2</sup> As a case in point, let us assume that  $\mathcal{A}$  is the exact answer and  $A(N)$  the computed answer. If

$$\frac{\mathcal{A} - A(N)}{\mathcal{A}} = 10^{-9} \quad \text{then} \quad \log_{10} \left| \frac{\mathcal{A} - A(N)}{\mathcal{A}} \right| = -9 \quad (3.27)$$

We see in Fig. 5.4 that the error does show a rapid decrease for small  $N$ , consistent with an inverse power law (3.24). In this region the algorithm is converging. Yet as  $N$  keeps increasing, the error starts looking somewhat erratic, yet with a slow increase on average. In accordance with (3.26), in this region the roundoff error has grown larger than the approximation error and will continue to grow for increasing  $N$ . Clearly then, the smallest total error would be obtained if we can stop the calculation at the minimum near  $10^{-14}$ , that is, when  $\epsilon_{\text{aprx}} \simeq \epsilon_{\text{ro}}$ .

In realistic calculation you would not know the exact answer; after all, if you did, then why would you bother with the computation? However, you may know the exact answer for a similar calculation, and you can use that similar calculation to perfect your numerical technique. Alternatively, now that you understand how the total error in a computation behaves, you should be able to look at a table or, better yet, graph (Fig. 5.4) of your answer, and deduce the manner in which your algorithm is converging. Specifically, at some point you should see that your answer begins to change only in the digits several places to the right of the decimal point, with that place moving further to the right as your calculation executes more steps. Eventually, however, as the number of steps gets truly large, the roundoff error should lead to some fluctuation in some high decimal place, and then to a decrease in stability of your answer. You should quit the calculation before this occurs.

Based upon the previous reasoning, another approach is to assume that the exact answer to your problem is  $\mathcal{A}$ , while that obtained by your algorithm after  $N$  steps is  $A(N)$ . The trick then is to examine the behavior of the computed

<sup>2</sup> Note that most computer languages set their log function to  $\ln = \log_e$ . Yet since  $x \stackrel{\text{def}}{=} a^{\log_a x}$ , the conversion to base 10 is simply  $\log_{10} x = \ln x / \ln 10$ .

$A(N)$  for values of  $N$  large enough for the approximation to be converging according to (3.24), that is, so that

$$A(N) \simeq \mathcal{A} + \frac{\alpha}{N^\beta} \quad (3.28)$$

but not that large so that the roundoff error term in (3.26) dominates. We then run our computer program with  $2N$  steps, which should give a better answer still. If the roundoff error is not yet dominating, then we can eliminate the unknown  $\mathcal{A}$  by subtraction:

$$A(N) - A(2N) \approx \frac{\alpha}{N^\beta} \quad (3.29)$$

To see if these assumptions are correct, and determine what level of precision is possible for the best choice of  $N$ , you would plot  $\log_{10}|(A(N) - A(2N))/A(2N)|$  versus  $\log_{10} N$ , similar to what we have done in Fig. 5.4. If you obtain a rapid straight-line drop off, then you know you are in the region of convergence and can deduce a value for  $\beta$  from the slope. As  $N$  gets larger, you should see the graph change from the previous straight line decrease to a slow increase as the roundoff error begins to dominate. Before this is a good place to quit. In any case, now you are in control of your error.

### 3.11

#### Minimizing the Error (Method)

In order to see more clearly how different kinds of errors enter into a computation, let us examine a case where we know  $\alpha$  and  $\beta$ , for example,

$$\epsilon_{\text{aprx}} \simeq \frac{1}{N^2} \quad \Rightarrow \quad \epsilon_{\text{tot}} \simeq \sqrt{N}\epsilon_m \quad (3.30)$$

where the total error is the sum of roundoff and approximation errors. This total error is a minimum when

$$\frac{d\epsilon_{\text{tot}}}{dN} = 0 \Rightarrow N^{\frac{5}{2}} = \frac{4}{\epsilon_m} \quad (3.31)$$

For a single-precision calculation ( $\epsilon_m \simeq 10^{-7}$ ), the minimum total error occurs when

$$N^{\frac{5}{2}} \simeq \frac{4}{10^{-7}} \quad \Rightarrow \quad N \simeq 1099 \quad \Rightarrow \quad \epsilon_{\text{tot}} \simeq 4 \times 10^{-6} \quad (3.32)$$

This shows that for a typical algorithm, most of the error is due to roundoff. Observe, too, that even though this is the minimum error, the best we can do is to get some 40 times machine precision (the double-precision results are better).

Seeing that the total error is mainly the roundoff error  $\propto \sqrt{N}$ , an obvious way to decrease the error is to use a smaller number of steps  $N$ . Let us assume that we do this by finding another algorithm that converges more rapidly with  $N$ , for example, one with the approximation error behaving like

$$\epsilon_{\text{aprx}} \simeq \frac{2}{N^4} \quad (3.33)$$

The total error is now

$$\epsilon_{\text{tot}} = \epsilon_{\text{RO}} + \epsilon_{\text{aprx}} \simeq \frac{2}{N^4} + \sqrt{N}\epsilon_m \quad (3.34)$$

The number of points for the minimum error is found as before,

$$\frac{d\epsilon_{\text{tot}}}{dN} = 0 \quad \Rightarrow \quad N^{\frac{9}{2}} \quad \Rightarrow \quad N \simeq 67 \quad \Rightarrow \quad \epsilon_{\text{tot}} \simeq 9 \times 10^{-7} \quad (3.35)$$

The error is now smaller by a factor of 4, with only  $\frac{1}{16}$  as many steps needed. Subtle are the ways of the computer. In this case it is not that the better algorithm is more elegant, but rather, by being quicker and using fewer steps, it produces less roundoff error.

**Exercise:** Repeat the preceding error estimates for double precision. □

### 3.12

#### Error Assessment

As you have seen in the previous chapter, the mathematical definition of the exponential function  $e^{-x}$  is

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} \quad (3.36)$$

As long as  $x^2 < \infty$ , this series converges in the mathematical sense. Accordingly, a first attempt at an algorithm to compute the exponential might be

$$e^{-x} \simeq \sum_{n=0}^N \frac{(-x)^n}{n!} \quad (3.37)$$

Clearly for (3.37) to be a good algorithm, we need to have the first ignored term,  $(-x)^{N+1}/(N+1)!$ , to be small in comparison to the sum we are computing. This is expected to occur for  $x$  values at which the numerical summation is converging to an answer. In addition, we also need to have the *sum* of all the ignored terms to be small compared to the sum we are keeping, and the sum we are keeping to be a good approximation to the mathematical sum.

While, in principle, it should be faster to see the effects of error accumulation in this algorithm by using single-precision numbers (floats) in your programming, C and Java tend to use double-precision mathematical libraries, and so it is hard to do a pure single-precision computation. Accordingly, do these exercises in double precision, as you should for all scientific calculations involving floating-point numbers.

1. Write a program that calculates  $e^{-x}$  as the finite sum (3.37). (If you have done this in the previous chapter, then you may reuse that program and its results.)
2. Calculate your series for  $x \leq 1$  and compare it to the built-in function `exp(x)` (you may assume that the built-in exponential function is exact). You should pick an  $N$  for which the next term in the series is no more than  $10^{-7}$  of the sum up to that point,

$$\frac{|(-x)^{N+1}|}{(N+1)!} \leq 10^{-7} \left| \sum_{n=0}^N \frac{(-x)^n}{n!} \right| \quad (3.38)$$

3. Examine the terms in the series for  $x \simeq 10$  and observe the significant subtractive cancellations that occur when large terms add together to give small answers. In particular, print out the near-perfect cancellation at  $n \simeq x - 1$ .
4. See if better precision is obtained by being clever and using  $\exp(-x) = 1/\exp(x)$  for large  $x$  values. This eliminates subtractive cancellation, but does not eliminate all roundoff errors.
5. By progressively increasing  $x$  from 1 to 10, and then from 10 to 100, use your program to determine experimentally when the series starts to lose accuracy, and when the series no longer converges.
6. Make a series of graphs of the error versus  $N$  for different values of  $x$ .

Because this series summation is such a simple and correlated process, the roundoff error does not accumulate randomly as it might for a more complicated computation, and we do not obtain the error behavior (3.28). To really see this error behavior, try this test with the integration rules discussed in Chap. 5.

## 4

# Object-Oriented Programming: Kinematics ⊙

In this chapter we provide examples of object-oriented programming (OOP) using the C++ language. Even though this subject fits in well with our earlier discussion of programming principles, we have put it off until now and marked it as optional (the ⊙) since we do not use explicit OOP in the projects and because it is difficult for the new programmer. However, we suggest that everyone read through Section 4.2.1.

### 4.1

#### Problem: Superposition of Motions

The isotropy of space implies that motion in one direction is independent of motion in other directions. So, for example, when a soccer ball is kicked, we have acceleration in the vertical direction and simultaneous, yet independent, uniform motion in the horizontal direction. In addition, Galilean invariance (velocity independence of Newton's laws of motion) tells us that when an acceleration is added in to uniform motion, the distance covered due to the acceleration adds to the distance covered by uniform motion.

Your **problem** is to use the computer to describe motion in such a way that velocities and accelerations in each direction are treated as separate entities or objects, independent of motion in other directions. In this way the problem is viewed consistently by both the programming philosophy and the basic physics.

### 4.2

#### Theory: Object-Oriented Programming

We will analyze this problem from an object-oriented programming viewpoint. While the *objects* in OOP are often graphical on a computer screen, the objects in our problem are the motions in each dimension. By reading through and running the programs in this project, you should become familiar with the concepts of OOP.

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5

## 4.2.1

**OOP Fundamentals**

Object-oriented programming (OOP) has a precise definition. The concept is general and the *object* can be a component of a program with the properties of *encapsulation*, *abstraction*, *inheritance*, and *polymorphism* (to be defined shortly). Of interest to us is OOP's programming paradigm, which aims to simplify writing large programs by providing a framework for reusing components developed and tested in previous problems. A true object-oriented language has four characteristics [5,6]:

**Encapsulation:** The data and the *methods* used to produce or access the data are encapsulated into an entity called an *object*. For our 1D problem, we take the data as the initial position and velocity of the soccer ball, and the object as the solution  $x(t)$  of the equations of motion that gives the position of the ball as a function of time. As part of the OOP philosophy, data are manipulated only via distinct *methods*.

**Abstraction:** Operations applied to objects must give expected results according to the nature of the objects. For example, summing two matrices always gives another matrix. The programmer can in this way concentrate on solving the problem rather than on details of implementation.

**Inheritance:** Objects inherit characteristics (including code) from their ancestors, yet may be different from their ancestors. In our problem, motion in two dimensions inherits the properties of 1D motion in each of two dimensions, and accelerated motion inherits the properties of uniform motion.

**Polymorphism:** Different objects may have *methods* with the same name, yet the method may differ for different objects. Child objects may have *member* functions with the same name but properties differing from those of their ancestors. In our problem, a member function *archive*, which contains the data to be plotted, will be redefined depending on whether the motion is uniform or accelerated.

## 4.3

**Theory: Newton's Laws, Equation of Motion**

Newton's second law of motion relates the force vector  $\mathbf{F}$  acting on a mass  $m$  to the acceleration vector  $\mathbf{a}$  of the mass:

$$\mathbf{F} = m\mathbf{a} \tag{4.1}$$

When the vectors are resolved into Cartesian components, each component yields a second-order differential equation:

$$F_i = m \frac{d^2 x_i}{dt^2} \quad (i = 1, 2, 3) \quad (4.2)$$

If the force in the  $x$  direction vanishes,  $F_x = 0$ , the equation of motion (4.2) has a solution corresponding to uniform motion in the  $x$  direction with a constant velocity  $v_{0x}$ :

$$x = x_0 + v_{0x}t \quad (4.3)$$

Equation (4.3) is the *base* or *parent* object in our project. If the force in the  $y$  direction also vanishes, then there also will be uniform motion in the  $y$  direction:

$$y = y_0 + v_{0y}t \quad (4.4)$$

In our project we consider uniform  $x$  motion as a parent and view the  $y$  motion as a child.

Equation (4.2) tells us that a constant force in the  $x$  direction causes a constant acceleration  $a_x$  in that direction. The solution of the  $x$  equation of motion with uniform acceleration is

$$x = x_0 + v_{0x}t + \frac{1}{2}a_x t^2 \quad (4.5)$$

For projectile motion without air resistance, we usually have no  $x$  acceleration and a negative  $y$  acceleration due to gravity,  $a_y = -g = -9.8 \text{ m/s}^2$ . The  $y$  equation of motion is then

$$y = y_0 + v_{0y}t - \frac{1}{2}gt^2 \quad (4.6)$$

We define this accelerated  $y$  motion to be a child to the parent uniform  $x$  motion.

#### 4.4

##### OOP Method: Class Structure

The *class structure* we use to solve our problem contains the objects:

<b>Parent class Um1D:</b>	1D uniform motion for given initial conditions,
<b>Child class Um2D:</b>	2D uniform motion for given initial conditions,
<b>Child class Am2d:</b>	2D accelerated motion for given initial conditions.

The *member functions* include

x:	position after time $t$ ,
archive:	creator of a file of position versus time.

For our projectile motion, *encapsulation* is the combination of the initial conditions ( $x_0, v_{x0}$ ) with the member functions used to compute  $x(t)$ . Our member functions are the creator of the class of uniform 1D motion `Um1D`, its destructor  $\sim$  `Um1D`, and the creator `x(t)` of a file of  $x$  as a function of time  $t$ . *Inheritance* is the child class `Um2D` for uniform motion in both  $x$  and  $y$  directions, it being created from the parent class `Um1D` of 1D uniform motion. *Abstraction* is present (although not used powerfully) by the simple addition of motion in the  $x$  and  $y$  directions. Polymorphism is present by having the member function that creates the output file be different for 1D and 2D motions. In our implementation of OOP, the class `Am2D` for accelerated motion in two dimensions inherits uniform motion in two dimensions (which, in turn, inherits uniform 1D motion), and adds to it the attribute of acceleration.

#### 4.5

##### Implementation: Uniform 1D Motion, `unim1d.cpp`

For 1D motion we need a program that outputs positions along a line as a function of time,  $(x, t)$ . For 2D motion we need a program that outputs positions in a plane as a function of time  $(x, y, t)$ . Time varies in discrete steps of  $\Delta t = \text{delt} = T/N$ , where the total time  $T$  and the number of steps  $N$  are input parameters. We give here program fragments that can be pasted together into a complete C++ program (the complete program is on the diskette).

Our parent class `Um1D` of uniform motion in one dimension contains

x00:	the initial position,
delt:	the time step,
vx0:	the initial velocity,
time:	the total time of the motion,
steps:	the number of time steps.

To create it, we start with the C++ headers:

#include <stdio.h>	/* Input-output libe */
#include <stdlib.h>	/* Math libe */

The encapsulation of the data and the member functions is achieved via Class `Um1D`:

## 4.5.1

**Uniform Motion in 1D, Class Um1D**

```

class Um1D {                                     /* Create base class
*/
public:
    double x00, delt, vx, time;      /* Initial position , velocity , dt */
    int steps;                      /* Time steps */
    Um1D(double x0, double dt, double vx0, double ttot); /* Constructor */
    ~Um1D(void);                  /* Class Destructor */

    double x(double tt);           /* x(t) */
    void archive();                /* send x vs t to file */
};

```

Next, the variables `x0`, `delt`, `vx`, and `time` are initialized by the constructor of the class `Um1D`:

```

Um1D::Um1D(double x0, double dt, double vx0, double ttot) {
    /* Constructor Um1D */
    x00 = x0;
    delt = dt;
    vx = vx0;
    time = ttot;
    steps = ttot/delt;
}

```

After that, we make the destructor of the class, which also prints the time when the class is destroyed:

```

Um1D::~Um1D(void) {
    /* Destructor of class Um1D */
    printf("Class Um1D destroyed \n");
}

```

Given the initial position  $x_0$ , the member function returns the position after time  $dt$ :

```

double Um1D::x(double tt) {                   /* x=x0+dt*v
*/
    return x00+tt*vx;
}

```

The algorithm is implemented in a member routine, and the positions and times are written to the file `Motion1D.dat`:

```

void Um1D::archive() {                         /* Produce x vs t file
*/
    FILE *pf;
    int i;
    double xx, tt;
    if ( (pf = fopen("Motion1D.dat","w+")) == NULL ) {
        printf("Could not open file \n");
        exit(1);
    }
}

```

```

}
tt = 0.0;
for ( i = 1; i <= steps; i++ ) {
    xx = x(tt);                                /* Computes x=x0+t*v */

    fprintf(pf, " %f  %f \n", tt ,xx);
    tt = tt+delt;
}
fclose(pf);
}

```

The main program defines an object (class) `unimotx` of type `Um1D` and gives initial numeric values to the data:

```

main() {
    double inix , inivx , dtim , ttotal ;
    inix = 5.0;
    dtim = 0.1;
    inivx = 10.0;
    ttotal = 4.0;
    Um1D unimotx(inix , dtim , inivx , ttotal); /* Class constructor */
    unimotx.archive();                            /* Produce y vs x file */
}

```

#### 4.5.2

#### Implementation: Uniform Motion in 2D, Child `Um2D`, `unimot2d.cpp`

The first part of the program is the same as before. We now make the child class `Um2D` from the class `Um1D`.

```

#include <stdio.h> #include <stdlib.h> class Um1D {
/* Base class created */
public:
    double x00 , delt , vx , time; /* Initial conditions , parameters */
                                         /* Time step */
    int steps;                      /* constructor */
    Um1D(double x0, double dt, double vx0, double ttot); /* Class destructor */
    ~Um1D(void);                   /* Send x vs t to file */

    double x(double tt);           /* x=x0+v*t */
    void archive();                /* Um1D Constructor */
};                                /* Class Um1D destructor */
Um1D::Um1D(double x0, double dt, double vx0, double ttot) {
    x00 = x0;
    delt = dt;
    vx = vx0;
    time = ttot;
    steps = ttot/delt;
}
Um1D::~Um1D(void) {                /* Class Um1D destroyed \ n" */
    printf("Class Um1D destroyed \ n");
}
double Um1D::x(double tt) {         /* x=x0+dt*v */

```

```

    return x00+tt*vx;
}
void Um1D::archive() { /* Produce x vs t file */
    FILE *pf;
    int i;
    double xx, tt;
    if ( (pf = fopen("Motion1D.dat", "w+"))==NULL ) {
        printf("Could not open file \n");
        exit(1);
    }
    tt = 0.0;
    for ( i = 1; i <= steps; i++ ) {
        xx = x(tt); /* computes x=x0+t*v */
        fprintf(pf,"%f %f \n", tt, xx);
        tt = tt+delt;
    }
    fclose(pf);
}

```

#### 4.5.3

##### **Class Um2D: Uniform Motion in 2D**

To include another degree of freedom, we define a new class that inherits the *x* component of uniform motion from the parent *Um1D*, as well as the *y* component of uniform motion from the parent *Um1D*. The new data for the class are the initial *y* position *y00* and the velocity in the *y* direction *vy0*. A new member *y* is included to describe the *y* motion. Note, in making the constructor of the *Um2D* class, that our interest in the data *y* versus *x* leads to the member *archive* being redefined. This is polymorphism in action.

```

class Um2D : public Um1D { /* Child class , parent Um1D */
public: /* Data accessible to other code */
    double y00, vy; /* member functions accessible to all */
    Um2D(double x0,double dt,double vx0,double ttot,double y0,double
          vy0);
    ~Um2D(void); /* destructor of Um2D class */
    double y(double tt); /* Added motion for 2D */
    void archive(); /* redefinition for 2D */
};

```

Observe how the *Um2D* constructor initializes the data in *Um1D*, *y00*, and *vy*:

```

Um2D::Um2D(double x0, double dt, double vx0, double tott,
           double y0, double vy0):
    Um1D(x0,dt,vx0,tott) {
    y00 = y0;
    vy = vy0;
}

```

The destructor of the new class is

```
Um2D::~Um2D(void) {
    printf("Class Um2D is destroyed \n");
}
```

The new member of class `Um2D` accounts for the  $y$  motion:

```
double Um2D::y(double tt) {
    return y00+tt*vy;
}
```

The new member function for two dimensions contains the data of the  $y$  and  $x$  positions, and incorporates the polymorphism property of the objects:

```
void Um2D::archive() { /* Uniform motion in 2D */
    FILE *pf;
    int i;
    double xx, yy, tt;
    if ( (pf = fopen("Motion2D.dat","w+")) == NULL ) {
        printf("Could not open file \n");
        exit(1);
    }
    tt = 0.0;
    for (i = 1; i <= steps; i++) {
        xx = x(tt); /* uses member function x */
        yy = y(tt); /* add the second dimension */
        fprintf(pf,"%f %f \n", yy, xx);
        tt = tt + delt;
    }
    fclose(pf);
}
```

The differences with the previous `main` program are the inclusion of the  $y$  component of the motion and the constructor `unimotxy` of class type `Um2D`:

```
main() {
    double inix, iniy, inivx, inivy, dtim, ttotal;
    inix = 5.0;
    dtim = 0.1;
    inivx = 10.0;
    ttotal = 4.0;
    iniy = 3.0;
    inivy = 8.0;
    /* class constructor */
    Um2D unimotxy(inix, dtim, inivx, ttotal, iniy, inivy);
    unimotxy.archive(); /* To obtain file of y vs x */
}
```

## 4.5.4

**Implementation: Projectile Motion, Child Accm2D, *accm2d.cpp***

Consider now the problem of the motion of a soccer ball kicked at  $(x, y) = (0, 0)$  with initial velocity  $(v_{0x}, v_{0y}) = (14, 14)$  m/s. This is, of course, motion of a projectile in a uniform gravitational field which we know is described by a parabolic trajectory. We define a new child class `Accm2D` derived from the parent class `Um2D` that adds acceleration to the motion. The first part of the program is the same as the one for uniform motion in two dimensions, but now there is an extension with the class `Accm2D`:

```
#include <stdio.h> #include <stdlib.h> class Um1D {
/* Base class created */
public:
    double x00, delt, vx, time;      /* Initial conditions, parameters */
    int steps;                      /* Time steps to write in file */
    /* Constructor */
    Um1D(double x0, double dt, double vx0, double ttot);
    ~Um1D(void);                  /* Destructor */
    /* x=x0+v* dt */
    void archive();                /* send x vs t to file */
};

/* Constructor */
Um1D::Um1D(double x0, double dt, double vx0, double ttot) {
    x00 = x0;
    delt = dt;
    vx = vx0;
    time = ttot;
    steps = ttot/delt;
}

Um1D::~Um1D(void) {             /* Destructor of the class Um1D */
    printf("Class Um1D destroyed \n");
}

double Um1D::x(double tt) {      /* x=x0+dt*v */
    return x00+tt*vx;
}

void Um1D::archive() {           /* Produce file of x vs t */
    FILE *pf;
    int i;
    double xx, tt;
    if ( (pf = fopen("Motion1D.dat","w+")) == NULL ) {
        printf("Could not open file \n ");
        exit(1);
    }
    tt=0.0;
    for (i = 1; i<= steps; i++) { /* x=x0+dt*v, change x0 */
        xx = x(tt);
        fprintf(pf,"%f %f \n ",tt,xx);
        tt = tt+delt;
    }
}
```

```

        fclose(pf);
    }

class Um2D : public Um1D {      /* Child class Um2D, parent Um1D */
    public:                      /* Data: code; functions: all members */
    double y00,vy;
                                /* constructor of Um2D class */
    Um2D(double x0, double dt, double vx0, double ttot, double y0,
         double vy0);
    ~Um2D(void);
                                /* destructor of Um2D class */
    double y(double tt);
    void archive();           /* redefine member for 2D */
};

/* Construct class Um2D */
Um2D::Um2D(double x0, double dt, double vx0, double ttot,
            double y0, double vy0):Um1D(x0, dt, vx0, ttot) {
    y00 = y0;
    vy = vy0;
}

Um2D::~Um2D(void) {
    printf("Class Um2D is destroyed \n ");
}

double Um2D::y(double tt) {
    return y00+tt*vy;
}

void Um2D::archive() {
    FILE *pf;
    int i;
    double xx, yy, tt;
                                /* now in 2D, still uniform */
    if ( (pf = fopen("Motion2D.dat","w+")) == NULL ) {
        printf("Could not open file \n ");
        exit(1);
    }
    tt = 0.0;
    for ( i = 1; i <= steps; i++ ) {
        xx = x(tt);                /* uses member function x */
        yy = y(tt);                /* adds second dimension */
        fprintf(pf,"%f %f \n ",yy,xx );
        tt = tt+delt;
    }
    fclose(pf);
}

```

#### 4.5.5

##### Accelerated Motion in Two Directions

A child class `Accm2D` is created from the parent class `Um2D`. It will inherit uniform motion in two dimensions and add acceleration in both the  $x$  and  $y$  directions. It has member functions:

1. Constructor of class.
2. Destructor of class.
3. A new member `xy` that gives the  $x$  and  $y$  components of acceleration.
4. An *archive* to override the member function of same name in the parent class for 2D uniform motion.

```
class Accm2D : public Um2D {
public:
    double ax, ay;
    Accm2D(double x0, double dt, double vx0, double ttot, double y0,
           double vy0, double accx, double accy);
    ~Accm2D(void);
    void xy(double *xxac, double *yyac, double tt);
    void archive();
};
```

Observe the method used to initialize the classes `Am2d` and `Um2D`:

```
Accm2D::Accm2D(double x0, double dt, double vx0, double ttot,
                double y0, double vy0, double accx, double accy):
    Um2D(x0, dt, vx0, ttot, y0, vy0) {
    ax = accx;
    ay = accy;
}

Accm2D::~Accm2D(void) {
    printf(" Class Accm2D destroyed \n ");
}
```

Next we introduce a member function to show the inheritance of the parent class functions `xpox` and `y` and to include the two components of acceleration:

```
void Accm2D::xy(double *xxac, double *yyac, double tt) {
    double dt2;
    dt2 = 0.5*tt*tt;
    *xxac = x(tt) + ax*dt2;
    *yyac = y(tt) + ay*dt2;
}
```

To override *archive* (which creates the data file), we redefine *archive* to take into account the acceleration:

```
void Accm2D::archive() {
    FILE *pf;
    int i;
    double tt, xxac, yyac;
    if ( (pf = fopen("Motion.dat", "w+")) == NULL ) {
        printf("Could not open file \n ");
        exit(1);
    }
```

```

tt = 0.0;
for(i = 1;i<=steps;i++) {
    xy(&xxac, &yyac, tt);
    fprintf(pf,"%f %f \n ",xxac, yyac);
    tt = tt + delt;
}
fclose(pf);
}

```

Next a file is produced with the  $y$  and  $x$  positions of the ball as functions of time:

```

main() {
    double inix, iniy, inivx, inivy, aclx, acly, dtim, ttotal;
    inix = 0.0;
    dtim = 0.1;
    inivx = 14.0;
    ttotal = 4.0;
    iniy = 0.0;
    inivy = 14.0;
    aclx = 0.0;
    acly = -9.8;
    Accm2D acmo2d(inix, dtim, inivx, ttotal, iniy, inivy, aclx, acly);
    printf(" \n ");
    printf(" \n ");
    acmo2d.archive();
}

```

## 4.6

### Assessment: Exploration, shms.cpp

The superposition of independent simple harmonic motion in each of two dimensions can be studied with OOP. Define a class `ShmX` for harmonic motion in the  $x$  direction:

$$x = A_x \sin(\omega_x t + \phi_x) \quad (4.7)$$

where  $A_x$  is the amplitude,  $\omega_x$  is the angular frequency, and  $\phi_x$  is the phase. Define another class `ShmY` for independent harmonic motion in the  $y$  direction:

$$y = A_y \sin(\omega_y t + \phi_y) \quad (4.8)$$

Now employ the concept of *multiple inheritance* to define a child class `ShmXY` of both `ShmX` and `ShmY`. It should have a member function to write a file with the  $x$  and  $y$  positions at several times (which can then be used to plot Lissajous figures). To obtain multiple inheritance use `class ShmXY : public ShmX, public ShmY`. To obtain the constructor, use something like this:

```
ShmXY::ShmXY(double Axi, double delx, double wx, double tx,  
    double dtx, double Ayi, double dely, double wy, double ty, double  
    dty)  
: ShmX(Axi, delx, wx, tx, dtx), ShmY(Ayi, dely, wy, ty, dty)
```

## 5

# Integration

### 5.1

#### Problem: Integrating a Spectrum

**Problem:** An experiment has measured  $dN(t)/dt$ , the number of particles per unit time entering a counter. Your **problem** is to integrate this spectrum to obtain the number of particles  $N(1)$  that entered the counter in the first second:

$$N(1) = \int_0^1 \frac{dN(t)}{dt} dt \quad (5.1)$$

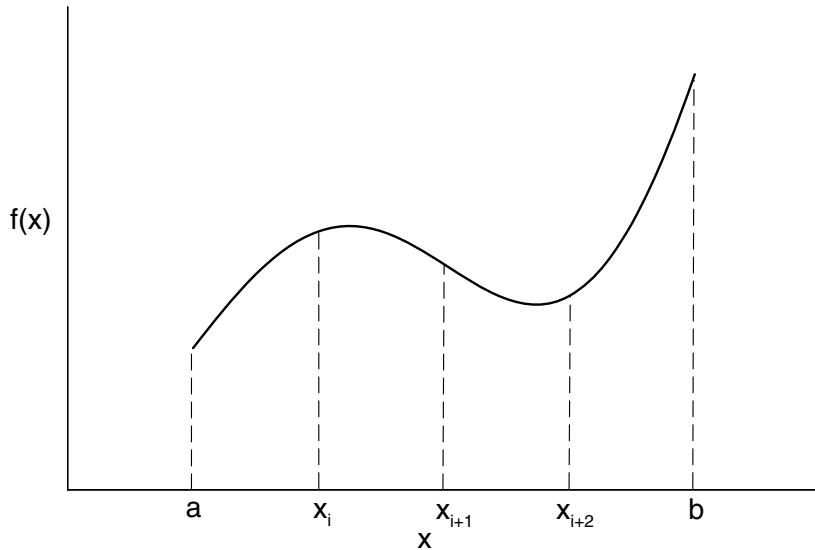
Although the integrand we will give you later can be integrated analytically, we wish to evaluate (5.1) for an arbitrary integrand.

### 5.2

#### Quadrature as Box Counting (Math)

The integration of a function may require some cleverness to do analytically, but it is relatively straightforward on a computer. A traditional way to do numerical integration by hand is to take a piece of graph paper and count the number of boxes or *quadrilaterals* lying below a curve of the integrand. For this reason numerical integration is also called *numerical quadrature*, even when it becomes more sophisticated than simple box counting. The Riemann definition of an integral is the limit of the sum over boxes as the width  $h$  of the box approaches zero:

$$\int_a^b f(x) dx = \lim_{h \rightarrow 0} \left[ h \sum_{i=1}^{(b-a)/h} f(x_i) \right] \quad (5.2)$$



**Fig. 5.1** The integral  $\int_a^b f(x) dx$  is the area under the graph of  $f(x)$  from  $a$  to  $b$ . Here we break up the area into four regions of equal widths.

The numerical integral of a function  $f(x)$  is approximated as the equivalent of a finite sum over boxes of height  $f(x)$  and width  $w_i$ :

$$\int_a^b f(x) dx \approx \sum_{i=1}^N f(x_i) w_i \quad (5.3)$$

which is similar to the Riemann definition (5.2), except that there is no limit to infinitesimal box size. Equation (5.3) is the standard form for all integration algorithms; the function  $f(x)$  is evaluated at  $N$  points in the interval  $[a, b]$ , and the function values  $f_i \equiv f(x_i)$  are summed with each term in the sum weighted by  $w_i$ . While, in general, the sum in (5.3) will give the exact integral only when  $N \rightarrow \infty$ , for polynomials it may be exact for finite  $N$ . The different integration algorithms amount to different ways of choosing the points and weights. Generally, the precision increases as  $N$  gets larger, with the round-off error eventually limiting the increase. Because the “best” approximation depends on the specific behavior of  $f(x)$ , there is no universally best approximation. In fact, some of the automated integration schemes found in subroutine libraries will switch from one method to another until they find one that works well.

In general, you should not attempt a numerical integration of an integrand that contains a singularity without first removing the singularity by hand.<sup>1</sup>

<sup>1</sup> In Chap. 30 we show how to remove such a singularity even when the integrand is unknown.

You may be able to do this very simply by breaking the interval down into several subintervals, so the singularity is at an endpoint where a Gauss point never falls, or by a change of variable:

$$\int_{-1}^1 |x| f(x) dx = \int_{-1}^0 f(-x) dx + \int_0^1 f(x) dx \quad (5.4)$$

$$\int_0^1 x^{1/3} dx = \int_0^1 3y^3 dy \quad (y = x^{1/3}) \quad (5.5)$$

$$\int_0^1 \frac{f(x) dx}{\sqrt{1-x^2}} = 2 \int_0^1 \frac{f(1-y^2) dy}{\sqrt{2-y^2}} \quad (y^2 = 1-x). \quad (5.6)$$

Likewise, if your integrand has a very slow variation in some region, you can speed up the integration by changing to a variable that compresses that region and places few points there. Conversely, if your integrand has a very rapid variation in some region, you may want to change to variables that expand that region to ensure that no oscillations are missed.

**Tab. 5.1** Elementary weights for uniform-step integration rules.

Name	Degree	Elementary weights
Trapezoid	1	$(1, 1) \frac{h}{2}$
Simpson's	2	$(1, 4, 1) \frac{h}{3}$
	$\frac{3}{8}$	$(1, 3, 3, 1) \frac{3}{8}h$
Milne	4	$(14, 64, 24, 64, 14) \frac{h}{45}$

### 5.3

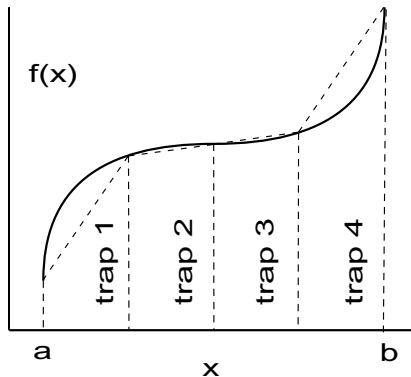
#### Algorithm: Trapezoid Rule

The trapezoid and Simpson integration rules use values of  $f(x)$  at evenly spaced values of  $x$ . They use  $N$  points  $x_i$  ( $i = 1, N$ ), evenly spaced at a distance  $h$  apart throughout the integration region  $[a, b]$  and *include the endpoints*. This means that there are  $N - 1$  intervals of length  $h$ :

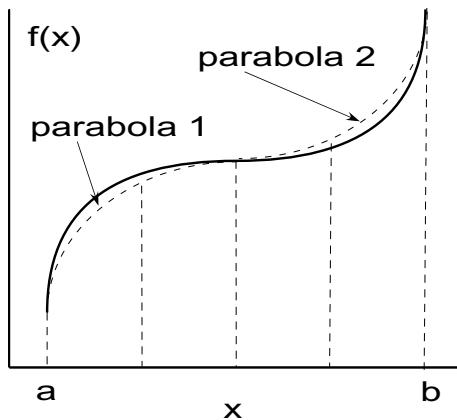
$$h = \frac{b-a}{N-1} \quad x_i = a + (i-1)h \quad i = 1, N \quad (5.7)$$

Notice that we start our counting at  $i = 1$ , and that Simpson's rule requires an *odd* number of points  $N$ .

The trapezoid rule takes the integration interval  $i$  and constructs a trapezoid of width  $h$  in it (Fig. 5.2). This approximates  $f(x)$  by a straight line in that interval  $i$ , and uses the average height  $(f_i + f_{i+1})/2$  as the value for  $f$ . The



**Fig. 5.2** Straight-line sections used for the trapezoid rule.



**Fig. 5.3** Two parabolas used in Simpson's rule.

area of a single trapezoid is in this way

$$\int_{x_i}^{x_i+h} f(x)dx \approx \frac{h(f_i + f_{i+1})}{2} = \frac{1}{2}hf_i + \frac{1}{2}hf_{i+1} \quad (5.8)$$

In terms of our standard integration formula (5.3), the “rule” in (5.8) is for  $N = 2$  points with weight  $w_i \equiv \frac{1}{2}$  (Tab. 5.1).

In order to apply the trapezoid rule to the entire region  $[a, b]$ , we add the contributions from each subinterval:

$$\int_a^b f(x)dx \approx \frac{h}{2}f_1 + hf_2 + hf_3 + \cdots + hf_{N-1} + \frac{h}{2}f_N \quad (5.9)$$

You will notice that because each internal point gets counted twice, it has a weight of  $h$ , whereas the endpoints get counted just once and on that account have weights of only  $h/2$ . In terms of our standard integration rule (5.32), we

have

$$w_i = \left\{ \frac{h}{2}, h, \dots, h, \frac{h}{2} \right\} \quad (5.10)$$

In Listing 5.1 we provide a simple implementation of the trapezoid rule.

**Listing 5.1:** The program `Trap.java` integrates the function  $f(t) = t^2$  via the trapezoid rule. Note how the step size  $h$  depends on the interval and how the endpoint weights are set.

```
// Trap.java trapezoid-rule integration of parabola

public class Trap {
    public static final double A = 0., B = 3.;      // Constant endpoints
    public static final int N = 100;                  // N points (not intervals)

    public static void main(String[] args) {          // Main does summation

        double sum, h, t, w;
        int i;

        h = (B - A) / (N - 1);                      // Initialization
        sum = 0.;

        for (i=1 ; i <= N; i=i + 1) {
            t = A + (i-1) * h;
            if (i==1 || i==N) w = h/2.; else w = h;      // End wt=h/2
            sum = sum + w * t * t;
        }
        System.out.println(sum);
    }
}

// OUTPUT
9.000459136822773
```

## 5.4

### Algorithm: Simpson's Rule

For each interval, Simpson's rule approximates the integrand  $f(x)$  by a parabola (Fig. 5.3):

$$f(x) \approx \alpha x^2 + \beta x + \gamma \quad (5.11)$$

with the intervals still equally spaced. The area of each section is then the integral of this parabola

$$\int_{x_i}^{x_i+h} (\alpha x^2 + \beta x + \gamma) dx = \frac{\alpha x^3}{3} + \frac{\beta x^2}{2} + \gamma x \Big|_{x_i}^{x_i+h} \quad (5.12)$$

This is equivalent to integrating the Taylor series up to the quadratic term. In order to relate the parameters  $\alpha$ ,  $\beta$ , and  $\gamma$  to the function, we consider an

interval from  $-1$  to  $+1$ , in which case

$$\int_{-1}^1 (\alpha x^2 + \beta x + \gamma) dx = \frac{2\alpha}{3} + 2\gamma \quad (5.13)$$

But we notice that

$$\begin{aligned} f(-1) &= \alpha - \beta + \gamma & f(0) &= \gamma & f(1) &= \alpha + \beta + \gamma \\ \Rightarrow \alpha &= \frac{f(1)+f(-1)}{2} - f(0) & \beta &= \frac{f(1)-f(-1)}{2} & \gamma &= f(0) \end{aligned} \quad (5.14)$$

In this way we can express the integral as the weighted sum over the values of the function at three points:

$$\int_{-1}^1 (\alpha x^2 + \beta x + \gamma) dx = \frac{f(-1)}{3} + \frac{4f(0)}{3} + \frac{f(1)}{3} \quad (5.15)$$

Because three values of the function are needed, we generalize this result to our problem by evaluating the integral over two adjacent intervals, in which case we evaluate the function at the two endpoints and the middle (Tab. 5.1):

$$\begin{aligned} \int_{x_i-h}^{x_i+h} f(x) dx &= \int_{x_i}^{x_i+h} f(x) dx + \int_{x_i-h}^{x_i} f(x) dx \\ &\simeq \frac{h}{3} f_{i-1} + \frac{4h}{3} f_i + \frac{h}{3} f_{i+1} \end{aligned} \quad (5.16)$$

Simpson's rule requires the elementary integration to be over *pairs* of intervals, which in turn requires that the total number of intervals be even or the number of points  $N$  be odd. In order to apply Simpson's rule to the entire interval, we add up the contributions from each pair of subintervals, counting all but the first and last endpoints twice:

$$\int_a^b f(x) dx \approx \frac{h}{3} f_1 + \frac{4h}{3} f_2 + \frac{2h}{3} f_3 + \frac{4h}{3} f_4 + \dots + \frac{4h}{3} f_{N-1} + \frac{h}{3} f_N \quad (5.17)$$

In terms of our standard integration rule (5.3), we have

$$w_i = \left\{ \frac{h}{3}, \frac{4h}{3}, \frac{2h}{3}, \frac{4h}{3}, \dots, \frac{4h}{3}, \frac{h}{3} \right\} \quad (5.18)$$

The sum of these weights provides a useful check on your integration:

$$\sum_{i=1}^N w_i = (N-1)h \quad (5.19)$$

*Remember* that the number of points  $N$  must be odd for Simpson's rule.

## 5.5

### Integration Error (Analytic Assessment)

In general, you want to choose an integration rule that gives an accurate answer using the least number of integration points. We obtain a feel for the absolute *approximation* or *algorithmic error*  $E$  and the relative error  $\epsilon$ , by expanding  $f(x)$  in a Taylor series around the midpoint of the integration interval. We then multiply that error by the number of intervals  $N$  to estimate the error for the entire region  $[a, b]$ . For the trapezoid and Simpson's rules this yields

$$E_t = O\left(\frac{[b-a]^3}{N^2}\right) f^{(2)} \quad E_s = O\left(\frac{[b-a]^5}{N^4}\right) f^{(4)} \quad \epsilon_{t,s} = \frac{E_{t,s}}{f} \quad (5.20)$$

We see that the third derivative term in Simpson's rule cancels (much like the central difference method in differentiation). Equations (5.20) are illuminating by showing how increasing the sophistication of an integration rule leads to an error that falls off with a higher inverse power of  $N$ , yet that is also proportional to higher derivatives of  $f$ . Consequently, for small intervals and  $f(x)$  functions with well-behaved high derivatives, Simpson's rule should converge more rapidly than the trapezoid rule.

To be more specific, we assume that after  $N$  steps the *relative* roundoff error is random and of the form

$$\epsilon_{ro} \approx \sqrt{N}\epsilon_m \quad (5.21)$$

where  $\epsilon_m$  is the machine precision,  $\epsilon \sim 10^{-7}$  for single precision and  $\epsilon \sim 10^{-15}$  for double precision. Because most scientific computations are done with doubles, we will assume double precision. We want to determine an  $N$  that minimizes the total error, that is, the sum of the approximation and roundoff errors,

$$\epsilon_{tot} = \epsilon_{ro} + \epsilon_{approx} \quad (5.22)$$

This occurs, approximately, when the two errors are of equal magnitude, which we approximate even further by assuming that the two errors are equal:

$$\epsilon_{ro} = \epsilon_{approx} = \frac{E_{trap,simp}}{f} \quad (5.23)$$

To continue the search for optimum  $N$  for a general function  $f$ , we set the scale of function size and the lengths by assuming

$$\frac{f^{(n)}}{f} \approx 1 \quad b - a = 1 \quad \Rightarrow \quad h = \frac{1}{N} \quad (5.24)$$

The estimate (5.23), when applied to the **trapezoid rule**, yields

$$\sqrt{N}\epsilon_m \approx \frac{f^{(2)}(b-a)^3}{fN^2} = \frac{1}{N^2} \quad (5.25)$$

$$\Rightarrow N \approx \frac{1}{(\epsilon_m)^{2/5}} = (1/10^{-15})^{2/5} = 10^6, \quad (5.26)$$

$$\Rightarrow \epsilon_{\text{ro}} \approx \sqrt{N}\epsilon_m = 10^{-12} \quad (5.27)$$

The estimate (5.23), when applied to **Simpson's rule** yields

$$\sqrt{N}\epsilon_m = \frac{f^{(4)}(b-a)^5}{fN^4} = \frac{1}{N^4}, \quad (5.28)$$

$$\Rightarrow N = \frac{1}{(\epsilon_m)^{2/9}} = (1/10^{-15})^{2/9} = 2154, \quad (5.29)$$

$$\Rightarrow \epsilon_{\text{ro}} \approx \sqrt{N}\epsilon_m = 5 \times 10^{-14} \quad (5.30)$$

These results are illuminating in that they show how

- Simpson's rule is an improvement over the trapezoid rule;
- it is possible to obtain an error close to machine precision with Simpson's rule (and with other higher order integration algorithms);
- obtaining the best numerical approximation to an integral is not obtained by letting  $N \rightarrow \infty$ , but with a relatively small  $N \leq 1000$ .

**Tab. 5.2** Types of Gaussian integration rules.

Integral	Name	Integral	Name
$\int_{-1}^1 f(y)dy$	Gauss	$\int_{-1}^1 \frac{F(y)}{\sqrt{1-y^2}}dy$	Gauss–Chebyshev
$\int_{-\infty}^{\infty} e^{-y^2} F(y)dy$	Gauss–Hermite	$\int_0^{\infty} e^{-y} F(y)dy$	Gauss–Laguerre
$\int_0^{\infty} \frac{e^{-y}}{\sqrt{y}} F(y)dy$	Associated Gauss–Laguerre		

## 5.6

### Algorithm: Gaussian Quadrature

It is often useful to rewrite the basic integration formula (5.3) such that we separate a weighting function  $W(x)$  from the integrand:

$$\int_a^b f(x)dx \equiv \int_a^b W(x)g(x)dx \approx \sum_{i=1}^N w_i g(x_i) \quad (5.31)$$

In the Gaussian quadrature approach to integration, the  $N$  points and weights are chosen to make the approximation error actually vanish for  $g(x)$ , a  $2N - 1$  degree polynomial. To obtain this incredible optimization, the points  $x_i$  end up having a very specific distribution over  $[a, b]$ .

In general, if  $g(x)$  is smooth, or can be made smooth by factoring out some  $W(x)$ , Gaussian algorithms produce higher accuracy than lower order ones, or conversely, the same accuracy with a fewer number of points. If the function being integrated is not smooth (for instance, if it contains noise), then using a higher order method such as Gaussian quadrature may well lead to lower accuracy. Sometimes the function may not be smooth because it has different behaviors in different regions. In these cases it makes sense to integrate each region separately and then add the answers together. In fact, some of the “smart” integration subroutines will decide for themselves how many intervals to use and what rule to use in each interval.

All the rules indicated in Tab. 5.2 are Gaussian with the general form (5.31). We can see that in one case the weighting function is an exponential, in another a Gaussian, and in several an integrable singularity. In contrast to the equally spaced rules, there is never an integration point at the extremes of the intervals, yet all of the points and weights change as the number of points  $N$  changes.

Although we will leave it to the references on numerical methods for the derivation of the Gauss points and weights, we note here that for ordinary Gaussian (Gauss–Legendre) integration, the points  $y_i$  turn out to be the  $N$  zeros of the Legendre polynomials, with the weights related to the derivatives,  $P_N(y_i) = 0$ , and  $w_i = 2 / ((1 - y_i^2)[P'_N(y_i)]^2)$ . Subroutines to generate these points and weights are standard in mathematical function libraries, are found in tables such as those in [7], or can be computed. The *gauss* subroutines we provide on the CD also scale the points to a specified region. As a check that your points are correct, you may want to compare them to the four-point set in Tab. 5.3.

**Tab. 5.3** Points and weights for four-point Gaussian quadrature

$\pm y_i$	$w_i$
0.33998 10435 84856	0.65214 51548 62546
0.86113 63115 94053	0.34785 48451 37454

## 5.6.1

**Mapping Integration Points**

Our standard convention (5.3) for the general interval  $[a, b]$  is

$$\int_a^b f(x)dx \approx \sum_{i=1}^N f(x_i)w_i \quad (5.32)$$

With Gaussian points and weights, the  $y$  interval  $-1 < y_i \leq 1$  must be *mapped* onto the  $x$  interval  $a \leq x \leq b$ . Here are some mappings we have found useful in our work. In all cases  $(y_i, w'_i)$  are the elementary Gaussian points and weights for the interval  $[-1, 1]$ , and we want to scale to  $x$  with various ranges:

1.  $[-1, 1] \rightarrow [a, b]$  uniformly,  $\frac{a+b}{2} = \text{midpoint}$ :

$$x_i = \frac{b+a}{2} + \frac{b-a}{2}y_i \quad w_i = \frac{b-a}{2}w'_i \quad (5.33)$$

$$\Rightarrow \int_a^b f(x)dx = \frac{b-a}{2} \int_{-1}^1 f[x(y)]dy \quad (5.34)$$

2.  $[0 \rightarrow \infty], a = \text{midpoint}$ :

$$x_i = a \frac{1+y_i}{1-y_i} \quad w_i = \frac{2a}{(1-y_i)^2}w'_i \quad (5.35)$$

3.  $[-\infty \rightarrow \infty], \text{scale set by } a$ :

$$x_i = a \frac{y_i}{1-y_i^2} \quad w_i = \frac{a(1+y_i^2)}{(1-y_i^2)^2}w'_i \quad (5.36)$$

4.  $[b \rightarrow \infty] a + 2b = \text{midpoint}$ :

$$x_i = \frac{a+2b+ay_i}{1-y_i} \quad w_i = \frac{2(b+a)}{(1-y_i)^2}w'_i \quad (5.37)$$

5.  $[0 \rightarrow b], ab/(b+a) = \text{midpoint}$ :

$$x_i = \frac{ab(1+y_i)}{b+a-(b-a)y_i} \quad w_i = \frac{2ab^2}{(b+a-(b-a)y_i)^2}w'_i \quad (5.38)$$

As you can see, even if your integration range extends out to infinity, there will be points at large but not infinite  $x$ . As you keep increasing the number of grid points  $N$ , the last  $x_i$  get larger but always remains finite.

## 5.6.2

**Gauss Implementation**

**Listing 5.2:** `IntegGauss.java` integrates the function  $f(x)$  via Gaussian quadrature with points and weights generated by the method `gauss`, which is the same for all applications. The parameter `eps` controls the level of precision desired and should be set by the user, as should the value for `job`, which controls the mapping of the Gauss points to  $(a,b)$ .

```
// IntegGauss.java: Gauss quadrature, need include gauss method

import java.io.*; // Location of PrintWriter

public class IntegGauss {
    static final double max_in = 1001; // Numb intervals
    static final double vmin = 0., vmax = 1.; // Int ranges
    static final double ME = 2.7182818284590452354E0; // Euler's const

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
        int i;
        double result;

        PrintWriter t = new PrintWriter(
            new FileOutputStream("IntegGauss.dat"), true);
        for (i=3; i <= max_in; i += 2) {
            result = gaussint(i, vmin, vmax);
            t.println("" + i + " " + Math.abs(result - 1 + 1/ME));
        }
        System.out.println("Output in IntegGauss.dat");
    }

    public static double f (double x) // f(x)
        {return (Math.exp(-x));}

    public static double gaussint (int no, double min, double max) {
        int n;
        double quadra = 0.;
        double w[] = new double[2001], x[] = new double[2001];
        Gauss.gauss (no, 0, min, max, x, w); // Returns pts & wts
        for (n=0; n < no; n++) {
            quadra += f(x[n])*w[n]; } // Calculate integral
        return (quadra);
    }
}
```

Write a double-precision program to integrate an arbitrary function numerically using the trapezoid rule, the Simpson rule, and Gaussian quadrature. For our **problem** we assume exponential decay so that there actually is an analytic answer:

$$\frac{dN(t)}{dt} = e^{-t} \Rightarrow N(1) = \int_0^1 e^{-t} dt = 1 - e^{-1} \quad (5.39)$$

In Listing 5.2 we give an example of a program that calls the `gauss` method. Note that the `gauss` method is contained in the `Gauss` class file, which is

a different class file than the one containing `IntGauss`. Consequently, as is Java's conventions, the `gauss` method is called as `Gauss.gauss`, where `Gauss` is the name of the class containing `gauss`. In Listing 5.3 we show the class file `Gauss.java` containing a method `gauss.java` that computes the Gaussian points and weights.

**Listing 5.3:** The program `IntegGauss.java` integrates the function  $f(x)$  via Gaussian quadrature. The points and weights are generated in the method `gauss`, which is the same for all applications. However, the parameter `eps`, which controls the level of precision desired, should be set by the user, as should the value for `job`, which controls the mapping of the Gauss points onto arbitrary intervals (they are generated for  $-1 \leq x \leq 1$ ).

```
// gauss.java Gaussian quadrature points and weights
public class Gauss{

    public static void gauss(int npts, int job, double a, double b,
                           double x[], double w[]) {
        int m = 0, i = 0, j = 0;
        double t = 0., t1 = 0., pp = 0., p1 = 0., p2 = 0., p3 = 0., xi;
        double eps = 3.E-14; // Accuracy: ADJUST!
        m = (npts + 1)/2;
        for (i=1; i <= m; i++) {
            t = Math.cos(Math.PI*((double)i-0.25)/((double)npts + 0.5));
            t1 = 1;
            while((Math.abs(t-t1)) >= eps) {
                p1 = 1.; p2 = 0.;
                for (j=1; j <= npts; j++) {
                    p3 = p2; p2 = p1;
                    p1=((2.*(double)j-1)*t*p2-((double)j-1)*p3)/((double)j);
                }
                pp = npts*(t*p1-p2)/(t*t-1.);
                t1 = t; t = t1 - p1/pp;
            }
            x[i-1] = -t; x[npts-i] = t;
            w[i-1] = 2./((1.-t*t)*pp*pp);
            w[npts-i] = w[i-1];
            System.out.println(" x[" + i-1] + " x[" + i-1] + " w " + w[npts-i]);
        }
        if (job==0) {
            for (i=0; i < npts; i++) {
                x[i] = x[i]*(b-a)/2. + (b + a)/2.;
                w[i] = w[i]*(b-a)/2.;
            }
        }
        if (job==1) {
            for (i=0; i < npts; i++) {
                xi=x[i];
                x[i] = a*b*(1. + xi) / (b + a-(b-a)*xi);
                w[i] = w[i]*2.*a*b*b/((b + a-(b-a)*xi)*(b+a-(b-a)*xi));
            }
        }
        if (job==2) {
            for (i=0; i < npts; i++) {
                xi=x[i];
                x[i] = a*b*(1. + xi) / (b + a-(b-a)*xi);
                w[i] = w[i]*2.*a*b*b/((b + a-(b-a)*xi)*(b+a-(b-a)*xi));
            }
        }
    }
}
```

```

        x[ i ] = (b*x[i] + b + a + a) / (1. - x[i]);
        w[ i ] = w[i]*2.* (a + b)/((1. - x[i])*(1. - x[i]));
    }
}
return;
}
}

```

## 5.7

### Empirical Error Estimate (Assessment)

Compute the relative error  $\epsilon = |(\text{numeric-exact})/\text{exact}|$  for the trapezoid rule, Simpson's rule, and Gaussian quadrature. You should observe that

$$\epsilon \simeq CN^\alpha \Rightarrow \log \epsilon = \alpha \log N + \text{constant} \quad (5.40)$$

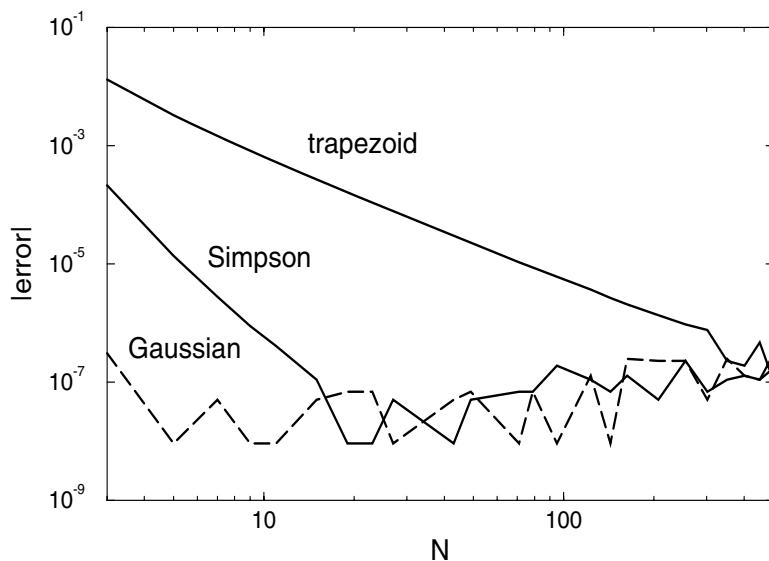
This means that a power-law dependence appears as a straight line on a log-log plot, and that if you use  $\log_{10}$ , then the ordinate on your log-log plot will be the negative of the number decimal places of precision in your calculation.

1. Present your data in the tabular form,

$N$	$\epsilon_T$	$\epsilon_S$	$\epsilon_G$
10	...	...	...

with spaces or tabs separating the fields. Try  $N$  values of 2, 10, 20, 40, 80, 160, .... (*Hint:* These are even numbers, which may not be the assumption of every rule.)

2. Make a plot of  $\log_{10} \epsilon$  versus  $\log_{10} N$  that gives the number of decimal places of precision obtained for each value of  $N$ . To illustrate, if  $\epsilon \simeq 10^{-7}$ , then  $\log_{10} \epsilon \simeq -7$ .
3. Use your plot or table to estimate the power-law dependence of the error  $\epsilon$  on the number of points  $N$  and to determine the number of decimal places of precision in your calculation. Do this for both the trapezoid and Simpson rules, and for both the algorithmic and the roundoff error regimes. (Note that it may be hard to reach the roundoff error regime for the trapezoid rule because the approximation error is so large.)



**Fig. 5.4** Log–log plot of the error in integration of exponential decay using the trapezoid rule, Simpson’s rule, and Gaussian quadrature, versus the number of integration points  $N$ . Approximately seven decimal places of precision are attainable with single precision (shown here) and 15 places with double precision.

## 5.8 Experimentation

Try two integrals for which the answers are less obvious:

$$F_1 = \int_0^{2\pi} \sin(100x)dx, \quad F_2 = \int_0^{2\pi} \sin^x(100x)dx \quad (5.41)$$

Explain why the computer may have trouble with these integrals.

## 5.9 Higher Order Rules (Algorithm)

As done with numerical differentiation, we can use the known functional dependence of the error on the interval size  $h$  to reduce the integration error. For simple rules such as trapezoid and Simpson’s, we have the analytic estimates (5.23), while for others you may have to experiment to determine the  $h$  dependence. To illustrate, if  $A(h)$  and  $A(h/2)$  are the values of the integral determined for the intervals  $h$  and  $h/2$ , respectively, we know that the integrals

have expansions with a leading error term proportional to  $h^2$ :

$$A(h) \approx \int_a^b f(x)dx + \alpha h^2 + \beta h^4 + \dots, \quad (5.42)$$

$$A\left(\frac{h}{2}\right) \approx \int_a^b f(x)dx + \frac{\alpha h^2}{4} + \frac{\beta h^4}{16} + \dots. \quad (5.43)$$

Consequently, we make the  $h^2$  term vanish by computing the combination

$$\frac{4}{3}A\left(\frac{h}{2}\right) - \frac{1}{3}A(h) \approx \int_a^b f(x)dx - \frac{\beta h^4}{4} + \dots \quad (5.44)$$

Clearly this particular trick (Romberg's extrapolation) works only if the  $h^2$  term dominates the error, and then only if the derivatives of the function are well behaved. An analogous extrapolation can also be made for other algorithms.

In Tab. 5.1 we gave the weights for several equal-interval rules. Whereas the Simpson's rule used two intervals, the 3/8 rule uses three, and the Milne<sup>2</sup> rule four. (These are single-interval rules and must be strung together to obtain a rule *extended* over the entire integration range. This means that the points that end one interval and begin the next get weighted twice.) You can easily determine the number of elementary intervals integrated over, and check whether you and we have written the weights right, by summing the weights for any rule. The sum is the integral of  $f(x) = 1$  and must equal  $h$  times the number of intervals (which, in turn, equals  $b - a$ ):

$$\sum_{i=1}^N w_i = h \times N_{\text{intervals}} = b - a \quad (5.45)$$

<sup>2</sup> There is, not coincidentally, a Milne Computer Center at Oregon State University, although there is no longer a central computer in it.

## 6

# Differentiation

### 6.1

#### **Problem 1: Numerical Limits**

A particle is moving through space, and you record its position as a function of time  $x(t)$  in a table. Your **problem** is to determine its velocity  $v(t) = dx/dt$  when all you have is this table of  $x$  versus  $t$ .

### 6.2

#### **Method: Numeric**

You probably did rather well in your first calculus course and feel competent at taking derivatives. However, you probably did not take derivatives of a table of numbers using the elementary definition:

$$\frac{df(x)}{dx} \stackrel{\text{def}}{=} \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (6.1)$$

In fact, even a computer runs into errors with this kind of limit because it is wrought with subtractive cancellation; the computer's finite word length causes the numerator to fluctuate between 0 and the machine precision  $\epsilon_m$  as the denominator approaches zero.

### 6.3

#### **Forward Difference (Algorithm)**

The most direct method for numerical differentiation of a function starts by expanding it in a Taylor series. This series advances the function one small step forward:

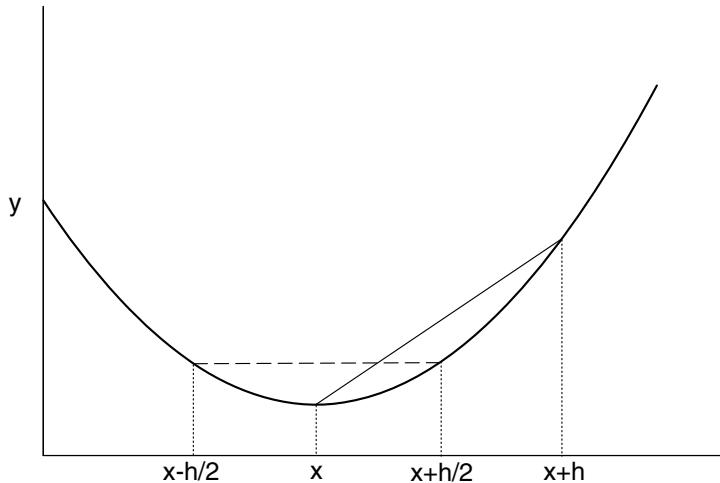
$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \dots \quad (6.2)$$

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5



**Fig. 6.1** Forward-difference approximation (solid line) and central-difference approximation (dashed line) for the numerical first derivative at point  $x$ . The central difference is seen to be more accurate.

where  $h$  is the step size (Fig. 6.1). We obtain the *forward-difference* derivative algorithm by solving (6.2) for  $f'(x)$ :

$$f'_{fd}(x) \stackrel{\text{def}}{=} \frac{f(x+h) - f(x)}{h}, \simeq f'(x) + \frac{h}{2}f''(x) + \dots \quad (6.3)$$

You can think of this approximation as using two points to represent the function by a straight line in the interval from  $x$  to  $x+h$ .

The approximation (6.3) has an error proportional to  $h$  (unless the heavens looks down kindly upon you and makes  $f''$  vanish). We can make the approximation error smaller and smaller by making  $h$  smaller and smaller. For too small an  $h$ , however, precision will be lost through the subtractive cancellation on the LHS of (6.3), and the decreased approximation error becomes irrelevant. As a case in point, let  $f(x) = a + bx^2$ . The exact derivative is  $f' = 2bx$ , while the computed derivative is

$$f'_{fd}(x) \approx \frac{f(x+h) - f(x)}{h} = 2bx + bh \quad (6.4)$$

This clearly becomes a good approximation only for small  $h$  ( $h \ll 2x$ ).

## 6.4

### Central Difference (Algorithm)

An improved approximation to the derivative starts with the basic definition (6.1). Rather than making a single step of  $h$  forward, we form a *central difference*

by stepping forward by  $h/2$  and backward by  $h/2$  (Fig. 6.1):

$$f'_{cd}(x) \stackrel{\text{def}}{=} \frac{f(x + h/2) - f(x - h/2)}{h} = D_{cd}f(x, h) \quad (6.5)$$

where we use the symbol  $D_{cd}$  for central difference. When the Taylor series for  $f(x \pm h/2)$  are substituted into (6.5), we obtain

$$f'_{cd}(x) \approx f'(x) + \frac{1}{24}h^2 f^{(3)}(x) + \dots \quad (6.6)$$

The important difference from (6.3) is that when  $f(x - h/2)$  is subtracted from  $f(x + h/2)$ , all terms containing an odd power of  $h$  in the Taylor series cancel. Therefore, the central-difference algorithm becomes accurate to one order higher in  $h$ , that is,  $h^2$ . If the function is well behaved, that is, if  $f^{(3)}h^2/24 \ll f^{(2)}h/2$ , then you can expect the error with the central-difference method to be smaller than with the forward difference (6.3).

If we now return to our polynomial example (6.4), we find that for this parabola, the central difference gives the exact answer independent of  $h$ :

$$f'_{cd}(x) \approx \frac{f(x + h/2) - f(x - h/2)}{h} = 2bx \quad (6.7)$$

## 6.5

### Extrapolated Difference (Method)

Because a differentiation rule based on keeping a certain number of terms in a Taylor series also provides an expression for the error (the terms not included), we can try to reduce the error by being clever. While the central difference (6.5) makes the error term proportional to  $h$  vanish, we can make the term proportional to  $h^2$  also vanish by algebraically *extrapolating* from relatively large  $h$ , and thus small roundoff error, to  $h \rightarrow 0$ :

$$f'_{ed}(x) \simeq \lim_{h \rightarrow 0} D_{cd}f(x, h) \quad (6.8)$$

We introduce the required, additional information by forming the central difference with step size  $h/2$ :

$$D_{ed}f(x, h/2) \stackrel{\text{def}}{=} \frac{f(x + h/4) - f(x - h/4)}{h/2} \quad (6.9)$$

$$\approx f'(x) + \frac{h^2 f^{(3)}(x)}{96} + \dots \quad (6.10)$$

We now eliminate the quadratic error term as well as the linear error term in (6.6) by forming the combination

$$f'_{ed}(x) \stackrel{\text{def}}{=} \frac{4D_{cd}f(x, h/2) - D_{cd}f(x, h)}{3} \quad (6.11)$$

$$\approx f'(x) - \frac{h^4 f^{(5)}(x)}{4 \times 16 \times 120} + \dots \quad (6.12)$$

If  $h = 0.4$  and  $f^{(5)} \simeq 1$ , then there will be only one place of the roundoff error and the truncation error is approximately machine precision  $\epsilon_m$ ; this is the best you can hope for.

A good way of computing (6.11) is to group the terms as

$$f'_{ed}(x) = \frac{1}{3h} \left\{ 8 \left[ f(x + \frac{h}{4}) - f(x - \frac{h}{4}) \right] - \left[ f(x + \frac{h}{2}) - f(x - \frac{h}{2}) \right] \right\} \quad (6.13)$$

The advantage to (6.13) is that it may reduce the loss of precision that occurs when large and small numbers are added together, only to be subtracted from other large numbers; it is better to first subtract the large numbers from each other and then add the difference to the small numbers.

When working with these and similar higher order methods, it is important to remember that while they may work as designed for well-behaved functions, they may fail badly for functions containing noise, as may result from computations or measurements. If noise is large, it may be better to first fit the data with some analytic function using the techniques of Chap. 8 and then differentiate the fit.

Regardless of the algorithm, the point to remember is that evaluating the derivative of  $f(x)$  at  $x$  requires you to know the values of  $f$  surrounding  $x$ . We shall use this same idea when we solve ordinary and partial differential equations.

## 6.6

### Error Analysis (Assessment)

The approximation errors in numerical differentiation decrease with decreasing step size  $h$ , while roundoff errors increase with decreasing step size (you have to take more steps and do more calculations). Recall from our discussion in Chap. 3 that the least overall approximation occurs for an  $h$  that makes the total error  $\epsilon_{\text{approx}} + \epsilon_{\text{ro}}$  a minimum, and that a rough guide this occurs when  $\epsilon_{\text{ro}} \approx \epsilon_{\text{approx}}$ .

Because differentiation subtracts two numbers close in value, we will assume that the roundoff error for differentiation is machine precision:

$$\begin{aligned} f' &\approx \frac{f(x+h) - f(x)}{h} \approx \frac{\epsilon_m}{h} \\ \Rightarrow \quad \epsilon_{\text{ro}} &\approx \frac{\epsilon_m}{h} \end{aligned} \quad (6.14)$$

The approximation error with the forward-difference algorithm (6.3) is  $\mathcal{O}(h)$ , while that with the central-difference algorithm (6.6) is  $\mathcal{O}(h^2)$ :

$$\epsilon_{\text{approx}}^{\text{fd}} \approx \frac{f^{(2)}h}{2}, \quad \epsilon_{\text{approx}}^{\text{cd}} \approx \frac{f^{(3)}h^2}{24} \quad (6.15)$$

Consequently, roundoff and approximation errors become equal when

$$\begin{aligned} \frac{\epsilon_m}{h} &\approx \epsilon_{\text{approx}}^{\text{fd}} = \frac{f^{(2)}h}{2} & \frac{\epsilon_m}{h} &\approx \epsilon_{\text{approx}}^{\text{cd}} = \frac{f^{(3)}h^2}{24} \\ \Rightarrow \quad h_{\text{fd}}^2 &= \frac{2\epsilon_m}{f^{(2)}} & \Rightarrow \quad h_{\text{cd}}^3 &= \frac{24\epsilon_m}{f^{(3)}} \end{aligned} \quad (6.16)$$

We take  $f' \approx f^{(2)} \approx f^{(3)}$  (which may be crude in general, though not bad for  $e^x$  or  $\cos x$ ), and assume double precision,  $\epsilon_m \approx 10^{-15}$ :

$$\begin{aligned} h_{\text{fd}} &\approx 4 \times 10^{-8} & h_{\text{cd}} &\approx 3 \times 10^{-5} \\ \Rightarrow \quad \epsilon_{\text{fd}} &\simeq \frac{\epsilon_m}{h_{\text{cd}}} \simeq 3 \times 10^{-8}, & \Rightarrow \quad \epsilon_{\text{cd}} &\simeq \frac{\epsilon_m}{h_{\text{cd}}} \simeq 3 \times 10^{-11} \end{aligned} \quad (6.17)$$

This may seem backward because the better algorithm leads to a larger  $h$  value. It is not. The ability to use a larger  $h$  means that the error in the central-difference method is some 1000 times smaller than the error in the forward-difference method here.

## 6.7

### Error Analysis (Implementation and Assessment)

1. Use forward-, central-, and extrapolated-difference algorithms to differentiate the functions  $\cos x$  and  $e^x$  at  $x = 0.1, 1.,$  and  $100.$ 
  - (a) Print out the derivative and its relative error  $\mathcal{E}$  as functions of  $h.$  Reduce the step size  $h$  until it equals machine precision  $h \approx \epsilon_m.$
  - (b) Plot  $\log_{10} |\mathcal{E}|$  versus  $\log_{10} h$  and check whether the number of decimal places obtained agrees with the estimates in the text.
  - (c) See if you can identify regions where truncation error dominate at large  $h$  and the roundoff error at small  $h$  in your plot. Do the slopes agree with our predictions?

**6.8****Second Derivatives (Problem 2)**

Let us say that you have measured the position  $x(t)$  versus time for a particle. Your **problem** is to determine the force on the particle. Newton's second law tells us that the force and acceleration are linearly related:

$$F = ma = m \frac{d^2x}{dt^2} \quad (6.18)$$

where  $F$  is the force,  $m$  is the particle's mass, and  $a$  is the acceleration. So if we can determine the acceleration  $a(t) = d^2x/dt^2$  from the  $x(t)$  values, we can determine the force.

The concerns we expressed about errors in first derivatives are even more valid for second derivatives where additional subtractions may lead to additional cancellations. Let us look again at the central-difference method:

$$f'(x) \simeq \frac{f(x + h/2) - f(x - h/2)}{h} \quad (6.19)$$

This algorithm gives the derivative at  $x$  by moving forward and backward from  $x$  by  $h/2$ . We take the second derivative  $f^{(2)}(x)$  to be the central difference of the first derivative:

$$f^{(2)}(x) \simeq \frac{f'(x + \frac{h}{2}) - f'(x - \frac{h}{2})}{h}, \quad (6.20)$$

$$\simeq \frac{[f(x + h) - f(x)] - [f(x) - f(x - h)]}{h^2} \quad (6.21)$$

$$\simeq \frac{f(x + h) + f(x - h) - 2f(x)}{h^2} \quad (6.22)$$

As was true for first derivatives, we can determine the second derivative at  $x$  by evaluating the function in the region surrounding  $x$ . Although the form (6.22) is more compact and requires fewer steps than (6.21), it may increase subtractive cancellation by first storing the "large" number  $f(x + h) + f(x - h)$ , and then subtracting another large number  $2f(x)$  from it. We ask you to explore this difference as an exercise.

**6.8.1****Second Derivative Assessment**

Write a program to calculate the second derivative of  $\cos x$  using the central-difference algorithms (6.21) and (6.22). Test it over four cycles. Start with  $h \approx \pi/10$  and keep reducing  $h$  until you reach machine precision.

## 7

**Trial and Error Searching**

Many computer techniques, such as multiplying matrices, follow a well-defined set procedure. The same basic operation is repeated for each element in the matrix, and stops when we have covered all the elements. In contrast, some computational techniques are trial-and-error algorithms in which the program goes through some numerical procedure, and quits only when the error is acceptably small. (We already did some of this when we summed a power series until the terms became small.) This type of programming is usually interesting because we must think hard in order to foresee how to have the computer act intelligently for all possible situations. These “trial and error” programs are also interesting to run because, like experiments, they depend very much on the initial conditions, and it is hard to predict exactly what the computer will come up as it searches for a solution.

## 7.1

**Quantum States in Square Well (Problem IA)**

Maybe the most standard problem in quantum mechanics is to solve for the energies of a particle of mass  $m$  bound within a 1D square well of radius  $a$ :<sup>1</sup>

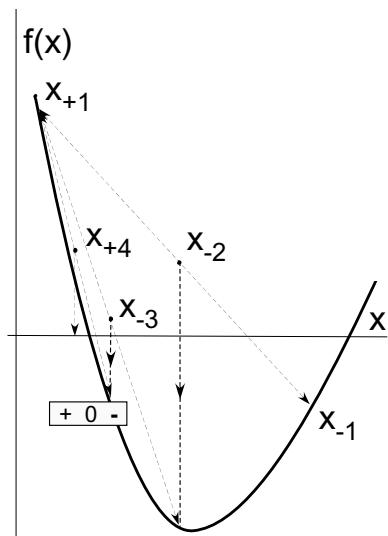
$$V(x) = \begin{cases} -V_0 & \text{for } |x| \leq a \\ 0 & \text{for } |x| \geq a \end{cases} \quad (7.1)$$

As shown in quantum mechanics texts [8], the energies of the bound states  $E = -E_B < 0$  within this well are solutions of the transcendental equations

$$\sqrt{2m(V_0 - E_B)} \tan\left[a\sqrt{2m(V_0 - E_B)}\right] = \sqrt{2mE_B} \quad (\text{even}), \quad (7.2)$$

$$\sqrt{2m(V_0 - E_B)} \cotan\left[a\sqrt{2m(V_0 - E_B)}\right] = \sqrt{2mE_B} \quad (\text{odd}) \quad (7.3)$$

<sup>1</sup> We solve this same problem in Chap. 16 using an approach that is applicable to most any potential, and which also provides the wave functions. The approach of this section only works for the eigenenergies of a square well.



**Fig. 7.1** A graphical representation of the steps involved in solving for a zero of  $f(x)$  using the bisection algorithm. The bisection algorithm takes the midpoint of the interval as the new guess for  $x$ , and so each step reduces the interval size by one half. Four steps are shown.

where even and odd refer to the symmetry of the wave function, and where we have chosen units such that  $\hbar = 1$ . To have a numerical problem to solve, we set  $2m = 1$ ,  $a = 1$ , and  $V_0 = 10$ , in which case there results

$$\sqrt{10 - E_B} \tan\left(\sqrt{10 - E_B}\right) = \sqrt{E_B} \quad (\text{even}) \quad (7.4)$$

$$\sqrt{10 - E_B} \cotan\left(\sqrt{10 - E_B}\right) = \sqrt{E_B} \quad (\text{odd}) \quad (7.5)$$

Your **problem** is to

1. Find several bound state energies  $E_B$  for even wave functions, that is, solution of (7.4).
2. Notice that the “10” in these equations is proportional to the strength of the potential that causes the binding. See if making the potential deeper, say by changing the 10 to a 20 or a 30, produces a larger number of, or deeper bound states.
3. Although we have yet to discuss the methods of solution, after we have, we want you to compare a solution found with the Newton–Raphson algorithms to one found with the bisection algorithm.

## 7.2

### Trial-and-Error Root Finding via Bisection Algorithm

The trial-and-error technique for root finding looks for a solution  $x$  of the equation  $f(x) = 0$ . Having the RHS = 0 is just a convention. Any equation such as  $10 \sin x = 3x^3$  can easily be converted to  $10 \sin x - 3x^3 = 0$ . The general procedure is the one in which we start with a guess value for  $x$ , substitute that guess into  $f(x)$  (the “trial”), and then see how far the RHS is from zero (the “error”). You then change  $x$  appropriately (a new guess) and try it out in  $f(x)$ . The procedure continues until  $f(x) \simeq 0$  to some desired level of precision, or until the changes in  $x$  are insignificant. As a safeguard, you will also want to set a maximum value for the number of trials.

The most elementary trial-and-error technique is the *bisection algorithm*. It is reliable, but slow. If you know some interval in which  $f(x)$  changes sign, then the bisection algorithm will always converge to the root by finding progressively smaller and smaller intervals in which the zero occurs. Other techniques, such as the Newton–Raphson method we describe next, may converge more quickly, but if the initial guess is not good, it may become unstable and fail completely.

The basis behind the bisection algorithm is shown in Fig. 7.1. We start with two values of  $x$ , between which we know a zero occurs. (You can determine these by making a graph or by stepping through different  $x$  values and looking for a sign change.) To be specific, let us say that  $f(x)$  is negative at  $x_-$  and positive at  $x_+$ :

$$f(x_-) < 0 \quad f(x_+) > 0 \quad (7.6)$$

Note that it may well be that  $x_- > x_+$  if the function changes from positive to negative as  $x$  increases. Thus we start with the interval  $x_+ \leq x \leq x_-$  within which we know a zero occurs. The algorithm then bisects this interval at

$$x = \frac{x_+ + x_-}{2} \quad (7.7)$$

and selects as its new interval the half in which the sign change occurs:

```
if ( f(x) * f(xPlus) > 0 ) xPlus = x
else xMinus = x
```

The process continues until the value of  $f(x)$  is less than a predefined level of precision, or until a predefined (large) number of subdivisions has occurred.

The example in Fig. 7.1 shows the first interval extending from  $x_- = x_{+1}$  to  $x_+ = x_{-1}$ . We then bisect that interval at  $x$ , and since  $f(x) < 0$  at the midpoint, we set  $x_- \equiv x_{-2} = x$  and label it as  $x_{-2}$  to indicate the second step. We then use  $x_{+2} \equiv x_{+1}$  and  $x_{-2}$  as the next interval and continue the process. We see that only  $x_-$  changes for the first three steps in this example, but that

for the fourth step,  $x_+$  finally changes. The changes then get too small for us to show.

### 7.2.1

#### Bisection Algorithm Implementation

1. The first step in implementing any search algorithm is to get an idea what your function looks like. For the present problem you do this by making a plot of  $f(E) = \sqrt{10 - E_B} \tan(\sqrt{10 - E_B}) - \sqrt{E_B}$  versus  $E_B$ . Note from your plot some approximate values at which  $f(E_B) = 0$ . Your program should be able to find more exact values for these zeros.
2. Write a program that implements the bisection algorithm and use it to find some solutions of (7.4).
3. *Warning:* because the tan function has singularities, you have to be careful. In fact, your graphics program (or Maple) may not function accurately near these singularities. One cure is to use a different, but equivalent form of the equation. Show that an equivalent form of (7.4) is

$$\sqrt{E} \cot(\sqrt{10 - E}) - \sqrt{10 - E} = 0 \quad (7.8)$$

4. Make a second plot of (7.8), which also has singularities, but at different places. Choose some approximate locations for zeros from this plot.
5. Compare the roots you find with those given by *Maple* or *Mathematica*.

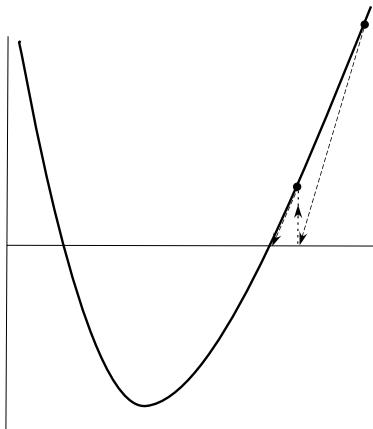
### 7.3

#### Newton–Raphson (Faster Algorithm)

The Newton–Raphson algorithm finds approximate roots of the equation

$$f(x) = 0 \quad (7.9)$$

more quickly than the bisection method. As we see graphically in Fig. 7.2, this algorithm is the equivalent of drawing a straight line  $f(x) \simeq mx + b$  tangent to the curve at an  $x$  value for which  $f(x) \simeq 0$ , and then using the intercept  $x = -b/m$  of that line with the  $x$  axis as an improved guess for the root. If the “curve” were a straight line, the answer would be exact; otherwise it is a good approximation if the guess is close enough to the root for  $f(x)$  to be nearly linear. The process continues until some set level of precision is reached. If a guess is in a region where  $f(x)$  is nearly linear (Fig. 7.2), then the convergence is much more rapid than the bisection algorithm.



**Fig. 7.2** A graphical representation of the steps involved in solving for a zero of  $f(x)$  using the Newton–Raphson method. The Newton–Raphson method takes the new guess as the zero of the line tangent to  $f(x)$  at the old guess. Because of the rapid convergence, only two steps can be shown.

The analytic formulation of the Newton–Raphson algorithm starts with an old guess  $x_0$ , and expresses the new guess  $x$  as a correction  $\Delta x$  to the old guess:

$$x_0 = \text{old guess} \quad \Delta x = \text{unknown correction} \quad (7.10)$$

$$\Rightarrow \quad x = x_0 + \Delta x = (\text{unknown}) \text{ new guess}. \quad (7.11)$$

We next expand the known function  $f(x)$  in a Taylor series around  $x_0$  and keep only the linear terms:

$$f(x = x_0 + \Delta x) \simeq f(x_0) + \left. \frac{df}{dx} \right|_{x_0} \Delta x \quad (7.12)$$

We then determine the correction  $\Delta x$  by determining the point at which this linear approximation to  $f(x)$  crosses the  $x$  axis:

$$f(x_0) + \left. \frac{df}{dx} \right|_{x_0} \Delta x = 0 \quad \Rightarrow \quad \Delta x = -\frac{f(x_0)}{\left. df/dx \right|_{x_0}} \quad (7.13)$$

The procedure is repeated starting at the improved  $x$  until some set level of precision is obtained.

The Newton–Raphson algorithm (7.13) requires evaluation of the derivative  $df/dx$  at each value of  $x_0$ . In many cases you may have an analytic expression for the derivative and can built that into the algorithm. However, and especially for more complicated problems, it is simpler and less error prone to use a numerical, forward-difference approximation to the derivative:<sup>2</sup>

$$\frac{df}{dx} \simeq \frac{f(x + \delta x) - f(x)}{\delta x} \quad (7.14)$$

<sup>2</sup> We discuss numerical differentiation in Chap. 6.

where  $\delta x$  is some small change in  $x$  that you just chose (different from the  $\Delta x$  in (7.13)), the exact value not mattering once a solution is found. While a central-difference approximation for the derivative would be more accurate, it would require additional evaluations of the  $f$ 's, and once you find a zero it does not matter how you got there. On the CD we give the programs `Newton_cd.java` (also Listing 7.1) and `Newton_fd.java`, which implement the derivative both ways.

**Listing 7.1:** The program `Newton_cd.java` uses the Newton–Raphson method to search for a zero of the function  $f(x)$ . A central-difference approximation is used to determine  $df/dx$ .

```
// Newton_cd.java: Newton–Raphson root finder , central diff deriv

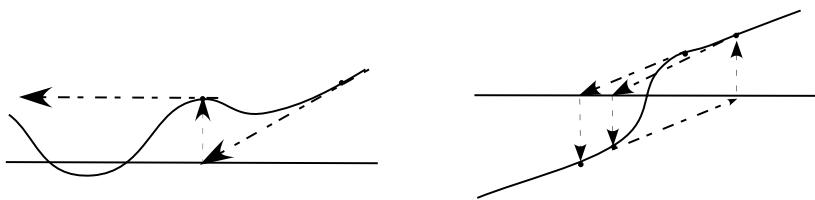
public class Newton_cd {
    public static double f(double x)          // Find zero of this function
    { return 2*Math.cos(x) - x; }

    public static void main(String[] argv) {
        double x = 2., dx = 1e-2, F= f(x), eps = 1e-6, df;
        int it, imax = 100;                      // Max no of iterations permitted
                                                // Iterate
        for ( it = 0; it <= imax; it++ ) {
            System.out.println("Iteration # = "+it+" x = "+x+" f(x) = "+F);
                                                // Central diff deriv
            df = ( f(x + dx/2) - f(x-dx/2) )/dx;
            dx = -F/df;
            x += dx;                                // New guess
            F = f(x);                                // Save for use
                                                // Check for convergence
            if ( Math.abs(F) <= eps ) {
                System.out.println("Root found, tolerance eps = " + eps);
                break;
            }
        }
    }
}
```

### 7.3.1

#### Newton–Raphson with Backtracking

Two examples of possible problems with Newton-Raphson are shown in Fig. 7.3. On the left we see a case where the search takes us to an  $x$  value where the function has a local minimum or maximum, that is, where  $df/dx = 0$ . Because  $\Delta x = -f/f'$ , this leads to a horizontal tangent (division by zero), and so the next guess is  $x = \infty$ , from where it is hard to return. When this happens, you need to start your search off with a different guess and pray that you do not fall into this trap again. In those cases where the correction is very large, but maybe not infinite, you may want to try backtracking (described below) and hope that by taking a smaller step you will not get into as much trouble.



**Fig. 7.3** Two examples of how Newton–Raphson may fail if the initial guess is not in the region where  $f(x)$  can be approximated by a straight line. *Left:* A guess lands at a local minimum/maxima, that is, a place where the derivative vanishes, and so the next guess ends up at  $x = \infty$ . *Right:* The search has fallen into an infinite loop.

On the right of Fig. 7.3 we see a case where a search falls into an infinite loop surrounding the zero, without ever getting there. A solution to this problem is called *backtracking*. As the name implies, it says that in those cases where the new guess  $x_0 + \Delta x$  leads to an increase in the magnitude of the function,  $|f(x_0 + \Delta x)|^2 > |f(x_0)|^2$ , you should backtrack somewhat and try a smaller guess, say  $x_0 + \Delta x/2$ . If the magnitude of  $f$  still increases, then you just need to backtrack some more, say by trying  $x_0 + \Delta x/4$  as your next guess, and so forth. Because you know that the tangent line leads to a local decrease in  $|f|$ , eventually an acceptable small enough step should be found.

The problem in both these cases is that the initial guesses were not close enough to the regions where  $f(x)$  is approximately linear. So again, a good plot may help produce a good first guess. Alternatively, you may want to start off your search with the bisection algorithm, and then switch to the faster Newton-Raphson when you get closer to the zero.

### 7.3.2

#### Newton–Raphson Implementation

1. Use the Newton–Raphson algorithm to find some energies  $E_B$  that are solutions of (7.4). Compare this solution with the one found with the bisection algorithm.
2. Again, notice that the “10” in this equation is proportional to the strength of the potential that causes the binding. See if making the potential deeper, say by changing the 10 to a 20 or a 30, produces more or deeper bound states. (Note that in contrast to the bisection algorithm, your initial guess must be close to the answer for Newton–Raphson to work.)
3. Modify your algorithm to include backtracking and then try it out on some problem cases.

**8****Matrix Computing and N-D Newton Raphson**

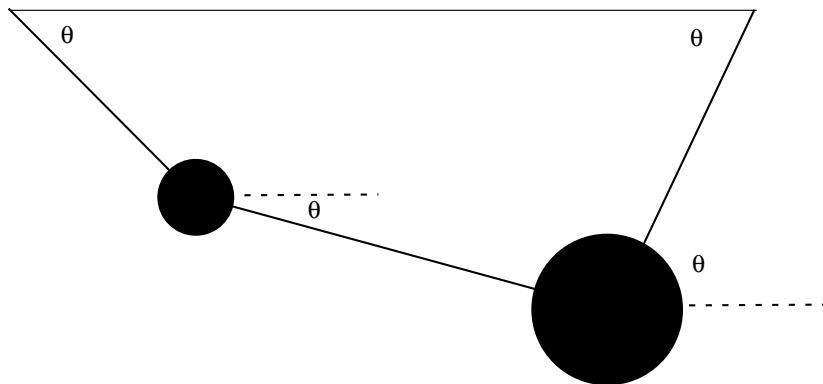
Physical systems are often modeled by systems of simultaneous equations. As the models are made more realistic, the matrices may become very large, and what with matrix manipulations intrinsically involving the continued repetition of a small number of simple instructions, computer become an excellent tool for these models. Because the steps are so predictable and straightforward, the codes can be made to run extraordinarily quickly by using clever techniques and by *tuning* them to a particular machine's architecture (see Chap. 13 for a discussion of computer architectures).

For the above reasons, it has become increasingly important for the computational scientist to use industrial-strength matrix subroutines from well-established scientific libraries. These subroutines are usually an order of magnitude or more faster than the elementary methods found in linear algebra texts<sup>1</sup>, are usually designed to minimize the roundoff error, and are often "robust," that is, have a high chance of being successful for a broad class of problems. For these reasons we recommend that you *do not write your own matrix subroutines*, but, instead, get them from a library. An additional value of the library routine is that you can run the same program on a workstation and a supercomputer and automatically have the numerically most intensive parts of it adapted to the RISC, parallel, or vector architecture of the individual computer.

The thoughtful and pensive reader may be wondering when a matrix is "large" enough to be worth the effort of using a library routine. Basically, if the summed sizes of all your matrices are a good fraction of your computer's RAM, if virtual memory is needed to run your program, or if you have to wait minutes or hours for your job to finish, then it is a good bet that your matrices are "large."

Now that you have heard the sales pitch, you may be asking, "What's the cost?" In the later part of this chapter we pay the costs of having to find what libraries are available, of having to find the name of the routine in that library,

<sup>1</sup> Although we prize the book [9] and what it has accomplished, we cannot recommend taking subroutines from it. They are neither optimized nor documented for easy and stand-alone use. The subroutine libraries recommended in this chapter are both.



**Fig. 8.1** Two masses connected by three pieces of string, and suspended from a horizontal bar of length  $L$ . The angles and the tensions in the strings are unknown.

of having to find the names of the subroutines your routine calls, and then of having to figure out how to call all these routines. And because many of the library routines are in Fortran, if you are a C programmer you will also be taxed by having to call a Fortran routine from your C program!

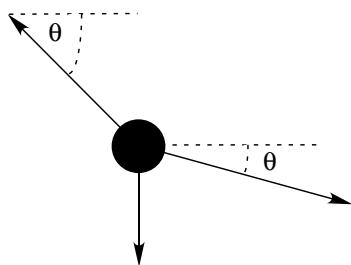
### 8.1

#### Two Masses on a String (Problem II)

Two masses with weights  $(W_1, W_2) = (10, 20)$  N are hung from two pieces of string and a horizontal bar of lengths  $(L, L_1, L_2, L_3,) = (8, 3, 4, 4)$  cm (Fig. 8.1). The **problem** is to find the angles made by the strings and the tensions exerted by the strings.

In spite of the fact that this is a simple problem requiring no more than first-year physics to formulate, the coupled transcendental equations that result cannot be solved by hand, and a computer is needed. However, even the computer cannot solve this directly, but rather must solve it by trial and error. For larger problems, like bridges, only a computed solution is possible.

In the sections to follow we solve the two-mass problem. Your **problem** is to test this solution for a variety of weights and lengths, and, for the more adventurous, to try extending it to the three-mass problem. In either case, check the physical reasonableness of your solution; the deduced tensions should be positive and of similar magnitude to the weights of the spheres, and the deduced angles should correspond to a physically realizable geometry, as confirmed with a sketch. Some of the **Exploration** you should do is see at what point your initial guess gets so bad that the computer is unable to find a physical solution.



**Fig. 8.2** A free body diagram for one weight in equilibrium.

### 8.1.1

#### Statics (Theory)

This physics problem is easy to convert to equations using the laws of statics, yet inhumanely painful in yielding an analytic solution. We start by writing down the geometric constraints that the horizontal length of the structure is  $L$ , and that the strings begin and end at the same height:

$$L_1 \cos \theta_1 + L_2 \cos \theta_2 + L_3 \cos \theta_3 = L \quad (8.1)$$

$$L_1 \sin \theta_1 + L_2 \sin \theta_2 - L_3 \sin \theta_3 = 0 \quad (8.2)$$

$$\sin^2 \theta_1 + \cos^2 \theta_1 = 1 \quad (8.3)$$

$$\sin^2 \theta_2 + \cos^2 \theta_2 = 1 \quad (8.4)$$

$$\sin^2 \theta_3 + \cos^2 \theta_3 = 1 \quad (8.5)$$

Observe that the last three equations include trigonometric identities as independent equations because we are treating  $\sin \theta$  and  $\cos \theta$  as independent variables; this makes the search procedure easier to implement. The basics physics says that since there are no accelerations, the sum of the forces in the horizontal and vertical directions must equal zero:

$$T_1 \sin \theta_1 - T_2 \sin \theta_2 - W_1 = 0 \quad (8.6)$$

$$T_1 \cos \theta_1 - T_2 \cos \theta_2 = 0 \quad (8.7)$$

$$T_2 \sin \theta_2 + T_3 \sin \theta_3 - W_2 = 0 \quad (8.8)$$

$$T_2 \cos \theta_2 - T_3 \cos \theta_3 = 0 \quad (8.9)$$

Here  $W_i$  is the weight of mass  $i$ ,  $T_i$  is the tension in string  $i$ , and the geometry is given in Fig. 8.2. Note that since we do not have a rigid structure, we cannot assume that the torques are in equilibrium.

## 8.1.2

**Multidimensional Newton–Raphson Searching**

Equations (8.1)–(8.9) are nine simultaneous nonlinear equations. While linear equations can be solved directly, nonlinear equations cannot [10]. You can use the computer to search for a solution, but there is no guarantee that it will find one. However, if you use your physical intuition to come up with a good initial guess for the solution, the computer is more likely to succeed.

We apply the Newton–Raphson algorithm to solve this set of equations, much like we did for a single equation. We start by renaming the nine unknown angles and tensions as the subscripted variable  $y_i$ , and placing the variable together as a vector:

$$\mathbf{y} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} \sin \theta_1 \\ \sin \theta_2 \\ \sin \theta_3 \\ \cos \theta_1 \\ \cos \theta_2 \\ \cos \theta_3 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} \quad (8.10)$$

The nine equations to be solved are written in a general form with zeros on the right-hand sides, and also placed in vector form:

$$f_i(x_1, x_2, \dots, x_N) = 0 \quad i = 1, N \quad (8.11)$$

$$\mathbf{f}(\mathbf{y}) = \begin{pmatrix} f_1(\mathbf{y}) \\ f_2(\mathbf{y}) \\ f_3(\mathbf{y}) \\ f_4(\mathbf{y}) \\ f_5(\mathbf{y}) \\ f_6(\mathbf{y}) \\ f_7(\mathbf{y}) \\ f_8(\mathbf{y}) \\ f_9(\mathbf{y}) \end{pmatrix} = \begin{pmatrix} 3x_4 + 4x_5 + 4x_6 - 8 \\ 3x_1 + 4x_2 - 4x_3 \\ x_7x_1 - x_8x_2 - 10 \\ x_7x_4 - x_8x_5 \\ x_8x_2 + x_9x_3 - 20 \\ x_8x_5 - x_9x_6 \\ x_1^2 + x_4^2 - 1 \\ x_2^2 + x_5^2 - 1 \\ x_3^2 + x_6^2 - 1 \end{pmatrix} = \mathbf{0} \quad (8.12)$$

The solution to these equations is quite a feat, a set of nine  $x_i$  values which make all nine  $f_i$ 's vanish simultaneously. Although these equations are not very complicated (the physics after all is elementary), the terms quadratic in  $x$  make them nonlinear and this makes it hard or impossible to find an analytic solution. In fact, even the numerical solution is by trial and error, that is, guess work. We guess a solution, expand the nonlinear equations into linear form,

solve the linear equations, and hope that this brings us closer to the nonlinear solution. Explicitly, let the approximate solution at any one stage be the set  $\{x_i\}$ . We then assume that there is a set of corrections  $\{\Delta x_i\}$  for which

$$f_i(x_1 + \Delta x_1, x_2 + \Delta x_2, \dots, x_9 + \Delta x_9) = 0 \quad i = 1, 9 \quad (8.13)$$

We solve for the approximate values of the  $\Delta x_i$ 's by assuming that our previous solution is close enough to the actual solution for two terms in the Taylor series to be accurate:

$$f_i(x_1 + \Delta x_1, \dots, x_9 + \Delta x_9) \simeq f_i(x_1, \dots, x_9) + \sum_{j=1}^9 \frac{\partial f_i}{\partial x_j} \Delta x_j = 0, \quad i = 1, 9 \quad (8.14)$$

We thus have a solvable set of nine linear equations in the nine unknowns  $\Delta x_i$ . To make this clearer, we write them out as nine explicit equations and then as a single matrix equation:

$$\begin{aligned} f_1 + \frac{\partial f_1}{\partial x_1} \Delta x_1 + \frac{\partial f_1}{\partial x_2} \Delta x_2 + \dots + \frac{\partial f_1}{\partial x_9} \Delta x_9 &= 0 \\ f_2 + \frac{\partial f_2}{\partial x_1} \Delta x_1 + \frac{\partial f_2}{\partial x_2} \Delta x_2 + \dots + \frac{\partial f_2}{\partial x_9} \Delta x_9 &= 0 \\ &\vdots \\ f_9 + \frac{\partial f_9}{\partial x_1} \Delta x_1 + \frac{\partial f_9}{\partial x_2} \Delta x_2 + \dots + \frac{\partial f_9}{\partial x_9} \Delta x_9 &= 0 \end{aligned}$$

i.e. 
$$\begin{pmatrix} f_1 \\ f_2 \\ \ddots \\ f_9 \end{pmatrix} + \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_9} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_9} \\ \ddots & \ddots & \ddots & \ddots \\ \frac{\partial f_9}{\partial x_1} & \frac{\partial f_9}{\partial x_2} & \dots & \frac{\partial f_9}{\partial x_9} \end{pmatrix} \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \ddots \\ \Delta x_9 \end{pmatrix} = 0 \quad (8.15)$$

Note now that the derivatives and the  $f$ 's are all evaluated at known values of the  $x_i$ 's, so only the vector of  $\Delta x_i$  values is unknown.

We write this equation in matrix notation as

$$\mathbf{f} + \mathbf{F}' \Delta \mathbf{x} = 0 \quad \Rightarrow \quad \mathbf{F}' \Delta \mathbf{x} = -\mathbf{f} \quad (8.16)$$

$$\Delta \mathbf{x} = \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \ddots \\ \Delta x_9 \end{pmatrix} \quad \mathbf{f} = \begin{pmatrix} f_1 \\ f_2 \\ \ddots \\ f_9 \end{pmatrix} \quad \mathbf{F}' = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_9} \\ \frac{\partial f_2}{\partial x_1} & \dots & \frac{\partial f_2}{\partial x_9} \\ \ddots & \ddots & \ddots \\ \frac{\partial f_9}{\partial x_1} & \dots & \frac{\partial f_9}{\partial x_9} \end{pmatrix}$$

Here we use **bold** to emphasize the vector nature of the columns of  $f_i$  and  $\Delta x_i$  values, and call the matrix of the derivatives  $\mathbf{F}'$  (it is also called  $\mathbf{J}$  sometimes because it is the *Jacobian* matrix)

The equation  $\mathbf{F}'\Delta\mathbf{x} = -\mathbf{f}$  is in the standard form for the solution of a linear equation (often written as  $\mathbf{Ax} = \mathbf{b}$ ), where  $\Delta\mathbf{x}$  is the vector of unknowns and  $\mathbf{b} = -\mathbf{f}$ . Matrix equations are solved using the techniques of linear algebra, and in the sections to follow we shall show how to do that on a computer. In a formal (and sometimes practical) sense, the solution of (8.16) is obtained by multiplying both sides of the equation by the inverse of the  $\mathbf{F}'$  matrix:

$$\Delta\mathbf{x} = -\mathbf{F}'^{-1}\mathbf{f} \quad (8.17)$$

where the inverse must exist if there is to be a unique solution. Although we are dealing with matrices now, this solution is identical in form to that of the 1D problem,  $\Delta x = -(1/f')f$ . In fact, one of the reasons we use formal or abstract notations for matrices is to reveal the simplicity that lies within.

Seeing that we have gone through a good number of steps to reach this point, let us summarize. Each time we solve for the corrections  $\Delta x_i$  we use them to improve the existing value of the guess  $x_i$ . This is our improved guess. We then repeat the entire procedure to obtain corrections to our improved guess, which leads to an even more improved guess. The technique is repeated until the improvements are smaller than a predefined tolerance limit, or until too many iterations are made. If the initial guess is close to a solution to the nonlinear equations, then convergence is usually found after just a few iterations; if the initial guess is not close, the technique may well fail.

As we indicated for the single-equation Newton–Raphson method, while for our two-mass problem we can derive analytic expressions for the derivatives  $\partial f_i / \partial x_j$ , there are  $9 \times 9 = 81$  such derivatives for this (small) problem, and entering them all is both time consuming and error prone. In contrast, and especially for more complicated problems, it is straightforward to program up a forward-difference approximation for the derivatives,

$$\frac{\partial f_i}{\partial x_j} \simeq \frac{f_i(x_j + \Delta x_j) - f_i(x_j)}{\Delta x_j} \quad (8.18)$$

where each individual  $x_j$  is varied independently since these are partial derivatives, and  $\Delta x_j$  are some arbitrary changes you input. While a central-difference approximation for the derivative would be more accurate, it would also require more evaluations of the  $f$ 's, and once we find a solution it does not matter how accurate our algorithm for the derivative was.

As we have already discussed for the 1D Newton–Raphson method, the method can fail if the initial guess is not close enough to the zero of  $f$  (here all  $N$  of them) for the  $f$ 's to be approximated as linear. The *backtracking* technique may be applied here as well, in the present case, progressively decreasing the corrections  $\Delta x_i$  until  $|f|^2 = |f_1|^2 + |f_2|^2 + \dots + |f_N|^2$  decreases.

## 8.2

### Classes of Matrix Problems (Math)

It helps to remember that the rules of mathematics apply even to the world's most powerful computers. To illustrate, you *should* have problems solving equations if you have more unknowns than equations, or if your equations are not linearly independent. But do not fret, while you cannot obtain a unique solution when there are not enough equations, you may still be able to map out a space of allowable solutions. At the other extreme, if you have more equations than unknowns, you have an *overdetermined* problem, which may not have a unique solution. This overdetermined problem is sometimes treated like data fitting in which a solution to a sufficient set of equations is found, tested on the unused equations, and then improved if needed. Not surprisingly, this latter technique is known as the *linear least-squares method* because it finds the best solution "on the average."

The most basic matrix problem is the system of linear equations you have to solve for the two-mass **problem**:

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{A}_{N \times N} \mathbf{x}_{N \times 1} = \mathbf{b}_{N \times 1} \quad (8.19)$$

where  $\mathbf{A}$  is a known  $N \times N$  matrix,  $\mathbf{x}$  is an unknown vector of length  $N$ , and  $\mathbf{b}$  is a known vector of length  $N$ . The best way to solve this equation is by Gaussian elimination or LU (lower–upper) decomposition. They yields the vector  $\mathbf{x}$  without explicitly calculating  $\mathbf{A}^{-1}$ . Another, albeit slower and less robust, method is to determine the inverse of  $\mathbf{A}$ , and then form the solution by multiplying both sides of (8.19) by  $\mathbf{A}^{-1}$ :

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \quad (8.20)$$

If you have to solve the matrix equation,

$$\mathbf{Ax} = \lambda\mathbf{x} \quad (8.21)$$

with  $\mathbf{x}$  an unknown vector and  $\lambda$  an unknown parameter, then the direct solution (8.20) will not be of much help because the matrix  $\mathbf{b} = \lambda\mathbf{x}$  contains the unknowns  $\lambda$  and  $\mathbf{x}$ . Equation (8.21) is *the eigenvalue problem*. It is harder to solve than (8.19) because solutions exist only for certain  $\lambda$  values (or possibly none depending on  $\mathbf{A}$ ). We use the identity matrix to rewrite (8.21) as

$$[\mathbf{A} - \lambda\mathbf{I}]\mathbf{x} = 0 \quad (8.22)$$

we see that multiplication by  $[\mathbf{A} - \lambda\mathbf{I}]^{-1}$  yields the *trivial solution*:

$$\mathbf{x} = 0 \quad (\text{trivial solution}) \quad (8.23)$$

While the trivial solution is a bona fide solution, it is trivial. For a more interesting solution to exist, there must be something that forbids us from multiplying both sides of (8.22) by  $[\mathbf{A} - \lambda\mathbf{I}]^{-1}$ . That something is the nonexistence

of the inverse. If you recall that Cramer's rule for the inverse requires division by  $\det[\mathbf{A} - \lambda \mathbf{I}]$ , it becomes clear that the inverse fails to exist (and in this way eigenvalues *do* exist) when

$$\det[\mathbf{A} - \lambda \mathbf{I}] = 0 \quad (8.24)$$

Those values of  $\lambda$  that satisfy the *secular equation* (8.24) are the eigenvalues of the original equation (8.21).

If you were interested only in the eigenvalues, you would have the computer to solve (8.24). To do that, you need a subroutine to calculate the determinant of a matrix, and then a search routine to zero in the solution of (8.24). Such routines are available in the libraries. The traditional way to solve the eigenvalue problem (8.21) for both eigenvalues and eigenvectors is by *diagonalization*. This is equivalent to successive changes of basis vectors, each change leaving the eigenvalues unchanged while continually decreasing the values of the off-diagonal elements of  $\mathbf{A}$ . The sequence of transformations is equivalent to continually operating on the original equation with a matrix  $\mathbf{U}$ :

$$\mathbf{U}\mathbf{A}(\mathbf{U}^{-1}\mathbf{U})\mathbf{x} = \lambda \mathbf{U}\mathbf{x} \quad (8.25)$$

$$(\mathbf{U}\mathbf{A}\mathbf{U}^{-1})(\mathbf{U}\mathbf{x}) = \lambda \mathbf{U}\mathbf{x} \quad (8.26)$$

until one is found for which  $\mathbf{U}\mathbf{A}\mathbf{U}^{-1}$  is diagonal:

$$\mathbf{U}\mathbf{A}\mathbf{U}^{-1} = \begin{pmatrix} \lambda'_1 & & \cdots & 0 \\ 0 & \lambda'_2 & \cdots & 0 \\ 0 & 0 & \lambda'_3 & \cdots \\ 0 & \cdots & & \lambda'_N \end{pmatrix} \quad (8.27)$$

The diagonal values of  $\mathbf{U}\mathbf{A}\mathbf{U}^{-1}$  are the eigenvalues, with eigenvectors

$$\mathbf{x}_i = \mathbf{U}^{-1}\hat{\mathbf{e}}_i \quad (8.28)$$

that is, the eigenvectors are the columns of the matrix  $\mathbf{U}^{-1}$ . A number of routines of this type are found in the subroutine libraries.

### 8.2.1

#### Practical Aspects of Matrix Computing

Many scientific programming bugs arise from the improper use of arrays.<sup>2</sup> This may be due to the extensive use of matrices in scientific computing or the complexity of keeping track of indices and dimensions. In any case, here are some rules of thumb to observe.

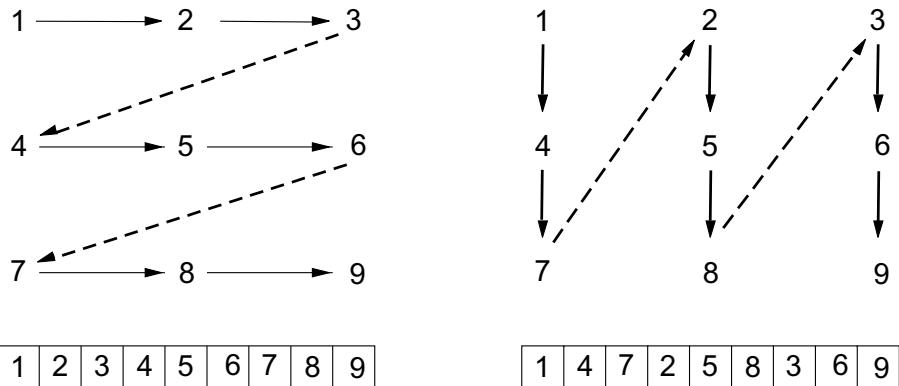
<sup>2</sup> Even a vector  $V(N)$  is called an "array," albeit a 1D one.

- **Computers are finite:** Unless you are careful, you can run out of memory or run very slowly when dealing with large matrices. As a case in point, let us say that you store data in a 4D array with each index having a *physical dimension* of 100:  $a[100][100][100][100]$ . This array of  $(100)^4$  64-byte words occupies  $\approx 1\text{ GB}$  (gigabyte) of memory.
- **Processing time:** Matrix operations such as inversion require on the order of  $N^3$  steps for a square matrix of dimension  $N$ . Therefore, doubling the dimensions of a 2D square matrix (as happens when the number of integration steps are doubled) leads to an *eightfold* increase in processing time.
- **Paging:** Many computer systems have *virtual memory* in which disk space is used when a program runs out of RAM (see Chap. 13 for a discussion of how computers arrange memory). This is a slow process that requires writing a full *page* of words to the disk. If your program is near the memory limit at which paging occurs, even a slight increase in the dimensions of a matrix may lead to an order-of-magnitude increase in running time.
- **Matrix storage:** While we may think of matrices as multidimensional blocks of stored numbers, the computer stores them sequentially as linear strings of numbers. For instance, a matrix  $a(3,3)$  in Java and C is stored in *row-major order* (Fig. 8.3, left), while in Fortran it is stored in *column-major order* as (Fig. 8.3, right):
 

```
a(1,1) a(2,1) a(3,1) a(1,2) a(2,2) a(3,2) a(1,3) a(2,3) a(3,3),
```

 It is important to keep this linear storage scheme in mind in order to write a proper code and to permit mixing Fortran and C programs.
- **Subscript 0:** It is standard in C and Java to have array indices begin with the value 0. While this is now permitted in Fortran, the standard has been to start indices at 1. On that account, the same matrix element will be referenced differently in the different languages:

Location	Java/C element	Fortran element
Lowest	$a[0][0]$	$a(1,1)$
	$a[0][1]$	$a(2,1)$
	$a[1][0]$	$a(3,1)$
	$a[1][1]$	$a(1,2)$
	$a[2][0]$	$a(2,2)$
Highest	$a[2][1]$	$a(3,2)$



**Fig. 8.3** Left: Row-major order used for matrix storage in C and Pascal. Right: column-major order used for matrix storage in Fortran. On the bottom is shown how successive matrix elements are stored in a linear fashion in memory.

Care is also needed when programming up equations in the two schemes because the implementation of the mathematics may differ. To illustrate, the Legendre polynomials are often computed via a recursion relation

$$(l+1)P_{l+1} - (2l+1)xP_l + lP_{l-1} = 0 \quad (l = 0, 1, \dots) \quad (8.29)$$

starting with  $P_{-1} = 0$  and  $P_0 = 1$ . However, if the index scheme starts at  $l = 1$ , you would need to use the mathematically equivalent, but computationally different, relation

$$iP_i - (2i-1)xP_{i-1} + (i-1)P_{i-2} = 0 \quad (i = 1, 2, \dots) \quad (8.30)$$

- **Physical and logical dimensions\***: When you run a program, you issue commands such as `double a[3][3]` or `Dimension a(3, 3)` that tell the compiler how much memory it needs to set aside for the array `a`. This is called *physical memory*. Sometimes you may run programs without the full complement of values declared in the declaration statements, for example, as a test case. The amount of memory you actually use to store numbers is the matrix's *logical size*.

Modern programming techniques, such as those used in Java, C, and Fortran90, permit *dynamic memory allocation*, that is, you may use variables as the dimension of your arrays and read in the values of the variables at run time. With these languages you should read in the sizes of your arrays at run time, and thus give them the same physical and logical sizes. However, Fortran77, which is the language used for many library routines, required the dimensions to be specified at compile time, and so the physical and logical sizes may well differ. To see why care is needed if the physical and logical sizes of the arrays differ, imagine that you declared `a[3][3]`, but defined elements only up to `a[2][2]`. Then the `a` in storage would look like

```
a[1][1]' a[1][2]' a[1][3] a[2][1]' a[2][2]' a[2][3] a[3][1] a[3][2] a[3][3]
```

where only the prime elements have values assigned to them. Clearly, the defined `a` values do not occupy sequential locations in memory, and so an algorithm processing this matrix cannot assume that the next element in memory is the next element in your array. This is the reason why subroutines from a library usually need to know *both* the physical and logical sizes of your arrays.

- **Passing sizes to subprograms\***: It is needed when the logical and physical dimensions of arrays differ, as is true with some library routines, but probably not with the programs you write. In cases such as using external libraries, you must also watch that the sizes of your matrices do not exceed the bounds which have been declared in the subprograms. This may occur *without* an error message, and probably will give you the wrong answers. In addition, if you are running a C program that calls a Fortran subroutine (something we discuss in Section 9.5), you need to pass *pointers* to variables and not the actual values of the variables to the Fortran subprograms (Fortran makes *reference calls*, which means it deals only with pointers as subprogram arguments). Here we have a program possibly ruining some data stored nearby:

<pre>Program Main   Dimension a(100), b(400)</pre>	<i>// In main program</i>
<pre>Subroutine Sample(a)   Dimension a(10)   a(300) = 12</pre>	<i>// In subroutine</i> <i>// Smaller dimension</i> <i>// Out of bounds, but no message</i>

One way to ensure size compatibility among main programs and subroutines is to declare array sizes only in your main program and then to pass those sizes along to your subprograms as arguments.

- **Equivalence, pointers, references manipulations\***: Once upon a time computers had such limited memories that programmers would conserve memory by having different variables occupy the *same* memory location; the theory being that this would cause no harm as long as these variables were not being used at the same time. This was done by use of `Common` and `Equivalence` statements in Fortran, and by manipulations using pointers and references in other languages. These types of manipulations are obsolete (the bane of object-oriented programming) and can cause endless grief; *do not do them unless it is a matter of life or death!*
- **Say what is happening**: You decrease errors and problems by using self-explanatory labels for your indices (subscripts), telling what your variables

mean, and describing your storage schemes.

- **Clarity versus efficiency:** When dealing with matrices, you have to balance the clarity of the operations being performed against the efficiency with which the computer does them. For example, having one matrix with many indices such as  $V[L, Nre, Nspin, k, kp, Z, A]$  may be neat packaging, but it may require the computer to jump through large blocks of memory to get to the particular values needed (large *strides*) as you vary  $k$ ,  $kp$ , and  $Nre$ . The solution would be to have several matrices such as  $V1[Nre, Nspin, k, kp, Z, A]$ ,  $V2[Nre, Nspin, k, kp, Z, A]$ , and  $V3[Nre, Nspin, k, kp, Z, A]$ .
- **Tests:** Always test a library routine on a small problem whose answer you know. Then you will know if you are supplying it with the right arguments and if you have all the links working. We give exercises in Section 8.2.3 that are useful for this purpose.

### 8.2.2

#### Implementation: Scientific Libraries, WWW

Some major scientific and mathematical libraries available include (search the Web for them):

NETLIB	A WWW metalib of free math libraries	ScaLAPACK	Distributed Memory LAPACK
LAPACK	Linear Algebra Pack	JLAPACK	LAPACK library in Java
SLATEC	Comprehensive Math & Statistical Pack	ESSL	Engr & Sci Subroutine Lib (IBM)
IMSL	International Math & Statistical Libs	CERNLIB	European Cntr Nuclear Research
BLAS	Basic Linear Algebra Subprograms	JAMA	Java Matrix Lib
NAG	Numerical Algorithms Group (UK Labs)	Lapack++	Linear Algebra in C++
TNT	C++ Template Numerical Toolkit	GNUMATH	Linear Algebra in C & C++

Except for ESSL, IMSL, and NAG, all these libraries are in the public domain. However, even these proprietary (\$\$\$) ones are frequently available on a central computer or via an institution-wide site license. General subroutine libraries are treasures to possess because they typically contain routines for almost everything you might want to do, such as

Linear algebra manipulations	Matrix operations	Interpolation, fitting
Eigenvalue analysis	Signal processing	Sorting and searching
Solution of linear equations	Differential equations	Roots, zeros, & extrema
Random-number operations	Statistical functions	Numerical quadrature

LAPACK is a free, portable, modern (1990) library of Fortran 77 routines for solving the most common problems in numerical linear algebra. It is designed to be efficient on a wide range of high-performance computers, under the proviso that the hardware vendor has implemented an efficient set of BLAS (Basic Linear Algebra Subroutines). The name LAPACK is an acronym for Linear Algebra Package. In contrast to LAPACK, the SLATEC library contains general purpose mathematical and statistical Fortran routines, and is consequently more general. Nonetheless, it is not tuned to the architecture of a particular machine the way LAPACK is.

Often a subroutine library will supply only Fortran routines, and this requires the C programmer to call a Fortran routine (we describe how to do that in Section 9.5). In some cases, C-language routines may also be available, but they may not be optimized for a particular machine.

#### 8.2.2.1 JAMA: Java Matrix Library<sup>⊖</sup>

JAMA is a basic linear algebra package for Java, developed at the US National Institute of Science [11]. We recommend it since it works well, is natural and understandable to nonexperts, is free, helps make scientific codes more universal and portable, and because not much else is available. JAMA provides object-oriented classes that construct true `Matrix` objects, that add and multiply matrices, that solve matrix equations, and prints out entire matrices in aligned row-by-row format. JAMA is intended to serve as the standard matrix class for Java.<sup>3</sup>

#### 8.2.2.2 JAMA Examples

The first example deals with  $\mathbf{Ax} = \mathbf{b}$  for a  $3 \times 3$   $\mathbf{A}$  and  $3 \times 1$   $\mathbf{x}$  and  $\mathbf{b}$ :

```
double[][] array = { {1.,2.,3}, {4.,5.,6}, {7.,8.,10.} }; Matrix A
= new Matrix(array); Matrix b = Matrix.random(3,1); Matrix x =
A.solve(b); Matrix Residual = A.times(x).minus(b);
```

The vectors and matrices are declared and created as `Matrix` variables, with  $\mathbf{b}$  given random values. It then solves the  $3 \times 3$  linear system of equations  $\mathbf{Ax} = \mathbf{b}$  with the single command `Matrix x = A.solve(b)`, and computes the residual  $\mathbf{Ax} - \mathbf{b}$  with the command `Residual = A.times(x).minus(b)`.

<sup>3</sup> A sibling matrix package, *Jampack* [11], has also been developed at NIST and the University of Maryland, and it works for complex matrices as well.

**Listing 8.1:** The sample program `JamaEigen.java` uses the JAMA matrix library to solve the eigenvalue problem. Note that JAMA defines and manipulates the new data type (object) `Matrix`, which differs from an array, but can be created from one.

```
/*
 * JamaEigen.java: eigenvalue problem with NIST JAMA
 * JAMA must be in same directory, or include JAMA in CLASSPATH
 * uses Matrix.class, see Matrix.java or HTML documentation */
import Jama.*;
import java.io.*;

public class JamaEigen {

    public static void main(String[] argv) {
        double[][] I = { {2./3,-1./4,-1./4}, {-1./4,2./3,-1./4},
                         {-1./4,-1./4,2./3} };
        Matrix MatI = new Matrix(I);           // Form Matrix from 2D arrays
        System.out.print("Input Matrix" );
        MatI.print(10, 5);                  // Jama Matrix print

        // Jama eigenvalue finder
        EigenvalueDecomposition E = new EigenvalueDecomposition(MatI);
        double[] lambdaRe = E.getRealEigenvalues();           // Real eigens
        double[] lambdaIm = E.getImagEigenvalues();          // Imag eigens
        System.out.println("Eigenvalues: \t lambdaRe[]=" +
                           + lambdaRe[0]+", "+lambdaRe[1]+", "+lambdaRe[2]);
        // Get matrix of eigenvectors
        Matrix V = E.getV();
        System.out.print("\n Matrix with column eigenvectors ");
        V.print(10, 5);
        Matrix Vec = new Matrix(3,1);           // Extract single eigenvector
        Vec.set( 0, 0, V.get(0, 0) );
        Vec.set( 1, 0, V.get(1, 0) );
        Vec.set( 2, 0, V.get(2, 0) );
        System.out.print("First Eigenvector, Vec" );
        Vec.print(10,5);
        Matrix LHS = MatI.times(Vec);         // Should get Vec as answer
        Matrix RHS = Vec.times(lambdaRe[0]);
        System.out.print("Does LHS = RHS?" );
        LHS.print(18, 12);
        RHS.print(18, 12);
    }
}
```

Our second JAMA example arises in the solution for the principal-axes system for a cube, and requires us to find a coordinate system in which the inertia tensor is diagonal. This entails solving the eigenvalue problem,

$$\mathbf{I}\vec{\omega} = \lambda\vec{\omega} \quad (8.31)$$

where  $\mathbf{I}$  is the original inertia matrix,  $\vec{\omega}$  is an eigenvector,  $\lambda$  is an eigenvalue, and we are using arrows to indicate vectors. The program `JamaEigen.java` in Listing 8.1 solves for the eigenvalues and vectors, and produces output of the form

```

Input Matrix
 0.66667   -0.25000   -0.25000
 -0.25000   0.66667   -0.25000
 -0.25000   -0.25000   0.66667

Eigenvalue: lambda.Re[] = 0.1666666666666666, 0.9166666666666666,
0.9166666666666666

Matrix with column eigenvectors
 -0.57735   -0.70711   -0.40825
 -0.57735    0.70711   -0.40825
 -0.57735    0.00000    0.81650

First Eigenvector, Vec
 -0.57735
 -0.57735
 -0.57735

Does LHS = RHS?
 -0.096225044865   -0.096225044865
 -0.096225044865   -0.096225044865
 -0.096225044865   -0.096225044865

```

Look at `JamaEigen` and notice how we first set up the array `I` with all the elements of the inertia tensor, and then we create a matrix `MatI` with the same elements as the array. We then solve the eigenvalue problem with the creation of an eigenvalue object `E` via the JAMA command:

```
EigenDecomposition E = new EigenDecomposition(MatI);
```

Next we extract (get) a vector `lambdaRe` of length 3 containing the three (real) eigenvalues, `lambdaRe[0], lambdaRe[1], lambdaRe[2]`:

```
14 double[] lambdaRe = E.getRealEigenvalues();
```

Next we create a  $3 \times 3$  matrix `v` containing the eigenvectors in the three columns of the matrix with the JAMA command:

```
Matrix V = E.getV();
```

which takes the eigenvector object `E` and gets the vectors from it. Then we form a vector `vec` (a  $3 \times 1$  `Matrix`) containing a single eigenvector by extracting the elements from `v` with a `get` method, and assigning them with a `set` method:

```
Vec.set(0,0,v.get(0,0));
```

Our final JAMA example, `JamaFit.java` in Listing 8.2, demonstrates many of the features of JAMA. It arises in the context of least-square-fitting, as discussed in Section 9.4 where we give the equations being used to fit the parabola  $y(x) = b_0 + b_1x + b_2x^2$  to a set of  $N_D$  measured data points  $(y_i, y_i \pm \sigma_i)$ .

**Listing 8.2:** `JamaFit.java` performs a least-squares fit of a parabola to data using the JAMA matrix library to solve the set of linear equations  $\mathbf{ax} = \mathbf{b}$ . For illustration, the equation is solved both directly and by matrix inversion, and several techniques for assigning values to JAMA's `Matrix` are used.

```

/*
 * JamaFit: NIST JAMA matrix libe least-squares parabola fit
 * y(x) = b0 + b1 x + b2 xx JAMA libe must be in same directory as
 * program, or modify CLASSPATH to include JAMA
 */
import Jama.*;
```

```

import java.io.*;
public class JamaFit {
    public static void main(String[] argv) {
        double [] x = {1., 1.05, 1.15, 1.32, 1.51, 1.68, 1.92};
        double [] y = {0.52, 0.73, 1.08, 1.44, 1.39, 1.46, 1.58};
        double [] sig = {0.1, 0.1, 0.2, 0.3, 0.2, 0.1, 0.1};
        int Nd = 7; // Number of data points
        double sig2, s, sx, sxx, sy, sxxx, sxxxx, sxy, sxxy, rhl;
        double [][] Sx = new double[3][3]; // Create 3x3 array
        double [][] Sy = new double[3][1]; // Create 3x1 array

        s = sx = sxx = sy = sxxx = sxxxx = sxy = sxxy = sxxxy = 0;
        // Generate matrix elements
        for (int i=0; i <= Nd-1; i++) {
            sig2 = sig[i]*sig[i];
            s += 1./sig2;
            sx += x[i]/sig2;
            sy += y[i]/sig2;
            rhl = x[i]*x[i];
            sxx += rhl/sig2;
            sxy += rhl*y[i]/sig2;
            sxxx += x[i]*y[i]/sig2;
            sxxxx += rhl*x[i]/sig2;
            sxxy += rhl*rhl/sig2;
        }
        // Assign arrays
        Sx[0][0] = s;
        Sx[0][1] = Sx[1][0] = sx;
        Sx[0][2] = Sx[2][0] = Sx[1][1] = sxx;
        Sx[1][2] = Sx[2][1] = sxxx;
        Sx[2][2] = sxxxx;
        Sy[0][0] = sy;
        Sy[1][0] = sxy;
        Sy[2][0] = sxxy;
        // Form Jama Matrices
        Matrix MatSx = new Matrix(Sx);
        Matrix MatSy = new Matrix(3, 1);
        MatSy.set(0, 0, sy);
        MatSy.set(1, 0, sxy);
        MatSy.set(2, 0, sxxy);
        // Determine inverse
        Matrix B = MatSx.inverse().times(MatSy);
        Matrix Itest = MatSx.inverse().times(MatSx); // Test inverse
        // Jama print
        System.out.print("B Matrix via inverse");
        B.print(16, 14);
        System.out.print("MatSx.inverse().times(MatSx) ");
        Itest.print(16, 14); // Direct solution too
        B = MatSx.solve(MatSy);
        System.out.print("B Matrix via direct");
        B.print(16, 14);
        // Extract using Jama get & Print parabola coefficients
    }
}

```

```

System.out.println("FitParabola2 Final Results");
System.out.println("\n");
System.out.println("y(x) = b0 + b1 x + b2 x^2");
System.out.println("\n");
System.out.println("b0 = "+B.get(0,0));
System.out.println("b1 = "+B.get(1,0));
System.out.println("b2 = "+B.get(2,0));
System.out.println("\n");
System.out.println("// Test fit
for (int i=0; i <= Nd-1; i++) {
    s=B.get(0,0)+B.get(1,0)*x[i]+B.get(2,0)*x[i]*x[i];
    System.out.println
        ("i, x, y, yfit = "+i+", "+x[i]+", "+y[i]+", "+s);
}
}

```

### 8.2.2.3 Other Netlib Libraries

Our example of using LAPACK has assumed that someone has been nice and placed the library on the computer. Life can be rough, however, and you may have to get the routines yourself. Probably the best place to start looking for them is Netlib, a repository of free software, documents, and databases of interest to computational scientists. Information, as well as subroutines, is available over the internet.

### 8.2.2.4 SLATEC Common Math Library

SLATEC (Sandia, Los Alamos, Air Force Weapons Laboratory Technical Exchange Committee) is a portable, nonproprietary, mathematical subroutine library available from Netlib. We recommend it highly and also recommend that you get more information about it from Netlib. SLATEC (CML) contains over 1400 general purpose mathematical and statistical Fortran routines. It is more general than LAPACK, which is devoted to linear algebra, yet not necessarily tuned to the particular architecture of a machine the way LAPACK is. The full library contains a guide, table of contents, and documentation via comments in the source code. The subroutines are classified by the Guide to Available Mathematical Software (GAMS) system.

For our masses on strings **Problem** we have found the needed routines:

<b>snsq-s, dnsq-d</b>	Find zero of $n$ -variable, nonlinear function
<b>snsqe-s, dnsqe-d</b>	Easy-to-use snsq

If you extract these routines, you will find that they need the following:

enorm.f	j4save.f	r1mach.f	xerprn.f	fdjac1.f	r1mpyq.f
xercnt.f	xersve.f	fdump.f	qform.f	r1upd.f	xerhlt.f
xgetua.f	dogleg.f	i1mach.f	qrfac.f	snsq.f	xerrmsg.f

Of particular interest in these “helper” routines are *i1mach.f*, *r1mach.f*, and *d1mach.f*. They tell LAPACK the characteristic of your particular machine when

the library is first installed. Without that knowledge, LAPACK does not know when convergence is obtained or what step sizes to use.

### 8.2.3

#### Exercises for Testing Matrix Calls

Before you direct the computer to go off crunching numbers on the million elements of some matrix, it is a good idea for you to try out your procedures on a small matrix, especially one for which you know the right answer. In this way it will only take you a short time to realize how hard it is to get the calling procedure perfectly right! Here are some exercises:

1. Find the inverse of  $\mathbf{A} = \begin{pmatrix} 4 & -2 & 1 \\ 3 & 6 & -4 \\ 2 & 1 & 8 \end{pmatrix}$ .

(a) As a general procedure, applicable even if you do not know the analytic answer, check your inverse in both directions; that is, check that  $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ .

(b) Verify that  $\mathbf{A}^{-1} = \frac{1}{263} \begin{pmatrix} 52 & 17 & 2 \\ -32 & 30 & 19 \\ -9 & -8 & 30 \end{pmatrix}$ .

2. Consider the same matrix  $\mathbf{A}$  as before, now used to describe three simultaneous linear equations,  $\mathbf{Ax} = \mathbf{b}$ , or explicitly,

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Here the vector  $\mathbf{b}$  on the RHS is assumed to be known, and the problem is to solve for the vector  $\mathbf{x}$ . Use an appropriate subroutine to solve these equations for the three different  $\mathbf{x}$  vectors appropriate to these three different  $\mathbf{b}$  values on the RHS:

$$b_1 = \begin{pmatrix} +12 \\ -25 \\ +32 \end{pmatrix}, \quad b_2 = \begin{pmatrix} +4 \\ -10 \\ +22 \end{pmatrix}, \quad b_3 = \begin{pmatrix} +20 \\ -30 \\ +40 \end{pmatrix}$$

The solutions should be

$$x_1 = \begin{pmatrix} +1 \\ -2 \\ +4 \end{pmatrix}, \quad x_2 = \begin{pmatrix} +0.312 \\ -0.038 \\ +2.677 \end{pmatrix}, \quad x_3 = \begin{pmatrix} +2.319 \\ -2.965 \\ +4.790 \end{pmatrix}$$

3. Consider the matrix  $\mathbf{A} = \begin{pmatrix} \alpha & \beta \\ -\beta & \alpha \end{pmatrix}$ , where you are free to use any values you want for  $\alpha$  and  $\beta$ . Use a numerical eigenproblem solver to show that the eigenvalues and eigenvectors are the complex conjugates:

$$\mathbf{x}_{1,2} = \begin{pmatrix} +1 \\ \mp i \end{pmatrix} \quad \lambda_{1,2} = \alpha \mp i\beta$$

4. Use your eigenproblem solver to find the eigenvalues of the matrix

$$\mathbf{A} = \begin{pmatrix} -2 & +2 & -3 \\ +2 & +1 & -6 \\ -1 & -2 & +0 \end{pmatrix}$$

- (a) Verify that you obtain the eigenvalues  $\lambda_1 = 5, \lambda_2 = \lambda_3 = -3$ . Notice that double roots can cause problems. In particular, there is a uniqueness problem with their eigenvectors because any combinations of these eigenvectors would also be an eigenvector.
- (b) Verify that the eigenvector for  $\lambda_1 = 5$  is proportional to

$$\mathbf{x}_1 = \frac{1}{\sqrt{6}} \begin{pmatrix} -1 \\ -2 \\ +1 \end{pmatrix}$$

- (c) The eigenvalue  $-3$  corresponds to a double root. This means that the corresponding eigenvectors are degenerate, which, in turn, means that they are not unique. Two linearly independent ones are

$$\mathbf{x}_2 = \frac{1}{\sqrt{5}} \begin{pmatrix} -2 \\ +1 \\ +0 \end{pmatrix} \quad \mathbf{x}_3 = \frac{1}{\sqrt{10}} \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix}$$

In this case it is not clear what your eigenproblem solver will give for the eigenvectors. Try to find a relationship between your computed eigenvectors with the eigenvalue  $-3$  to these two linearly independent ones.

5. You are investigating a physical system that you model as the  $N = 100$  coupled, linear equations in  $N$  unknowns:

$$a_{11}y_1 + a_{12}y_2 + \cdots + a_{1N}y_N = b_1$$

$$a_{21}y_1 + a_{22}y_2 + \cdots + a_{2N}y_N = b_2$$

...

$$a_{N1}y_1 + a_{N2}y_2 + \cdots + a_{NN}y_N = b_N$$

In many cases, the  $a$  and  $b$  values are known, so your exercise is to solve for all the  $x$  values, taking  $\mathbf{a}$  as the *Hilbert* matrix, and  $\mathbf{b}$  as its first row:

$$[a_{ij}] = \mathbf{a} = \left[ \frac{1}{i+j-1} \right] = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{100} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{101} \\ \ddots & & & & & \\ \frac{1}{100} & \frac{1}{101} & \cdots & \cdots & & \frac{1}{199} \end{pmatrix}$$

$$[b_i] = \mathbf{b} = \left[ \frac{1}{i} \right] = \begin{pmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{3} \\ \ddots \\ \frac{1}{100} \end{pmatrix}$$

Compare to the analytic solution

$$\begin{pmatrix} y_1 \\ y_2 \\ \ddots \\ y_N \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \ddots \\ 0 \end{pmatrix}$$

#### 8.2.4

#### **Matrix Solution of Problem II**

We have now set up the solution to our problem of two mass on a string. Your **problem** will be to check out the physical reasonableness of the solution for a variety of weights and lengths. You should check that the deduced tensions are positive and that the deduced angles correspond to a physical geometry (to illustrate, with a sketch). Since this is a physics-based problem, we know that the sine and cosine functions must be less than 1 in magnitude, and that the tensions should be of similar magnitude to the weights of the spheres.

#### 8.2.5

#### **Explorations**

1. See at what point your initial guess gets so bad that the computer is unable to find a physical solution.
2. A possible problem with the formalism we have just laid out is that by incorporating the identity  $\sin^2 \theta_i + \cos^2 \theta_i = 1$  into the equations, we may be discarding some information about the sign of  $\sin \theta$  or  $\cos \theta$ . If you look at Fig. 8.1 you can observe that for some values of the weights and lengths,  $\theta_2$  may turn out to be negative, yet  $\cos \theta$  should remain

positive. We can build this condition into our equations by replacing  $f_7 - f_9$  with  $f$ 's based on the form

$$f_7 = x_4 - \sqrt{1 - x_1^2} \quad f_8 = x_5 - \sqrt{1 - x_2^2} \quad f_9 = x_6 - \sqrt{1 - x_3^2} \quad (8.32)$$

See if this makes any difference in the solutions obtained.

- 2.\* Solve the similar three-mass problem. The approach is the same, but the number of equations gets larger.

**9****Data Fitting**

*Data fitting is an art worthy of serious study. In this unit we just scratch the surface. We examine how to interpolate within a table of numbers and how to do a least-squares fit for linear functions. If a least-squares fit is needed for nonlinear functions, then some of the search routines, either your own or those obtained from scientific subroutine libraries, may be used. We also describe how to do that. In a recent work [12], simulated annealing (which we describe in Chap. 28) is used to assist the search in least-squares fitting.*

**9.1****Fitting Experimental Spectrum (Problem IIIA)**

The cross section measured for the resonant scattering of a neutron from a nucleus is given in Tab. 9.1. The first row is the energy; the second row, the experimental cross section; and the third row, the experimental error for each measurement. Your **problem** is to determine values for the cross sections at values of energy lying between those measured by experiment.

**Tab. 9.1** Experimental values for a scattering cross section  $\Sigma$  as a function of energy.

$i$	1	2	3	4	5	6	7	8	9
$E_i$ (MeV) [ $\equiv x_i$ ]	0	25	50	75	100	125	150	175	200
$f(E_i)$ (mb)	10.6	16.0	45.0	83.5	52.8	19.9	10.8	8.25	4.7
Error = $\pm \sigma_i$ (mb)	9.34	17.9	41.5	85.5	51.5	21.5	10.8	6.29	4.09

You can view your **problem** in a number of ways. The most direct is to numerically *interpolate* between the values of the experimental  $\Sigma(E_i)$  given in Tab. 9.1. This is direct and easy, but it ignores the possibility of there being experimental noise in the data in that it assumes that the data can be represented as a polynomial in  $E$ , at least over some small range.

In Section 9.4 we discuss another way to view this problem. Specifically, we start with what one believes to be the “correct” theoretical description of the

data,

$$f(E) = \frac{f_r}{(E - E_r)^2 + \Gamma^2/4} \quad (9.1)$$

where are constants to be determined by the fitting, and then adjust the parameters  $f_r$ ,  $E_r$ , and  $\Gamma$  to obtain a *best fit* to the data.<sup>1</sup> This is a “best” fit in a statistical sense, but, in fact, may not pass through all (or any) of the data points. For an easy, yet effective, introduction to statistical data analysis, we recommend [13].

These techniques of interpolation and least-squares fitting are powerful tools that let you treat tables of numbers as if they were analytic functions, and sometimes let you deduce statistically meaningful constants or conclusions from measurements. In general, you can view data fitting as *global* or *local*. In global fits, a single function in  $x$  is used to represent the entire set of numbers in a table such as Tab. 9.1. While it may be spiritually satisfying to find a single function that passes through all the data points, if that function is not the correct function for describing the data, the fit may have nonphysical behavior (such as large oscillations) between the data points. The rule of thumb is that if you must interpolate, then keep it local. Although we ask you to do one, global fits should be looked at with a jaundiced eye.

### 9.1.1

#### Lagrange Interpolation (Method)

Consider Tab. 9.1 as ordered data that we wish to interpolate. We call the independent variable  $x$ , with tabulated values  $x_i$  ( $i = 1, 2, \dots$ ), and we assume that the dependent variable is the function  $g(x)$ , with tabulated values  $g_i = g(x_i)$ . We assume that  $g(x)$  can be approximated as a polynomial of degree  $(n - 1)$  in each interval  $i$ :

$$g_i(x) \simeq a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \quad (x \simeq x_i) \quad (9.2)$$

Because our fit is local, we do not assume that one  $g(x)$  can fit all the data in the table, but instead will use a different polynomial, that is, a different set of  $a_i$  values, for each region of the table. While each polynomial is of low degree, there are several polynomials that are needed to cover the entire table. If some care is taken, the set of polynomials so obtained behaves well enough to be used in further calculations without introducing much unwanted noise or discontinuities.

The classic of interpolation formulas was created by Lagrange. He figured out a closed-form one that directly fits the  $(n - 1)$ -order polynomial (9.2) to  $n$

<sup>1</sup> Chapter 17 on Fourier analysis discusses yet another way to fit data.

values of the function  $g(x)$  evaluated at the points  $x_i$ . The formula is written as the sum of polynomials:

$$g(x) \simeq g_1\lambda_1(x) + g_2\lambda_2(x) + \cdots + g_n\lambda_n(x) \quad (9.3)$$

$$\lambda_i(x) = \prod_{j(\neq i)=1}^n \frac{x - x_j}{x_i - x_j} = \frac{x - x_1}{x_i - x_1} \frac{x - x_2}{x_i - x_2} \cdots \frac{x - x_n}{x_i - x_n} \quad (9.4)$$

For three points, (9.3) provides a second-degree polynomial, while for eight points it gives a seventh-degree polynomial. For example, here we use a four-point Lagrange interpolation to determine a third-order polynomial that reproduces each of the tabulated values:

$$\begin{aligned} g(x) &= \frac{(x-1)(x-2)(x-4)}{(0-1)(0-2)(0-4)}(-12) + \frac{x(x-2)(x-4)}{(1-0)(1-2)(1-4)}(-12) \\ &\quad + \frac{x(x-1)(x-4)}{(2-0)(2-1)(2-4)}(-24) + \frac{x(x-1)(x-2)}{(4-0)(4-1)(4-2)}(-60) \\ \Rightarrow g(x) &= x^3 - 9x^2 + 8x - 12 \end{aligned}$$

As a check we see that

$$g(4) = 4^3 - 9(4^2) + 32 - 12 = -60 \quad g(0.5) = -10.125 \quad (9.5)$$

If the data contain little noise, this polynomial can be used with some confidence within the range of data, but with risk beyond the range of data.

Notice that Lagrange interpolation makes no restriction that the points in the table be evenly spaced. As a check, it is also worth noting that the sum of the Lagrange multipliers equals one,  $\sum_{i=1}^n \lambda_i = 1$ . Usually the Lagrange fit is made to only a small region of the table with a small value of  $n$ , even though the formula works perfectly well for fitting a high-degree polynomial to the entire table. The difference between the value of the polynomial evaluated at some  $x$  and that of the actual function is equal to the *remainder*

$$R_n \approx \frac{(x - x_1)(x - x_2) \cdots (x - x_n)}{n!} g^{(n)}(\zeta) \quad (9.6)$$

where  $\zeta$  lies somewhere in the interpolation interval, but is otherwise undetermined. This shows that if significant high derivatives exist in  $g(x)$ , then it cannot be approximated well by a polynomial. In particular, if  $g(x)$  is a table of experimental data, then it is likely to contain noise, and then it is a bad idea to fit a curve through all the data points.

**Listing 9.1:** Lagrange.java makes a Lagrange interpolation of tabulated data.

```
//      Lagrange.java: Langrange interpolation of tabulated data

import java.io.*;                                     //Location of PrintWriter

public class Lagrange {
    public static void main(String[] argv) throws IOException,
                                                FileNotFoundException {
        PrintWriter w = new PrintWriter(
            new FileOutputStream("Lagr.dat"), true);
        PrintWriter ww = new PrintWriter(
            new FileOutputStream("Lagr_input.dat"), true);
        double x, y;
        int i,j,k; int end = 9;                                //Input data
        double xin[] = {0, 25, 50, 75, 100, 125, 150, 175, 200};
        double yin[] = {10.6, 6, 45, 83.5, 52.8, 19.9, 10.8, 88.25, 4.7};

        for (k=0; k<9; k++) ww.println( xin[k] + " " + yin[k]);
        for ( k=0; k<=1000;k++){
            x = k*0.2 ;
            y = inter(xin, yin, end, x);
            System.out.println("Lagrange x=" +x+ " , y= "+y);    // to file
            w.println( x + " " +y);
        }
        System.out.println("Lagrange Program Complete.");
        System.out.println("Fit in Lagr.dat, input in Lagr_input.dat");
    }

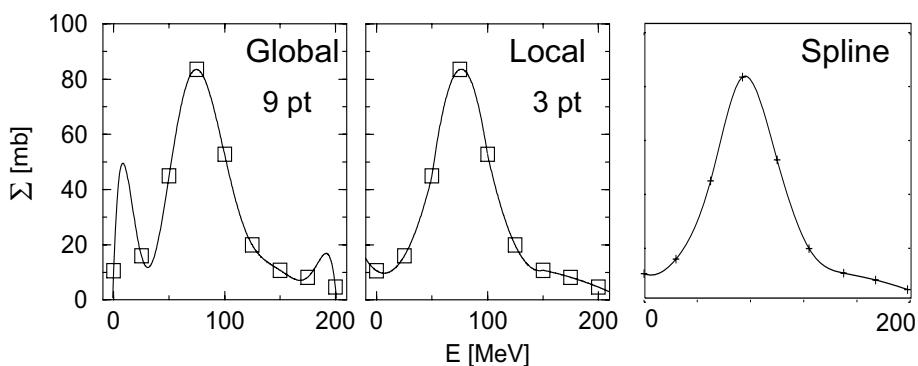
    public static double inter ( double xin[], double yin[],
                                int end, double x) {
        double lambda,y;  int i,j;
        y = 0.0;
        for ( i=1; i <= end; i++) {
            lambda = 1.0;
            for ( j=1; j<=end; j++) if (i != j)
                {lambda *= ((x - xin[j-1])/(xin[i-1] - xin[j-1]));}
            y += (yin[i-1] * lambda);
        }
        return y;
    }
}
```

### 9.1.2

#### Lagrange Implementation and Assessment

Consider the experimental neutron scattering data in Tab. 9.1. The expected theoretical functional form which describes these data is (9.1), and our empirical fits to these data are shown in Fig. 9.1.

1. Write a subroutine to perform an  $n$ -point Lagrange interpolation using (9.3). Treat  $n$  as an arbitrary input parameter. (You can also do this



**Fig. 9.1** *Left:* fit to cross section that places an eighth degree polynomial through all data. *Middle:* fit to cross section using Lagrange interpolation to place a series of third-degree polynomials through the data. *Right:* a cubic-splines fit to same data provides continuous first and second derivatives. Note the differences at low energies of the spline fit from the other two.

exercise with the spline fits discussed in Section 9.1.4.) Our program is given in Listing 9.1.

2. Use the Lagrange interpolation formula to fit the entire experimental spectrum with one polynomial. (This means that you want to fit all nine data points with an eight-degree polynomial.) Then use this fit to plot the cross section in steps of 5 MeV.
3. Use your graph to deduce the resonance energy  $E_r$  (your peak position) and  $\Gamma$  (the full width at half maximum). Compare your results with those predicted by our theorist friend,  $(E_r, \Gamma) = (78, 55)$  MeV.
4. A more realistic use of Lagrange interpolation is for local interpolation with a small number of points, such as three. Interpolate the preceding cross-section data in 5 MeV steps using three-point Lagrange interpolation. (Note, the end intervals may be special cases.)

This example shows how easy it is to go wrong with a high-degree polynomial fit. Although the polynomial is guaranteed to actually pass through all the data points, the representation of the function away from these points can be quite unrealistic. Using a low-order interpolation formula, say,  $n = 2$  or  $3$ , in each interval usually eliminates the wild oscillations. If these local fits are then matched together, as we discuss in the next section, a rather continuous curve results. Nonetheless, you must recall that if the data contain errors, a curve that actually passes through them may lead you astray. We discuss how to do this properly in Section 9.4.

## 9.1.3

**Explore Extrapolation**

We deliberately have not discussed *extrapolation* of data because it can lead to serious *systematic* errors; the answer you get may well depend more on the function you assume than on the data you input. Add some adventure to your life and use the programs you have written to extrapolate to values outside of the table. Compare your results to the theoretical Breit–Wigner shape (9.1).

## 9.1.4

**Cubic Splines (Method)**

If you tried to interpolate the resonant cross section with Lagrange interpolation, then you saw that fitting parabolas (three-point interpolation) within a table may avoid the erroneous and possibly catastrophic deviations of a high-order formula. (Two-point interpolation, which connects the points with straight lines, may not lead you far astray, but it is rarely pleasing to the eye or precise.) A sophisticated variation of  $n = 4$  interpolation, known as *cubic splines*, often leads to surprisingly eye-pleasing fits. In this approach (Fig. 9.1 right), cubic polynomials are fit to the function in each interval, with the additional constraint that the first and second derivatives of the polynomials must be continuous from one interval to the next. This continuity of slope and curvature is what makes the spline fit particularly eye-pleasing. It is analogous to what happens when you use the flexible drafting tool (a lead wire within a rubber sheath) from which the method draws its name.

The series of cubic polynomials obtained by spline-fitting a table can be integrated and differentiated, and is guaranteed to have well-behaved derivatives. The existence of meaningful derivatives is an important consideration. As a case in point, if the interpolated function is a potential, you can take the derivative to obtain the force. The complexity of simultaneously matching polynomials and their derivatives over all the interpolation points leads to many simultaneous, linear equations to be solved. This makes splines unattractive for hand calculations, yet easy for computers. Splines have made recent gains in popularity and applicability in both calculations and graphics. To illustrate, the smooth curves connecting points in most “draw” programs are usually splines.

The basic approximation of splines is the representation of the function  $g(x)$  in the subinterval  $[x_i, x_{i+1}]$  with a cubic polynomial:

$$g(x) \simeq g_i(x) \quad \text{for } x_i \leq x \leq x_{i+1} \quad (9.7)$$

$$g_i(x) = g_i + g'_i(x - x_i) + \frac{1}{2}g''_i(x - x_i)^2 + \frac{1}{6}g'''_i(x - x_i)^3 \quad (9.8)$$

This representation makes it clear that the coefficients in the polynomial equal the values of  $g(x)$  and to its first, second, and third derivatives at the tabulated

points  $x_i$ . Derivatives beyond the third vanish. The computational chore is to determine these derivatives in terms of the  $N$  tabulated values  $g_i$ . The matching of  $g_i$  from one interval to the next (at the *nodes*) provides the equations

$$g_i(x_{i+1}) = g_{i+1}(x_{i+1}) \quad i = 1, N - 1 \quad (9.9)$$

The matching of the first *and* second derivatives at each subinterval's boundary provides the equations

$$g'_{i-1}(x_i) = g'_i(x_i) \quad g''_{i-1}(x_i) = g''_i(x_i) \quad (9.10)$$

To provide the additional equations needed to determine all constants, the third derivatives at adjacent nodes are matched. Values for the third derivatives are found by approximating them in terms of the second derivatives:

$$g'''_i \simeq \frac{g''_{i+1} - g''_i}{x_{i+1} - x_i} \quad (9.11)$$

As discussed in Chap. 15 a *central difference approximation* would be better than the forward difference, yet (9.11) keeps the equations simpler.

It is straightforward though complicated to solve for all the parameters in (9.8). We leave that to other reference sources [9, 14]. We can see, however, that matching at the boundaries of the intervals results in only  $N - 2$  linear equations for  $N$  unknowns. Further input is required. It usually is taken to be the boundary conditions at the endpoints  $a = x_1$  and  $b = x_N$ , specifically, the second derivatives  $g''(a)$ , and  $g''(b)$ . There are several ways to determine these second derivatives.

**Natural spline:** Set  $g''(a) = g''(b) = 0$ , that is, permit the function to have a slope at the endpoints but no curvature. This is “natural” because the derivative vanishes for the flexible spline drafting tool (its ends being free).

**Input values for  $g'$  at boundaries:** The computer uses  $g'(a)$  to approximate  $g''(a)$ . If you do not know the first derivatives, you can calculate them numerically from the table of  $g_i$  values.

**Input values for  $g''$  at boundaries:** Knowing values is of course better than assuming values, but it requires more input. If the values of  $g''$  are not known, they can be approximated by applying a forward-difference approximation to the tabulated values:

$$g''(x) \simeq \frac{[g(x_3) - g(x_2)]/[x_3 - x_2] - [g(x_2) - g(x_1)]/[x_2 - x_1]}{[x_3 - x_1]/2} \quad (9.12)$$

#### 9.1.4.1 Cubic Spline Quadrature (Exploration)

A powerful integration scheme is to fit an integrand with splines, and then integrate the cubic polynomials analytically. If the integrand  $g(x)$  is known

only at its tabulated values, then this is about as good an integration scheme as possible; if you have the ability to actually calculate the function for arbitrary  $x$ , Gaussian quadrature may be preferable. We know that the spline fit to  $g$  in each interval is the cubic (9.8),

$$g(x) \simeq g_i + g'_i(x - x_i) + \frac{1}{2}g''_i(x - x_i)^2 + \frac{1}{6}g'''_i(x - x_i)^3 \quad (9.13)$$

It is easy to integrate this to obtain the integral of  $g$  for this interval, and then to sum over all intervals:

$$\int_{x_i}^{x_{i+1}} g(x) dx \simeq \left( g_i x + \frac{1}{2}g'_i x_i^2 + \frac{1}{6}g''_i x^3 + \frac{1}{24}g'''_i x^4 \right) \Big|_{x_i}^{x_{i+1}} \quad (9.14)$$

$$\int_{x_j}^{x_k} g(x) dx = \sum_{i=j}^k \int_{x_i}^{x_{i+1}} g(x) dx \quad (9.15)$$

Making the intervals smaller does not necessarily increase precision as subtractive cancellations in (9.14) may get large.

### 9.1.5

#### Spline Fit of Cross Section (Implementation)

**Listing 9.2:** The program `SplineAppl.java` performs a cubic spline fit to data. The arrays `x[]` and `y[]` are the data to fit, and the values of the fit at `Nfit` points are output into the file `Spline.dat`.

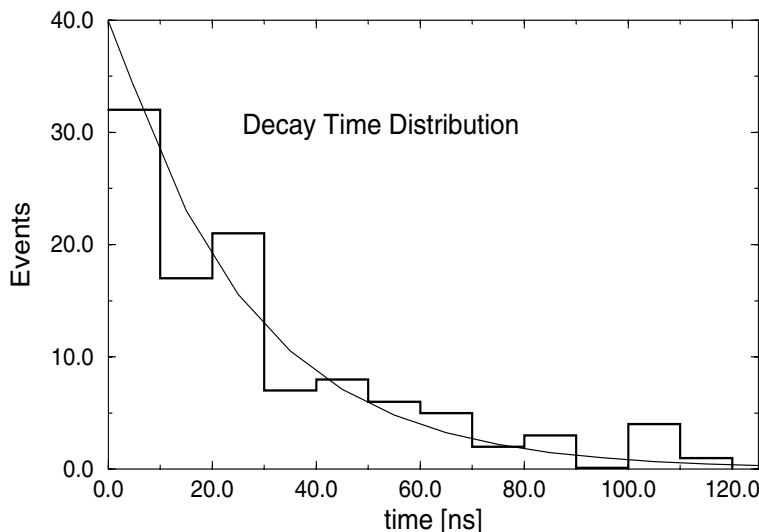
```
/* SplineAppl.java: Application version of cubic spline fitting
   interpolates array x[n], y[n], x0 < x1 ... < x(n-1)
   yp1, ypn: y' at ends, evaluated internally,
   y2[]: y" array; yp1, ypn > e30 for natural spline */
import java.io.*;
public class SplineAppl {
    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
        PrintWriter w =
            new PrintWriter(new FileOutputStream("Spline.dat"), true);
        PrintWriter q =
            new PrintWriter(new FileOutputStream("Input.dat"), true);
        // input
        double x[] = {0., 1.2, 2.5, 3.7, 5., 6.2, 7.5, 8.7, 9.9};
        double y[] = {0., 0.93, .6, -0.53, -0.96, -0.08, 0.94, 0.66, -0.46};
        int i, n = x.length, np = 15, klo, khi, k;
        double y2[] = new double[9]; double u[] = new double[n];
        double h, b, a, Nfit, p, qn, sig, un, yp1, ypn, xout, yout;
        // Output data
        for (i=0; i < n; i++) q.println (" " + x[i] + " " + y[i] + " ");
        // N output pts
        // Initial 1st deriv
```

```

yp1 = (y[1]-y[0])/(x[1]-x[0])
      - (y[2]-y[1])/(x[2]-x[1]) + (y[2]-y[0])/(x[2]-x[0]);
ypn = (y[n-1]-y[n-2])/(x[n-1]-x[n-2]) - (y[n-2]-y[n-3])
      /(x[n-2]-x[n-3]) + (y[n-1]-y[n-3])/(x[n-1]-x[n-3]);
// Natural spline
if (yp1 > 0.99e30) y2[0] = u[0] = 0. ;
else {
    y2[0] = (-0.5);
    u[0] = (3./(x[1]-x[0]))*((y[1]-y[0])/(x[1]-x[0])-yp1);
}
// Decomposition loop
for ( i=1; i <= n-2; i++ ) {
    sig = (x[i]-x[i-1])/(x[i+1]-x[i-1]);
    p = sig*y2[i-1] + 2. ;
    y2[i] = (sig-1.)/p;
    u[i] = (y[i+1]-y[i])/(x[i+1]-x[i])-(y[i]-y[i-1])/(x[i]-x[i-1]);
    u[i] = (6.*u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;
}
// Test for natural
if (ypn > 0.99e30) qn = un = 0. ;
else {
    qn = 0.5;
    un =
        (3./(x[n-1]-x[n-2]))*(ypn-(y[n-1]-y[n-2])/(x[n-1]-x[n-2]));
}
y2[n-1] = (un-qn*u[n-2])/(qn*y2[n-2] + 1.);
for ( k = n-2; k>= 0; k--) y2[k] = y2[k]*y2[k+1] + u[k];
// initialization ends, begin fit
for ( i=1; i <= Nfit; i++ ) { // Loop over xout
    xout = x[0] + (x[n-1]-x[0])*(i-1)/(Nfit);
    klo = 0;
    khi = n-1; // xout value
    // Bisection algor, klo, khi bracket
    while (khi-klo > 1) {
        k = (khi + klo) >> 1;
        if (x[k] > xout) khi = k; else klo = k;
    }
    h = x[khi]-x[klo];
    if (x[k] > xout) khi = k; else klo = k;
    h = x[khi]-x[klo];
    a = (x[khi]-xout)/h;
    b = (xout-x[klo])/h;
    yout = (a*y[klo] + b*y[khi] + ((a*a*a-a)*y2[klo]
        + (b*b*b-b)*y2[khi]))*(h*h) / 6.;
    w.println (" " + xout + " " + yout + " ");
}
System.out.println("data stored in Spline.dat");
}
}

```

Fitting a series of cubics to data is a little complicated to program up yourself, so we recommend using a library routine. While we have found quite a few Java-based spline applications available on the internet, none seemed appropriate for interpreting a simple set of numbers. That being the case, we have adapted the `splint.c` and the `spline.c` functions from [9] to produce



**Fig. 9.2** A reproduction of the experimental measurement in [15] of the number of decays of a  $\pi$  meson as a function of time. Measurements are made during time intervals of 10-ns length. Each “Event” corresponds to a single decay.

the `SplineAppl.java` program shown in Listing 9.2. Your problem now is to carry out the assessment of Section 9.1.2 using cubic spline interpolation rather than Lagrange interpolation.

## 9.2

### Fitting Exponential Decay (Problem IIIB)

Figure 9.2 presents actual experimental data on the number of decays  $\Delta N$  of the  $\pi$  meson as a function of time [15]. Notice that the time has been “binned” into  $\Delta t = 10$  ns intervals, and that the smooth curve gives the theoretical exponential decay law. Your **problem** is to deduce the lifetime  $\tau$  of the  $\pi$  meson from these data (the tabulated lifetime of the pion is  $2.6 \times 10^{-8}$  s).

#### 9.2.1

##### Theory to Fit

Assume that we start with  $N_0$  particles at time  $t = 0$  that can decay to other particles.<sup>2</sup> If we wait a short time  $\Delta t$ , then a small number  $\Delta N$  of the particles will decay *spontaneously*, that is, with no external influences. This decay is a stochastic process, which means that there is an element of chance involved

<sup>2</sup> Spontaneous decay is discussed further and simulated in Section 11.2.

in just when a decay will occur, and so no two experiments are expected to give exactly the same results. The basic law of nature for spontaneous decay is that the number of decays  $\Delta N$  in the time interval  $\Delta t$  is proportional to the number of particles  $N(t)$  present at that time, and to the time interval

$$\Delta N(t) = -\frac{1}{\tau} N(t) \Delta t \quad \Rightarrow \quad \frac{\Delta N(t)}{\Delta t} = -\lambda N(t) \quad (9.16)$$

Here  $\tau = 1/\lambda$  is the *lifetime* of the particle, with  $\lambda$  the rate parameter. The actual decay *rate* is given by the second equation in (9.16). If the number of decays  $\Delta N$  is very small compared to the number of particles  $N$ , and if we look at vanishingly small time intervals, then the difference equation (9.16) becomes the differential equation

$$\frac{dN(t)}{dt} \simeq -\lambda N(t) = \frac{1}{\tau} N(t) \quad (9.17)$$

This differential equation has an exponential solution for the number, as well as for the decay rate:

$$N(t) = N_0 e^{-t/\tau} \quad \frac{dN(t)}{dt} = -\frac{N_0}{\tau} e^{-t/\tau} = \frac{dN}{dt}(0) e^{-t/\tau} \quad (9.18)$$

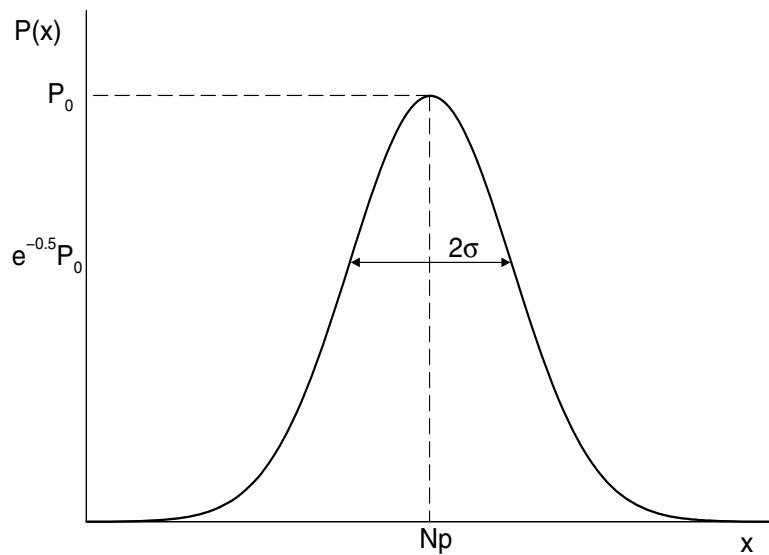
Equation (9.18) is the theoretical formula we wish to “fit” to the data in Fig. 9.2. The output of such a fit is a “best value” for the lifetime  $\tau$ .

### 9.3

#### Theory: Probability and Statistics

The field of statistics is an attempt to apply the mathematical theory of probability to describe natural events, such as coin flips, in which there is an element of chance or randomness. Some basic elements of probability upon which statistics is based are as follows.

1. The probability  $P(x)$  of an event  $x$  equals the fraction of times that the event occurs in a series of experiments.
2. Probability lies in the range  $0 \leq P(x) \leq 1$ , with 0 corresponding to an impossible event, and 1 to a sure thing.
3. If  $P(x)$  is the probability of an event occurring, then  $1 - P(x)$  is the probability of the event *not* occurring (the complement or opposite).
4. Given a population to be sampled, the probability of one sample having a particular value equals the fraction of the population having that value.



**Fig. 9.3** A Gaussian distribution of  $m$  successes in  $N$  trials, each with probability  $p$ .

5. The mean value  $\bar{f}$  of a function of  $x$  is given by the weighted sum

$$\overline{f(x)} \equiv \langle f(x) \rangle \stackrel{\text{def}}{=} \sum_{i=1}^{\infty} P(x_i) f(x_i) \quad (9.19)$$

The basic building block of statistics is the *binomial distribution* function

$$P_B(x) = \binom{N}{x} p^x (1-p)^{N-x} = \frac{N!}{(N-x)!x!} p^x (1-p)^{N-x} \quad (9.20)$$

This function gives the probability  $P_B(x)$  that an independent event (say heads) will occur  $x$  times in the  $N$  trials. Here  $p$  is the probability of occurrences of an individual event; for example, the probability of “heads” in any one toss is  $p = \frac{1}{2}$ . The variable  $N$  is the number of *trials* (measurements) in which that event can occur; for example, the number of times we flip the coin. For coin flipping, the probability of success  $p$  and the probability of failure  $(1-p)$  are both  $\frac{1}{2}$ , but in the general case  $p$  can be any number between 0 and 1.

As a matter of convenience, we eliminate one of the factorials in the binomial distribution (9.20) by considering the limit in which the number of measurements of trials  $N \rightarrow \infty$ . If, in addition, the probability  $p$  of an individual event (heads) remains finite as  $N \rightarrow \infty$ , we have **Gaussian** or **normal** statistics, and the probability function takes the simple form (Fig. 9.3)

$$P_G(x) = \lim_{N \rightarrow \infty, p \neq 0} P_B(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left[ -\frac{(x-\mu)^2}{2\sigma^2} \right] \quad (9.21)$$

Here  $\mu \equiv \bar{x}$  is the mean and  $\sigma$  is the variance:

$$\mu = Np \quad \sigma = \sqrt{Np(1-p)} \quad (9.22)$$

The Gaussian distribution is generally a very good approximation to the binomial distribution for  $N > 10$ , where is used to describe an experiment in which  $N$  measurements of the variable  $x$  are made. The average of these measurements is  $\mu$  and the “error” or uncertainty in  $\mu$  is  $\sigma$ . As an example, in  $N=1000$  coin flips, the probability of a head is  $p = \frac{1}{2}$  and the average number of heads  $\mu$  should be  $Np = N/2 = 500$ . As shown in Fig. 9.3, the Gaussian distribution has a width

$$\sigma = \sqrt{Np(1-p)} \propto \sqrt{N} \quad (9.23)$$

This means that the distribution actually gets wider and wider as more measurements are made. Yet the *relative width*, whose inverse gives us an indication of the probability of obtaining the average  $\mu$ , decreases with  $N$ :

$$\frac{\text{width}}{N} \propto \frac{\sqrt{N}}{N} = \frac{1}{\sqrt{N}} \rightarrow 0 \quad (N \rightarrow \infty) \quad (9.24)$$

Using a normal distribution to describe events that follow some other distribution function can lead to incorrect results. For example, another limit of the binomial distribution is the **Poisson** distribution. In the Poisson distribution, the number of trials  $N \rightarrow \infty$ , yet the probability of an individual success  $p \rightarrow 0$  in such a way that the product  $Np$  remains finite:

$$P_P(x) = \lim_{N \rightarrow \infty, p \rightarrow 0} P_B(x) = \frac{\mu^x e^{-\mu}}{x!} \quad (9.25)$$

A **Poisson** distribution describes radioactive decay experiments or telephone interchanges where there may be a very large number of trials (each microsecond when the counter is on), but a low probability of an event (a decay or phone call) occurring in this  $1 \mu s$ . As we see in Fig. 9.4, the Poisson distribution is quite asymmetric for small  $\mu$ , and in this way quite different from a Gaussian distribution. For  $\mu \gg 1$ , the Poisson distribution approaches a Gaussian distribution.

We use these distribution functions to determine probabilities as if events were continuous. For example, we take

$$P(x) dx = \text{probability that } x \text{ lies in the interval } x \leq x \leq x + dx. \quad (9.26)$$

If we then want to calculate the *mean* of some  $x$ , it would be

$$\bar{x} \stackrel{\text{def}}{=} \sum_{i=1}^{\infty} P(x_i)x_i = \int dx P(x)x \quad (9.27)$$

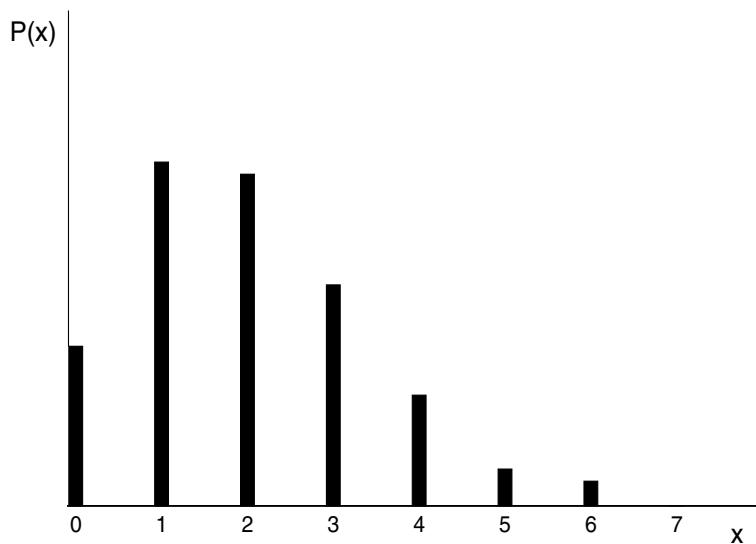


Fig. 9.4 A Poisson distribution for  $m$  successes with total success  $a = 2$ .

where the integral form would be used in those cases where we may approximate discrete values by a continuous distribution function. Likewise, the mean of an arbitrary function of  $x$  is

$$\overline{f(x)} \stackrel{\text{def}}{=} \sum_{i=1}^{\infty} P(x_i)f(x_i) = \int dx P(x)f(x) \quad (9.28)$$

When we apply these techniques to the normal distribution (9.21), we verify that  $\mu$  is the mean value of  $x$ , and that  $\sigma$  is the root of the mean-square deviation of  $x$  from its average:

$$\bar{x} = \int_{-\infty}^{+\infty} dx \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] x = \mu \quad (9.29)$$

$$\overline{x - \bar{x}} = \int_{-\infty}^{+\infty} dx \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] (x - \bar{x})^2 = \sigma^2 \quad (9.30)$$

#### 9.4

##### Least-Squares Fitting (Method)

Books have been written and careers have been spent discussing what is meant by a “good” fit to experimental data. We cannot do justice to the subject here and refer the reader to [9, 13, 14, 16]. However, we will emphasize two points:

1. If the data being fit contain errors, then the “best” fit in a statistical sense should not pass through all the data points.
2. Only for the simplest case of a linear, least-squares fit can we write down a closed-form solution to evaluate and obtain the fit. More realistic problems are usually solved by *trial-and-error* search procedures, sometimes using sophisticated subroutines libraries. However, in Section 9.4.5 we show how to conduct such a nonlinear search using familiar tools.

Imagine that you have measured  $N_D$  data values of the independent variable  $y$  as a function of the dependent variable  $x$ :

$$(x_i, y_i \pm \sigma_i) \quad i = 1, N_D \quad (9.31)$$

where  $\pm \sigma_i$  is the uncertainty in the  $i$ th value of  $y$ . (For simplicity we assume that all the errors  $\sigma_i$  occur in the dependent variable, although this is hardly ever true [14].) For our problem,  $y$  is the number of decays as a function of time, and  $x_i$  are the times. Our goal is to determine how well a mathematical function  $y = g(x)$  (also called *theory* or *model*) can describe these data. Alternatively, if the theory contains some parameters or constants, our goal can be viewed as determining best values for these parameters. We assume that the model function  $g(x)$  contains, in addition to the functional dependence on  $x$ , an additional dependence upon  $M_P$  parameters  $\{a_1, a_2, \dots, a_{M_P}\}$ . Notice that the parameters  $\{a_m\}$  are not variables, in the sense of numbers read from a meter, but rather are parts of the theoretical model such as the size of a box, the mass of a particle, or the depth of a potential well. For the exponential decay function (9.18), the parameters are the lifetime  $\tau$  and the initial decay rate  $dN(0)/dt$ . We indicate this as

$$g(x) = g(x; \{a_1, a_2, \dots, a_{M_P}\}) = g(x; \{a_m\}) \quad (9.32)$$

We use the chi-squared ( $\chi^2$ ) measure as a gauge of how well a theoretical function  $g$  reproduces data

$$\chi^2 \stackrel{\text{def}}{=} \sum_{i=1}^{N_D} \left( \frac{y_i - g(x_i; \{a_m\})}{\sigma_i} \right)^2 \quad (9.33)$$

where the sum is over the  $N_D$  experimental points  $(x_i, y_i \pm \sigma_i)$ . The definition (9.33) is such that smaller values of  $\chi^2$  are better fits, with  $\chi^2 = 0$  occurring if the theoretical curve went through the center of every data point. Notice also that the  $1/\sigma_i^2$  weighting means that measurements with larger errors<sup>3</sup> contribute less to  $\chi^2$ .

<sup>3</sup> If you are not given the errors, you can guess them on the basis of the apparent deviation of the data from a smooth curve, or you can weigh all points equally by setting  $\sigma_i \equiv 1$  and continue with the fitting.

*Least-squares fitting* refers to adjusting the theory until a minimum in  $\chi^2$  is found, that is, finding a curve that produces the least value for the summed squares of the deviations of the data from the function  $g(x)$ . In general, this is the best fit possible or the best way to determine the parameters in a theory. The  $M_P$  parameters  $\{a_m, m = 1, M_P\}$  that make  $\chi^2$  an extremum are found by solving the  $M_P$  equations:

$$\frac{\partial \chi^2}{\partial a_m} = 0 \quad \Rightarrow \quad \sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_m} = 0 \quad (m = 1, M_P) \quad (9.34)$$

More usually, the function  $g(x; \{a_m\})$  has a sufficiently complicated dependence on the  $a_m$  values for (9.34) to produce  $M_P$  simultaneous, nonlinear equations in the  $a_m$  values. In these cases, solutions are found by a trial-and-error search through the  $M_P$ -dimensional parameter space, as we do in Section 9.4.5. To be safe, when such a search is completed you need to check that the minimum  $\chi^2$  you found is *global* and not *local*. One way to do that is to repeat the search for a whole grid of starting values, and if different minima are found, to pick the one with the lowest  $\chi^2$ .

#### 9.4.1

##### Goodness of Fit (Theory)

When the deviations from theory are due to random errors and when these errors are described by a Gaussian distribution, there are some useful rules of thumb to remember [13]. You know that your fit is good if the value of  $\chi^2$  calculated via the definition (9.33) is approximately equal to the number of degrees of freedom  $\chi^2 \approx N_D - M_P$ , where  $N_D$  is the number of data points and  $M_P$  the number of parameters in the theoretical function. If your  $\chi^2$  is much less than  $N_D - M_P$ , it does not mean that you have a “great” theory or a really precise measurement; instead, you probably have too many parameters or have assigned errors ( $\sigma_i$  values) that are too large. In fact, too small a  $\chi^2$  may indicate that you are fitting the random scatter in the data rather than missing  $\sim \frac{1}{3}$  of the error bars, as expected for Gaussian statistics. If your  $\chi^2$  is significantly greater than  $N_D - M_P$ , the theory may not be good, you may have significantly underestimated your errors, or you may have errors which are not random.

#### 9.4.2

##### Least-Squares Fits Implementation

The  $M_P$  simultaneous equations (9.34) simplify considerably if the functions  $g(x; \{a_m\})$  depend *linearly* on the parameter values  $a_i$ :

$$g(x; \{a_1, a_2\}) = a_1 + a_2 x \quad (9.35)$$

In this case (also known as *linear regression*) there are  $M_P = 2$  parameters, the slope  $a_2$  and the  $y$  intercept  $a_1$ . Notice that while there are only two parameters to determine, there still may be an arbitrary number  $N_D$  of data points to fit. Remember that a unique solution is not possible unless the number of data points is equal to or greater than the number of parameters.

For this linear case, there are just two derivatives,

$$\frac{\partial g(x_i)}{\partial a_1} = 1 \quad \quad \frac{\partial g(x_i)}{\partial a_2} = x_i \quad (9.36)$$

and after substitution, the  $\chi^2$  minimization equations (9.34) can be solved [9]:

$$a_1 = \frac{S_{xx}S_y - S_xS_{xy}}{\Delta} \quad a_2 = \frac{SS_{xy} - S_xS_y}{\Delta} \quad (9.37)$$

$$S = \sum_{i=1}^{N_D} \frac{1}{\sigma_i^2} \quad S_x = \sum_{i=1}^{N_D} \frac{x_i}{\sigma_i^2} \quad S_y = \sum_{i=1}^{N_D} \frac{y_i}{\sigma_i^2} \quad (9.38)$$

$$S_{xx} = \sum_{i=1}^{N_D} \frac{x_i^2}{\sigma_i^2} \quad S_{xy} = \sum_{i=1}^{N_D} \frac{x_i y_i}{\sigma_i^2} \quad \Delta = SS_{xx} - S_x^2 \quad (9.39)$$

Statistics also gives you an expression for the *variance* or uncertainty in the deduced parameters:

$$\sigma_{a_1}^2 = \frac{S_{xx}}{\Lambda} \quad \sigma_{a_2}^2 = \frac{S}{\Lambda} \quad (9.40)$$

This is a measure of the uncertainties in the values of the fitted parameters arising from the uncertainties  $\sigma_i$  in the measured  $y_i$  values. A measure of the dependence of the parameters on each other is given by the *correlation coefficient*:

$$\rho(a_1, a_2) = \frac{\text{cov}(a_1, a_2)}{\sigma_{a_1} \sigma_{a_2}} \quad \text{cov}(a_1, a_2) = \frac{-S_x}{\Delta} \quad (9.41)$$

Here  $\text{cov}(a_1, a_2)$  is the covariance of  $a_1$  and  $a_2$  and vanishes if  $a_1$  and  $a_2$  are independent. The correlation coefficient  $\rho(a_1, a_2)$  lies in the range  $-1 \leq \rho \leq 1$ . Positive  $\rho$  indicates that the errors in  $a_1$  and  $a_2$  are likely to have the same sign; negative  $\rho$  indicates opposite signs.

**Listing 9.3:** `fit.java` makes a linear least-squares-fit to some tabulated data.

```
// Fit.java: Least-squares fit to tabulated data
```

```
public class fit {  
    static int data = 12; // Number of data points
```

```

public static void main(String[] argv) {

    int i, j;
    double s, sx, sy, sxx, sxy, delta, inter, slope;
    double x[] = new double[data]; double y[] = new double[data];
    double d[] = new double[data];
                                // Input data x
    for (i = 0; i<data; i++) x[i] = i*180+5;
                                // Input data y
    y[0] = 382; y[1] = 187; y[2] = 281; y[3] = 78;
    y[4] = 88; y[5] = 86; y[6] = 85; y[7] = 28;
    y[8] = 28; y[9] = 0.81; y[10] = 84; y[11] = 81;
                                // Input data delta y
    for (i = 0; i<data; i++) d[i] = 18.;
    s = sx = sy = sxx = sxy = 0.;                                // Init sums
                                // Calculate sums
    for (i = 0;i<data;i++) {
        s   +=      1. / (d[i]*d[i]);
        sx  +=      x[i] / (d[i]*d[i]);
        sy  +=      y[i] / (d[i]*d[i]);
        sxx +=  x[i]*x[i] / (d[i]*d[i]);
        sxy +=  x[i]*y[i] / (d[i]*d[i]);
    }
    delta  =  s*sxx - sx*sx;
    slope  =  (s*sxy - sx*sy) / delta;
    inter  =  (sxx*sy - sx*sxy) / delta;
    System.out.println("intercpt= "+inter+", "+Math.sqrt(sxx/delta));
    System.out.println("slope = "+slope+", "+Math.sqrt(s/delta));
    System.out.println("Fit Program Complete.");
}
}

```

Our program to perform a linear least-square fit is given in Listing 9.3. In it we realize that the preceding analytic solutions for the parameters are of the form found in statistics books, but are not optimal for numerical calculations because subtractive cancellation can make the answers unstable. As discussed in Chap. 3, a rearrangement of the equations can decrease this type of error. For example, [14] gives improved expressions that measure the data relative to their averages:

$$\begin{aligned}
a_1 &= \bar{y} - a_2 \bar{x} & a_2 &= \frac{S_{xy}}{S_{xx}} & \bar{x} &= \frac{1}{N} \sum_{i=1}^{N_d} x_i & \bar{y} &= \frac{1}{N} \sum_{i=1}^{N_d} y_i \\
S_{xy} &= \sum_{i=1}^{N_d} \frac{(x_i - \bar{x})(y_i - \bar{y})}{\sigma_i^2} & S_{xx} &= \sum_{i=1}^{N_d} \frac{(x_i - \bar{x})^2}{\sigma_i^2}
\end{aligned} \tag{9.42}$$

### 9.4.3

#### Exponential Decay Fit Assessment

Fit the exponential decay law (9.18) to the data in Fig. 9.2. This means finding values for  $\tau$  and  $\Delta N(0)/\Delta t$  that provides a best fit to the data, and then judging how good the fit is.

1. Construct a table ( $\Delta N / \Delta t_i, t_i$ ), for  $i = 1, N_D$  from Fig. 9.2. Because time was measured in bins,  $t_i$  should correspond to the middle of a bin.
2. Add an estimate of the error  $\sigma_i$  to obtain a table of the form  $(\Delta N / \Delta t_i \pm \sigma_i, t_i)$ . You can estimate the errors by eye, say, by estimating how much the histogram values appear to fluctuate about a smooth curve, or you can take  $\sigma_i \simeq \sqrt{\text{Events}}$ . (This last approximation is reasonable for large numbers, which this is not.)
3. In the limit of very large numbers, we would expect that a plot of  $\ln |\Delta N / \Delta t|$  versus  $t$  is a straight line:

$$\ln \left| \frac{\Delta N(t)}{\Delta t} \right| \simeq \ln \left| \frac{\Delta N_0}{\Delta t} \right| - \frac{1}{\tau} \Delta t.$$

This means that if we treat  $\ln |\Delta N(t) / \Delta t|$  as the dependent variable and time  $\Delta t$  as the independent variable, we can use our linear fit results. Plot  $\ln |\Delta N / \Delta t|$  versus  $\Delta t$ .

4. Make a least-squares fit of a straight line to your data and use it to determine the lifetime  $\tau$  of the  $\pi$  meson. Compare your deduction to the tabulated lifetime of  $2.6 \times 10^{-8}$  s and comment on the difference.
5. Plot your best fit on the same graph as the data and comment on the agreement.
6. Deduce the goodness of fit of your straight line and the approximate error in your deduced lifetime. Do these agree with what your “eye” tells you?

#### 9.4.4

##### **Exercise: Fitting Heat Flow**

The table below gives the temperature  $T$  along a metal rod whose ends are kept at fixed constant temperatures. The temperature is a function of the distance  $x$  along the rod.

$x_i$ (cm)	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0
$T_i$ (°C)	14.6	18.5	36.6	30.8	59.2	60.1	62.2	79.4	99.9

1. Plot up the data to verify the appropriateness of a linear relation

$$T(x) \simeq a + bx \quad (9.43)$$

2. Because you are not given the errors for each measurement, assume that the least-significant figure has been rounded off and so  $\sigma \geq 0.05$ . Use that to compute a least-squares, straight-line fit to these data.
3. Plot your best  $a + bx$  on the curve with the data.
4. After fitting the data, compute the variance and compare it to the deviation of your fit from the data. Verify that about one-third of the points miss the  $\sigma$  error band (that is what is expected for a normal distribution of errors).
5. Use your computed variance to determine the  $\chi^2$  of the fit. Comment on the value obtained.
6. Determine the variances  $\sigma_a$  and  $\sigma_b$  and check if it makes sense to use them as the errors in the deduced values for  $a$  and  $b$ .

#### 9.4.5

##### Nonlinear Fit of Breit–Wigner to Cross Section

**Problem:** Recall how we started Unit III of this chapter by interpolating the values in Tab. 9.1 giving the experimental cross section  $\Sigma$  as a function of energy. Although we did not use it, we also gave the theory describing these data, namely, the Breit–Wigner resonance formula (9.1):

$$f(E) = \frac{f_r}{(E - E_r)^2 + \Gamma^2/4} \quad (9.44)$$

Your **problem** is to determine what values for the parameters  $E_r$ ,  $f_r$ , and  $\Gamma$  in (9.44) provide the best fit to the data in Tab. 9.1.

Because (9.44) is not a linear function of the parameters  $(E_r, \Sigma_0, \Gamma)$ , the three equations that result from minimizing  $\chi^2$  are not linear equations and so cannot be solved by the techniques of *linear* algebra (matrix methods). However, in our study of the weights on a string problem in Unit I, we showed how to use the Newton–Raphson algorithm to search for solutions of simultaneous nonlinear equations. That technique involved expansion of the equations about the previous guess to obtain a set of linear equations, and then solving the linear equations with the matrix libraries discussed in Unit II. We now use this same combination of fitting, trial-and-error searching and matrix algebra, to conduct a nonlinear least-squares fit of (9.44) to the data in Tab. 9.1.

Recall that the condition for a best fit is to find values of the  $M_P$  parameters  $a_m$  in the theory  $g(x, a_m)$  that minimize  $\chi^2 = \sum_i [(y_i - g_i)/\sigma_i]^2$ . This leads to the  $M_P$  equations (9.34) to solve:

$$\sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_m} = 0 \quad (m = 1, M_P) \quad (9.45)$$

To find the form of these equations appropriate to our problem, we rewrite our theory function (9.44) in the notation of (9.45). We let

$$a_1 = f_r \quad a_2 = E_R \quad a_3 = \Gamma^2/4 \quad x = E \quad (9.46)$$

$$\Rightarrow \quad g(x) = \frac{a_1}{(x - a_2)^2 + a_3} \quad (9.47)$$

The three derivatives required in (9.45) are then

$$\frac{\partial g}{\partial a_1} = \frac{1}{(x - a_2)^2 + a_3} \quad \frac{\partial g}{\partial a_2} = \frac{-2a_1(x - a_2)}{[(x - a_2)^2 + a_3]^2} \quad \frac{\partial g}{\partial a_3} = \frac{-a_1}{[(x - a_2)^2 + a_3]^2}$$

Substitution of these derivatives into the best fit condition (9.45) yields the three simultaneous equations in  $a_1$ ,  $a_2$ , and  $a_3$  that we will need to solve in order to fit the  $N_D = 9$  data points  $(x_i, y_i)$  in Tab. 9.1:

$$\begin{aligned} \sum_{i=1}^9 \frac{y_i - g(x_i, a)}{(x_i - a_2)^2 + a_3} &= 0 & \sum_{i=1}^9 \frac{y_i - g(x_i, a)}{[(x_i - a_2)^2 + a_3]^2} &= 0, \\ \sum_{i=1}^9 \frac{\{y_i - g(x_i, a)\} (x_i - a_2)}{[(x_i - a_2)^2 + a_3]^2} &= 0 \end{aligned} \quad (9.48)$$

Even without the substitution of (9.44) for  $g(x, a)$ , it is clear that these three equations depend on the  $a$ 's in nonlinear fashions; this is what makes them hard to solve. However, we already know how to solve them! If you look back at Section 8.1.2, you will see that we derived there the method for conducting an  $N$ -dimensional Newton–Raphson search for the roots of

$$f_i(a_1, a_2, \dots, a_N) = 0 \quad i = 1, N \quad (9.49)$$

where we have made the change of variable  $y_i \rightarrow a_i$  for the present problem. We use that same formalism for the  $N = 3$  Eqs. (9.48) by writing them as

$$f_1(a_1, a_2, a_3) = \sum_{i=1}^9 \frac{y_i - g(x_i, a)}{(x_i - a_2)^2 + a_3} = 0 \quad (9.50)$$

$$f_2(a_1, a_2, a_3) = \sum_{i=1}^9 \frac{\{y_i - g(x_i, a)\} (x_i - a_2)}{[(x_i - a_2)^2 + a_3]^2} = 0 \quad (9.51)$$

$$f_3(a_1, a_2, a_3) = \sum_{i=1}^9 \frac{y_i - g(x_i, a)}{[(x_i - a_2)^2 + a_3]^2} = 0 \quad (9.52)$$

Because  $f_r \equiv a_1$  is the peak value of the cross section,  $E_R \equiv a_2$  is the energy at which the peak occurs, and  $\Gamma = 2\sqrt{a_3}$  is the full width of the peak at half maximum, good starting guesses for the  $a$ 's can be extracted from a graph

of the data. All that is needed then to apply the Newton–Raphson matrix equations (8.16) to our problem are expressions for the derivatives of the three  $f$ 's with respect to the three unknown  $a$ 's. As before, it is easy to evaluate all nine derivatives by use of two nested loops over  $i$  and  $j$ , along with the forward-difference approximation for the derivative,

$$\frac{\partial f_i}{\partial a_j} \simeq \frac{f_i(a_j + \Delta a_j) - f_i(a_j)}{\Delta a_j} \quad (9.53)$$

where  $\Delta a_j$  corresponds to a small, say  $\leq 1\%$ , change in the parameter value.

#### 9.4.5.1 Nonlinear Fit Implementation

Use the Newton–Raphson algorithm as outlined in Section 9.4.5 to conduct a nonlinear search for the best fit parameters of the Breit–Wigner theory (9.44) to the data in Tab. 9.1. Compare the deduced values of  $(f_r, E_R, \Gamma)$  to that obtained by inspection of the graph. The program `Newton_Jama.java` on the instructor's CD solves this problem.

## 9.5

### Appendix: Calling LAPACK from C

Calling a Fortran-based matrix library from Fortran is less trouble than calling it from some other language. However, if you refuse to be a Fortran programmer, there are often C and Java implementations of the Fortran libraries available, such as JAMA, JLAPACK, LAPACK++, and TNT for use from these languages.

Some of our research projects have required us to have Fortran and programs calling each other, and, after some experimentation, we have had success doing it under Unix. But care is needed in accounting for the somewhat different ways the compilers store subroutine names, for the quite different ways they store arrays with more than one subscript, and for the different data types available in the two languages.

**Tab. 9.2** Matching data types in C and Fortran.

C	Fortran	C	Fortran
char	Character	unsigned int	Logical*4
signed char	Integer*1	float	Real (Real*4)
unsigned char	Logical*1	structure of 2 floats	Complex
short signed int	Integer*2	double	Real*8
short unsigned int	Logical*2	structure of 2 doubles	Complex*16
signed int (long int)	Integer*4	char[n]	Character*n

The first thing you must do is ensure that the data types of the variables in the two languages are matched. The matching data types are given in Tab. 9.2.

Note that if the data are stored in arrays, your C calling program must convert to the storage scheme used in the Fortran subroutine before you can call the Fortran subroutine (and then convert back to the C scheme after the subroutine does its job if you have overwritten the original array and intend to use it again).

When a function is called in the C language, usually the actual value of the argument is passed to the function. In contrast, Fortran passes the address in memory where the value of the argument is to be found (a *reference pass*). If you do not ensure that both languages have the same passing protocol, your program will process the numerical value of the address of a variable as if it were the actual value of the variable (we are willing to place a bet on the correctness of the result). Here are some procedures for **calling Fortran from C**:

1. Use pointers for all arguments in your C programs. Generally this is done with the address operator &.
2. Do not have your program make calls such as `sub(1, N)` where the actual value of the constant “1” is fed to the subroutine. Instead, assign the value one to a variable, and feed that variable (actually a pointer to it) to the subroutine. For example:

```
one = 1.                                // Assign value to variable
sub(one)          // All Fortran calls are reference calls
sub(&one)        // In C, make pointer explicit
```

This is important because the value “1” in the subroutine call, in Fortran, anyway, is actually the address where the value “1” is stored. If the subroutine modifies that variable, it will modify the value of “1” at every place in your program! In C, it would change the first value in memory.

3. Depending on the particular operating system you are using, you may have to append an underscore \_ to the called Fortran subprogram names. As a case in point, `sub(one, N) → sub_(one, N)`. Generally, the Fortran compiler appends an underscore automatically to the names of its subprograms, while the C compiler does not (but you will need to experiment).
4. Use lowercase letters for the names of external functions. The exception is when the Fortran subprogram being called was compiled with a -U option, or the equivalent, for retaining uppercase letters.

## 9.5.1

**Calling LAPACK Fortran from C**

```
// Calling LAPACK from C to solve AX=B

#include <stdio.h>                                // I/O headers
#define size 100                                     // Dimension of Hilbert matrix

main() {
    int i, j, c1, c2, pivot[size], ok;
    double matrix[size][size], help[size*size], result[size];
                                            // Pointers for function call
    c1 = size;
    c2 = 1;                                         // Numbers as variables
    for (i = 0; i < c1; i++) {                      // Create Hilbert matrix
        for (j = 0; j < c1; j++) matrix[i][j] = 1.0/(i+j+1);
        result[i] = 1. / (i+1);                      // Create solution vector
    }
                                            // Transform matrix
    for (i = 0; i < size; i++)
        for(j = 0; j < size; j++) help[j+size*i] = matrix[j][i];

    dgesv_(&c1, &c2, help, &c1, pivot, result, &c1, &ok);
    for (j = 0; j<size; j++) printf("%e\n", result[j]);
}
```

You will notice here that the call to the Fortran subroutine `dgesv` is made as

```
dgesv_(&c1, &c2, help, &c1, pivot, result, &c1, &ok)
```

That is, lowercase letters are used and an underscore is added to the subroutine name. In addition, we convert the matrix  $A$ ,

```
for (i=0; i < size; i++)                         // Matrix transformation
    for (j=0; j < size; j++) help[j+size*i] = matrix[j][i];
```

This changes C's row-major order to Fortran's column-major order using a scratch vector.

## 9.5.2

**Compiling C Programs with Fortran Calls**

Multilanguage programs actually get created when the compiler links the object files together. The tricky part is that while Fortran automatically includes its math library, if your final linking is done with the C compiler, you may have to explicitly include the Fortran library as well as others. Here we give some examples that have worked at one time or another, on one machine or another; you probably will need to read the User's Guide and experiment to get this to work with your system:

```
> cc -O call.c f77sub.o -lm -lxml
```

DEC extended mathlib

```
> cc -O call.c f77sub.o -L/usr/lang/SC0.0 -lf77 Link, SunOS
> cc -O c_fort c_fort.o area_f.o -lxlf Link, AIX
> gcc -Wall call.c /usr/lib/libm.a -o call gcc explicit
> gcc -Wall call.c -lm -o call gcc shorthand version of above
```

## 10

### Deterministic Randomness

*Some people are attracted to computing by its deterministic nature; it is nice to have something in life where nothing is left to chance. Barring random machine errors or undefined variables, you must get the same output every time you feed your program the same input. Nevertheless, many computer cycles are used for Monte Carlo calculations that at their very core strive to include some elements of chance. These are calculations in which random numbers generated by the computer are used to simulate naturally random processes, such as thermal motion or radioactive decay, or to solve equations on the average. Indeed, much of the recognition of computational physics as a specialty has come about from the ability of computers to solve previously intractable thermodynamic and quantum mechanics problems using Monte Carlo techniques.*

**Problem:** Your **problem** in this chapter is to explore how computers can generate random numbers and how well they can do it. To check whether it really works, in Chap. 11 you *simulate* some random walks and spontaneous decays, and evaluate some multidimensional integrals. Other applications, such as thermodynamics and lattice quantum mechanics, are considered in later chapters.

#### 10.1

##### Random Sequences (Theory)

We define a sequence of numbers  $r_1, r_2, \dots$  as *random* if there are no correlations among the numbers in the sequence. Yet randomness does not necessarily mean all numbers in the sequence are equally likely to occur. If all numbers in a sequence are equally likely to occur, then the sequence is *uniform*. To illustrate,  $1, 2, 3, 4, \dots$  is uniform but probably not random, while  $3, 1, 4, 2, 3, 1, 3, 2, 4, \dots$  may be random but does not appear to be uniform. In addition, it is possible to have a sequence of numbers that, in some sense, are random but have very short range correlations, for example,  $r_1(1 - r_1)r_2(1 - r_2)r_3(1 - r_3) \dots$ .

Mathematically, the likelihood of a random number occurring is described by a distribution function  $P(r)$ . This means the probability of finding  $r_i$  in the interval  $[r, r + dr]$  is  $P(r)dr$ . A *uniform* distribution means that  $P(r) =$

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5

constant. The standard random-number generator on computers generates uniform distributions ( $P = 1$ ) between 0 and 1. In other words, the standard random-number generator outputs numbers in this interval, each with an equal probability yet each independent of the previous number. As we shall see, numbers can also be generated nonuniformly and still be random.

By their very construction we know computers are deterministic and so they cannot truly create a random sequence. Although it may be a bit of work, if we know  $r_m$  and its preceding elements, it is always possible to figure out  $r_{m+1}$ . For this reason, computers generate “*pseudo*-random numbers”. By the very nature of their creation, computed random numbers must contain correlations and in this way are not truly random. (Yet with our incurable laziness we would not bother saying “*pseudo*” all the time.) While the more sophisticated generators do a better job at hiding the correlations, experience shows that if you look hard enough, or use these numbers enough you will notice correlations. A primitive alternative to generating random numbers is to read in a table of true random numbers, that is, numbers determined by naturally random processes such as radioactive decay. While not an attractive way to spend one’s time, it may provide a valuable comparison.

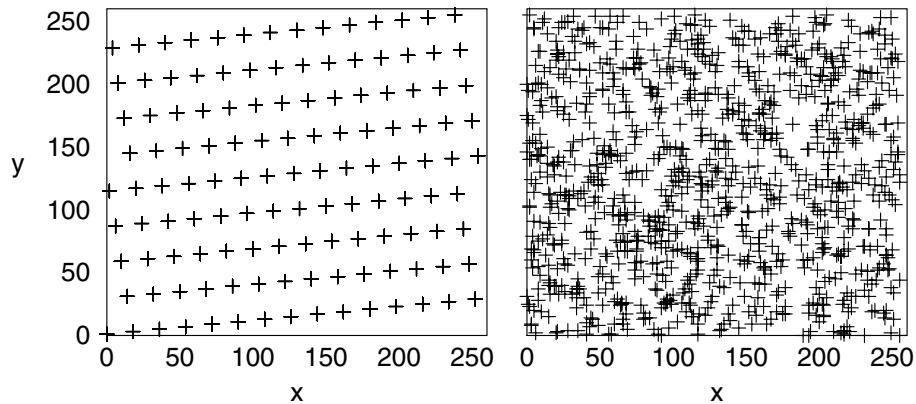
### 10.1.1 Random-Number Generation (Algorithm)

The *linear congruent* or *power residue* method is the most common way of generating a pseudo-random sequence of numbers  $\{r_1, r_2, \dots, r_k\}$  over the interval  $[0, M - 1]$ . You multiply the previous random number  $r_{i-1}$  by the constant  $a$ , add on another constant  $c$ , take the *modulus* by  $M$ , and then keep just the fractional part (remainder)<sup>1</sup> as the next random number  $r_i$ :

$$r_i \stackrel{\text{def}}{=} (a r_{i-1} + c) \bmod M = \text{remainder}\left(\frac{a r_{i-1} + c}{M}\right) \quad (10.1)$$

The value for  $r_1$  (the *seed*) is frequently supplied by the user, and *mod* is a built-in function on your computer for *remaindering* (it may be called *amod* or *dmod*). This is essentially a bit-shift operation that ends up with the least-significant part of the input number, and thus counts on the randomness of roundoff errors to produce a random sequence.

<sup>1</sup> You may obtain the same result for the modulus operation by subtracting  $M$  until any further subtractions would leave a negative number; what remains is the *remainder*.



**Fig. 10.1** Left: A plot of successive random numbers  $(x, y) = (r_i, r_{i+1})$ , generated with a deliberately “bad” generator. Right: a plot with the library routine `drand48`.

As an example, if  $c = 1, a = 4, M = 9$ , and you supply  $r_1 = 3$ , then you obtain the sequence

$$r_1 = 3, \quad (10.2)$$

$$r_2 = (4 \times 3 + 1) \bmod 9 = 13 \bmod 9 = \text{rem } \frac{13}{9} = 4 \quad (10.3)$$

$$r_3 = (4 \times 4 + 1) \bmod 9 = 17 \bmod 9 = \text{rem } \frac{17}{9} = 8 \quad (10.4)$$

$$r_4 = (4 \times 8 + 1) \bmod 9 = 33 \bmod 9 = \text{rem } \frac{33}{9} = 6 \quad (10.5)$$

$$r_{5-10} = 7, 2, 0, 1, 5, 3 \quad (10.6)$$

We get a sequence of length  $M = 9$ , after which the entire sequence repeats. If we want numbers in the range  $[0, 1]$ , we divide the  $r$ 's by  $M = 9$

$$0.333, 0.444, 0.889, 0.667, 0.778, 0.222, 0.000, 0.111, 0.555, 0.333.$$

This is still a sequence of length 9, but is no longer one of integers. As a general operating procedure

*Before using a random-number generator in your programs, you may check its range and that it is producing numbers that “look” random.*

Although this is not a strict mathematical test, your visual cortex is quite refined at recognizing patterns, and, in any case, it is easy to perform. For instance, Fig. 10.1 shows results from “good” and “bad” generators; it is really quite easy to tell them apart.

The rule (10.1) produces integers in the range  $[0, M - 1]$ , but not necessarily every integer. When a particular integer comes up a second time, the whole cycle repeats. In order to obtain a longer sequence,  $a$  and  $M$  should be large numbers, but not so large that  $ar_{i-1}$  overflows. On a scientific computer using 48-bit integer arithmetic, the built-in random-number generator

may use  $M$  values as large as  $2^{48} \simeq 3 \times 10^{14}$ . A 32-bit machine may use  $M = 2^{31} \simeq 2 \times 10^9$ . If your program uses approximately this many random numbers, you may need to reseed the sequence during intermediate steps to avoid repetitions.

Your computer probably has random-number generators that are better than the one you will compute with the power residue method. You may check this out in the manual or help pages (try the *man* command in Unix) and then test the generated sequence. These routines may have names like *rand*, *rn*, *random*, *srand*, *erand*, *drand*, or *drand48*.

We recommend a version of *drand48* as a random-number generator. It generates random numbers in the range  $[0, 1]$  with good spectral properties by using 48-bit integer arithmetic with the parameters<sup>2</sup>

$$M = 2^{48} \quad c = B \text{ (base 16)} = 13 \text{ (base 8)}, \quad (10.7)$$

$$a = 5DEECE66D \text{ (base 16)} = 273673163155 \text{ (base 8)}. \quad (10.8)$$

To initialize the random sequence you need to plant a seed in it. In Fortran you would call the subroutine *srand48* to plant your seed, while in Java you issue the statement `Random randnum = new Random(seed);` (see `RandNum.java` for details).

Definition (10.1) will generate  $r_i$  values in the range  $[0, M]$  or  $[0, 1]$  if you divide by  $M$ . If random numbers in the range  $[A, B]$  are needed, you need to only **scale**; for example

$$x_i = A + (B - A)r_i \quad 0 \leq r_i \leq 1 \quad \Rightarrow \quad A \leq x_i \leq B \quad (10.9)$$

### 10.1.2

#### Implementation: Random Sequence

For scientific work we recommend using an industrial-strength random-number generator. To see why, here we assess how *bad* a careless application of the power residue method can be.

1. Write a simple program to generate random numbers using the linear congruent method (10.1).
2. For pedagogical purposes, try the unwise choice:  $(a, c, M, r_1) = (57, 1, 256, 10)$ . Determine the *period*, that is, how many numbers are generated before the sequence repeats.
3. Take the pedagogical sequence of random numbers and look for correlations by observing clustering on a plot of successive pairs  $(x_i, y_i) =$

<sup>2</sup> Unless you know how to do 48-bit arithmetic and how to input numbers in different bases, we do not recommend that you try these numbers yourself. For pedagogical purposes, large numbers such as  $M = 112,233$  and  $a = 9999$  work well.

$(r_{2i-1}, r_{2i})$ ,  $i = 1, 2, \dots$  (Do *not* connect the points with lines.) You may “see” correlations (Fig. 10.1), which means that you should not use this sequence for serious work.

4. Test the built-in random-number generator on your computer for correlations by plotting the same pairs as above. (This should be good for serious work.)
5. Test the linear congruent method again with reasonable constants like those in (10.7)–(10.8). Compare the scatterplot you obtain with that of the built-in random-number generator. (This, too, should be good for serious work.)

**Listing 10.1:** RandNum.java calls the random-number generator from the Java utility class. Note that a different seed is needed for a different sequence.

```
// RandNum.java: random numbers via java.util.Random.class

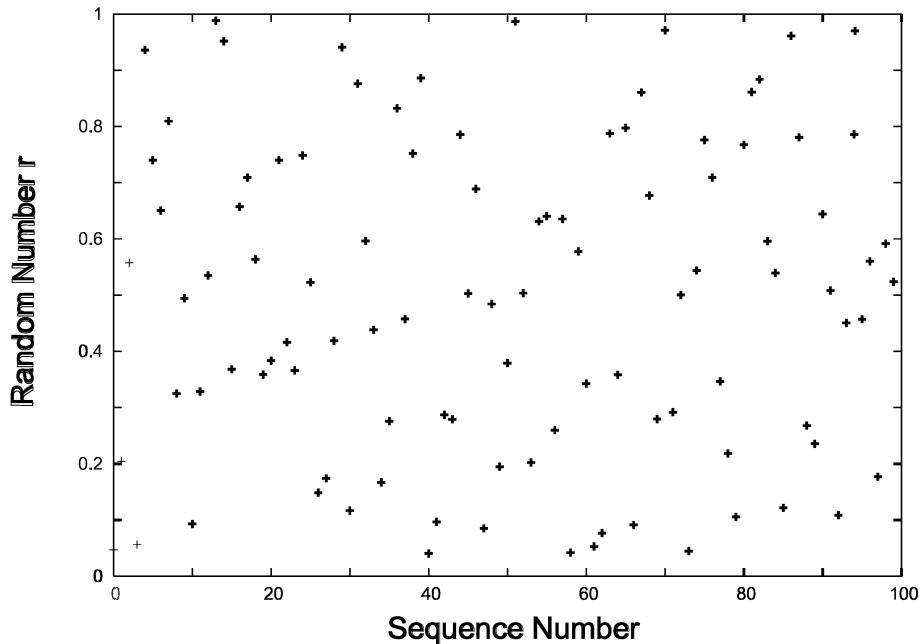
import java.io.*;
import java.util.*; // Location of PrintWriter
// Location of Random

public class RandNum {
    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
        PrintWriter q = new PrintWriter(
            new FileOutputStream("RandNum.DAT"), true);
        // Initialize 48 bit generator
        long seed = 899432;
        Random randnum = new Random(seed);
        int imax = 100, i = 0;

        // generate random numbers and store in data file
        for (i=1; i <= imax; i++) q.println(randnum.nextDouble());
        System.out.println(" ");
        System.out.println("RandNum Program Complete.");
        System.out.println("Data stored in RandNum.DAT");
        System.out.println(" ");
    }
} // End of class
```

### 10.1.3 Assessing Randomness and Uniformity

Because the computer’s random numbers are generated according to a definite rule, the numbers in the sequence must be correlated to each other. This can affect a simulation that assumes random events. Therefore, it is wise for you to test a random-number generator before you stake your scientific rep-



**Fig. 10.2** A plot of a uniform, pseudo-random sequence  $r_i$  versus  $i$ .

utation on results obtained with it. In fact, some tests are simple enough that you may make it a habit to run them simultaneously with your simulation.

In the examples to follow, we test for either randomness or uniformity.

1. Probably the most obvious, but often neglected, test for randomness and uniformity is to look at the numbers generated. For example, Tab. 10.1 is some output from `RandNum.java`. If you just look at these numbers, you know immediately that they all lie between 0 and 1, that they appear to differ from each other, and that there is no obvious pattern (like 0.3333).
2. If you now take this same list and plot it, the ordinate will be  $r_i$  and the abscissa, even though we do not state it, will be  $i$  (Fig. 10.2). Observe how there appears to be a uniform distribution between 0 and 1 and no particular correlation between points (although your eye and brain will connect the points and create some types of figures).
3. One simple test of uniformity evaluates the  $k$ th moment of the random-number distribution:

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^N x_i^k \quad (10.10)$$

**Tab. 10.1** A table of a uniform, pseudo-random sequence  $r_i$  generated by RandNum.java.

0.04689502438508175	0.20458779675039795	0.5571907470797255	0.05634336673593088
0.9360668645897467	0.7399399139194867	0.6504153029899553	0.8096333704183057
0.3251217462543319	0.49447037101884717	0.09307712613141128	0.32858127644188206
0.5351001685588616	0.9880354395691023	0.9518097953073953	0.36810077925659423
0.6572443815038911	0.7090768515455671	0.5636787474592884	0.3586277378006649
0.38336910654033807	0.7400223756022649	0.4162083381184535	0.3658031553038087
0.7484798900468111	0.522694331447043	0.14865628292663913	0.1741881539527136
0.41872631012020123	0.9410026890120488	0.1167044926271289	0.8759009012786472
0.5962535409033703	0.4382385414974941	0.166837081276193	0.27572940246034305
0.832243048236776	0.45757242791790875	0.7520281492540815	0.8861881031774513
0.04040867417284555	0.09690149294881334	0.2869627609844023	0.27915054491588953
0.7854419848382436	0.502978394047627	0.688866810791863	0.08510414855949322
0.48437643825285326	0.19479360033700366	0.3791230234714642	0.9867371389465821

If the random numbers are distributed with a *uniform* probability distribution  $P(x)$ , then (10.10) is approximately the moment of  $P(x)$ :

$$\frac{1}{N} \sum_{i=1}^N x_i^k \simeq \int_0^1 dx x^k P(x) + O(1/\sqrt{N}) \simeq \frac{1}{k+1} \quad (10.11)$$

If (10.11) holds for your generator, then you know that the distribution is uniform. If the deviation from (10.11) varies as  $1/\sqrt{N}$ , then you *also* know that the distribution is random.

- Another simple test determines the near-neighbor correlation in your random sequence by taking sums of products for small  $k$ :

$$C(k) = \frac{1}{N} \sum_{i=1}^N x_i x_{i+k} \quad (k = 1, 2, \dots) \quad (10.12)$$

If your random numbers  $x_i$  and  $x_{i+k}$  are distributed with the joint probability distribution  $P(x_i, x_{i+k})$  and are independent and uniform, then (10.12) can be approximated as an integral:

$$\frac{1}{N} \sum_{i=1}^N x_i x_{i+k} \simeq \int_0^1 dx \int_0^1 dy xy P(x, y) = \frac{1}{4} \quad (10.13)$$

If (10.13) holds for your random numbers, then you know that they are not correlated. If the deviation from (10.13) varies as  $1/\sqrt{N}$ , then you *also* know that the distribution is random.

- An effective test for randomness is performed visually by making a scatterplot of  $(x_i = r_{2i}, y_i = r_{2i+1})$  for many  $i$  values. If your points have noticeable regularity, the sequence is not random. If the points are random, they should uniformly fill a square with no discernible pattern (a cloud) (Fig. 10.1).

6. Another test is to run your calculation or simulation with the sequence  $r_1, r_2, r_3, \dots$ , and then again with the sequence  $(1 - r_1), (1 - r_2), (1 - r_3), \dots$ . Because both sequences should be random, if your results differ beyond statistics, then your sequence is probably not random.
7. Yet another test is to run your simulation with a sequence of true random numbers from a table and compare it to the results with the pseudo-random-number generator. In order to be practical, you may need to reduce the number of trials being made.
8. Test your random-number generator with (10.11) for  $k = 1, 3, 7$ , and  $N = 100, 10,000, 100,000$ . In each case print out

$$\sqrt{N} \left| \frac{1}{N} \sum_{i=1}^N x_i^k - \frac{1}{k+1} \right| \quad (10.14)$$

to check that it is of order 1.

9. Test the mildly correlated series  $r_1(1 - r_1)r_2(1 - r_2)r_3(1 - r_3)\dots$  with (10.13) for  $N = 100, 10,000, 100,000$ . Again print out the deviation from the expected result and divide the deviation by  $1/\sqrt{N}$  to check that it is of order 1.

## 11

# Monte Carlo Applications

Now that we have an idea of how to use the computer to generate a series of pseudo random numbers, we must build some confidence that using these numbers in a calculation is a way of incorporating the element of chance into a simulation. We do this first by simulating a random walk and then an atom decaying spontaneously. After that, we show how knowing the statistics of random numbers leads to the best way to evaluate multidimensional integrals.

### 11.1

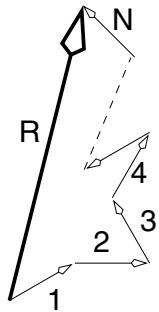
#### A Random Walk (Problem)

There are many physical processes, such as Brownian motion and electron transport through metals, in which a particle appears to move randomly. For example, consider a perfume molecule released in the middle of a classroom. It collides randomly with other molecules in the air and eventually reaches the instructor's nose. The **problem** is to determine how many collisions, on average, the molecule must make to travel a radial distance of  $R$ , if it travels a step length  $r_{\text{rms}}$  on the average (a root-mean-square average) between collisions.

##### 11.1.1

##### Random Walk Simulation

There are a number of ways to simulate a random walk, and (surprise, surprise) different assumptions give different physics. We will present the simplest approach for a 2D walk, with a minimum of theory, and end up with a model for normal diffusion. The research literature is full of discussions of various versions of this problem. For example, Brownian motion corresponds to the limit in which the individual step lengths approach zero with no time delay between steps. Additional refinements include collisions within a moving medium (abnormal diffusion), including the velocities of the particles, or even pausing between steps. Models such as these are discussed in Chap. 20.



**Fig. 11.1** Some of the  $N$  steps in a random walk that end up a distance  $R$  from the origin. Notice how the  $\Delta x$ 's for each step add algebraically.

In a random-walk simulation, such as that in Fig. 11.1 from the code `Walk.java` on the Instructor's CD, an artificial *walker* takes many steps, usually with the *direction* of each step *independent* from the direction of the previous one (Fig. 11.1). For our model, we start at the origin and take  $N$  steps in the  $x$ - $y$  plane of *lengths* (not coordinates)

$$(\Delta x_1, \Delta y_1), (\Delta x_2, \Delta y_2), (\Delta x_3, \Delta y_3), \dots, (\Delta x_n, \Delta y_N) \quad (11.1)$$

The radial distance  $R$  from the starting point traveled after  $N$  steps is

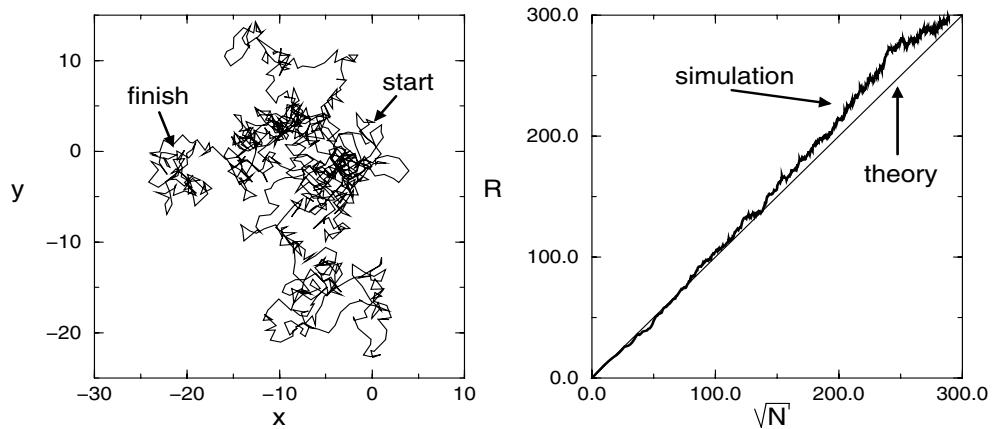
$$\begin{aligned} R^2 &= (\Delta x_1 + \Delta x_2 + \dots + \Delta x_N)^2 + (\Delta y_1 + \Delta y_2 + \dots + \Delta y_N)^2 \\ &= \Delta x_1^2 + \Delta x_2^2 + \dots + \Delta x_N^2 + 2\Delta x_1\Delta x_2 + 2\Delta x_1\Delta x_3 + 2\Delta x_2\Delta x_1 + \dots \\ &\quad + (x \rightarrow y) \end{aligned} \quad (11.2)$$

Equation (11.2) is valid for any walk. If it is random, the particle is equally likely to in any direction in each step. *On average*, for a large number of random steps, all the cross terms in (11.2) will vanish and we will be left with

$$\begin{aligned} R^2 &\simeq \Delta x_1^2 + \Delta x_2^2 + \dots + \Delta x_N^2 + \Delta y_1^2 + \Delta y_2^2 + \dots + \Delta y_N^2 = N\langle r^2 \rangle \\ \Rightarrow R &\simeq \sqrt{Nr_{\text{rms}}} \end{aligned} \quad (11.3)$$

Here  $r_{\text{rms}}$  is the average (*root-mean-squared*) step size.

In summary, (11.3) indicates that for a walk of  $N$  steps covering a total distance of  $Nr_{\text{rms}}$ , the walk ends up, on the average, a radial distance  $\sqrt{Nr_{\text{rms}}}$  from the starting point. For large  $N$  this is significantly less than  $Nr_{\text{rms}}$ , but is *not* zero (which is the average of the *displacement* vector). In our experience, practical simulations agree with this theory, but rarely perfectly, with the level of agreement depending upon how the averages are taken, and just how the randomness is built into each step.



**Fig. 11.2** *Left:* A computer simulation of a random walk. *Right:* the distance covered in a simulated random walk of  $N$  steps compared to the theoretical prediction (11.3).

### 11.1.2

#### Implementation: Random Walk

The program `Walk.java` on the instructor's CD contains our simulation of a random walk. Its key element is random values for the  $x$  and  $y$  components of each step,

```
x += (randnum.nextDouble() - 0.5); y += (randnum.nextDouble() - 0.5);
r[i] += Math.sqrt((x*x)+(y*y)); // radius
```

where we leave off the scaling factor that normalizes each step to length 1. When using your computer to simulate a random walk, you should only expect to obtain (11.3) as the average displacement after many trials, not necessarily as the answer for each trial. You may get different answers depending on how you take your random steps (Fig. 11.2). Start at the origin and take a 2D random walk with your computer.

1. To increase the amount of randomness, independently choose random values for  $\Delta x'$  and  $\Delta y'$  in the range  $[-1, 1]$ . Then normalize them so that the step is of unit length:

$$\Delta x = \frac{1}{L} \Delta x' \quad \Delta y = \frac{1}{L} \Delta y' \quad L = \sqrt{\Delta x'^2 + \Delta y'^2}$$

2. Use a plotting program to draw maps of several independent random walks, each of 1000 steps. Comment on whether these look like what you expect of a random walk.
3. If you have your walker taking  $N$  steps in a single trial, then conduct a total of  $K \approx \sqrt{N}$  trials. Each trial should have  $N$  steps and start with a different seed.

4. Calculate the mean-square distance  $R^2$  for each trial, and then take the average of  $R^2$  for all your  $K$  trials:

$$\langle R^2(N) \rangle = \frac{1}{K} \sum_{k=1}^K R_{(k)}^2(N)$$

5. Plot the root mean-square distance  $R_{\text{rms}} = \sqrt{\langle R^2(N) \rangle}$  as a function of  $\sqrt{N}$ . Values of  $N$  should start with a small number where  $R \simeq \sqrt{N}$  is not expected to be accurate, and end at a quite large value, where 2–3 places of accuracy should be expected on the average.

## 11.2

### Radioactive Decay (Problem)

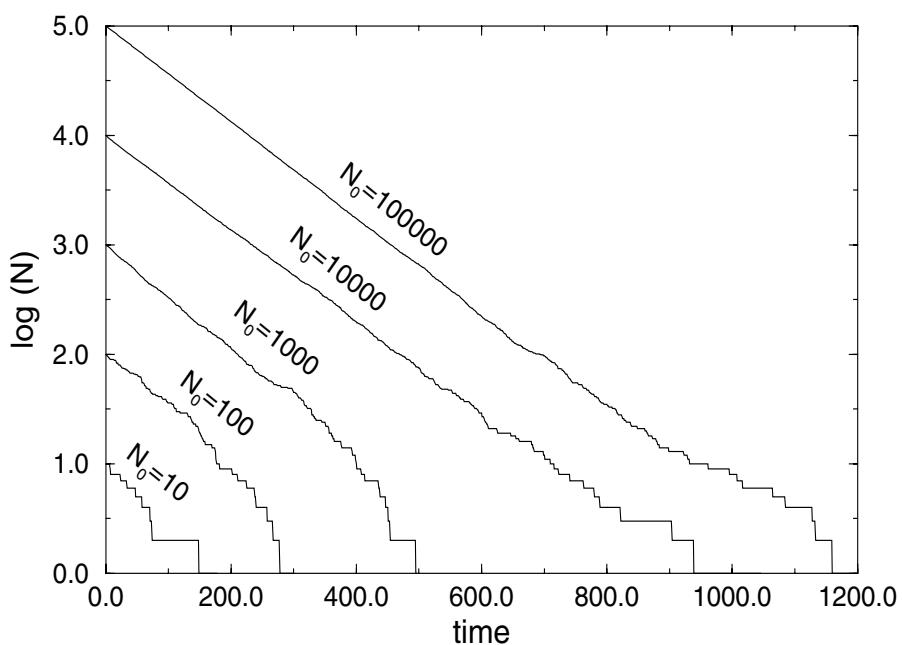
We have already encountered spontaneous radioactive decay in Chap. 8, where we fit an exponential function to a decay spectrum. Your **problem** now is to simulate how a small number of radioactive particles decay. In particular, you are to determine when radioactive decay looks exponential and when it looks *stochastic* (that is, determined by chance). Because the exponential decay law is only a large-number approximation to the natural process, our simulation should be closer to nature than the exponential decay law (Fig. 11.3). In fact, if you go to the CD and “listen” to the output of the decay simulation code, what you hear sounds very much like a Geiger counter: a convincing demonstration of how realistic the simulation is.

Spontaneous decay is a natural process in which a particle, with no external stimulation, and at one instant in time, decays into other particles. Because the exact moment when any one particle decays is random, it does not matter how long the particle has been around or what is happening to the other particles. In other words, the probability  $\mathcal{P}$  of any one particle decaying per unit time is a constant, and when that particle decays, it is gone forever. Of course, as the number of particles decreases with time, so will the number of decays. Nonetheless, the probability of any one particle decaying in some time interval is always the same constant as long as the particle still exists.

#### 11.2.1

##### Discrete Decay (Model)

Imagine having a sample of  $N(t)$  radioactive nuclei at time  $t$  (Fig. 11.3, left). Let  $\Delta N$  be the number of particles that decay in some small time interval  $\Delta t$ . We convert the statement “the probability  $\mathcal{P}$  of any one particle decaying per unit time is a constant” into an equation by noting that the decay probability per particle,  $\Delta N/N$ , is proportional to the length of the time interval over



**Fig. 11.3** Semilog plots of the results from several decay simulations. Notice how the decay appears exponential (like a straight line) when the number of nuclei is large, but stochastic for  $\log N \leq 2.0$ .

which we observe the particle

$$\mathcal{P} = \frac{\Delta N(t)}{N(t)} = -\lambda \Delta t \quad \Rightarrow \quad \frac{\Delta N(t)}{N(t)\Delta t} = -\lambda \quad (11.4)$$

where  $\lambda$  is a constant. Sure enough, (11.4) says that the probability of any one particle decaying per unit time is a constant.

Equation (11.4) is a *finite-difference equation* in which  $\Delta N(t)$  and  $\Delta t$  are experimental observables. Although it cannot be integrated the way a differential equation can, it can be solved numerically or algebraically. Because the decay process is random, we cannot predict an exact value for  $\Delta N(t)$ . Instead, we may think of  $\Delta N(t)$  as the average number of decays when observations are made of many identical systems of  $N$  radioactive particles.

We convert (11.4) into a finite-difference equation for the decay rate by multiplying both sides by  $N(t)$ :

$$\frac{\Delta N(t)}{\Delta t} = -\lambda N(t) \quad (11.5)$$

The absolute decay rate  $\Delta N(t)/\Delta t$  is called the *activity*, and because it is proportional to the number of particles present, it too has an exponential-like

decay in time. However, the dynamics of the simulation is still a random process, so eventually the activity too becomes stochastic. Actually, because the activity  $\Delta N(t)/\Delta t$  is proportional to the difference in random numbers, its stochastic nature become evident before that of  $N(t)$ .

### 11.2.2

#### Continuous Decay (Model)

When the number of particles  $N \rightarrow \infty$  and the observation time interval approaches zero, an approximate form of the radioactive decay law (11.5) results:

$$\frac{\Delta N(t)}{\Delta t} \rightarrow \frac{dN(t)}{dt} = -\lambda N(t) \quad (11.6)$$

This can be integrated to obtain the exponential decay law for the number and for the activity:

$$N(t) = N(0)e^{-\lambda t} = N(0)e^{-t/\tau} \quad (11.7)$$

$$\frac{dN}{dt}(t) = -\lambda N(0)e^{-\lambda t} = \frac{dN}{dt}(0)e^{-\lambda t} \quad (11.8)$$

In this limit we get exponential decay. We identify the decay rate  $\lambda$  with the inverse of the lifetime:

$$\lambda = \frac{1}{\tau} \quad (11.9)$$

We see from its derivation that exponential decay is a good description of nature only *on the average*, and only for a large number of particles. The basic law of nature (11.4) is always valid, but, as we will see in the simulation, (11.8) becomes less and less accurate as the number of particles gets smaller and smaller.

### 11.2.3

#### Decay Simulation

A program for simulating radioactive decay is surprisingly simple, but not without its subtleties. It increases time in discrete steps of  $\Delta t$ , and at each time counts how many nuclei have decayed during the last  $\Delta t$ . The simulation quits when there are no nuclei left. Such being the case, we have an outer loop over the time steps  $\Delta t$ , and an inner loop over the nuclei that are remaining at the present time. The pseudocode for this simulation is simple:

```
input N, lambda t=0 while N > 0
    DeltaN = 0
    for i = 1...N
        if (r_i < lambda) DeltaN = DeltaN + 1
```

```

end for
t = t +1
N = N - DeltaN
Output t, DeltaN, N
end while

```

At some point in writing the simulation program, we set the scale, or units, in which is measured. Since the decay rate parameter  $\lambda = 1/\tau$ , where  $\tau$  is the lifetime of the nucleus, picking a value of  $\lambda$  essentially sets the time scale. Since the simulation compares  $\lambda$  to a random number, and since random numbers are usually in the range  $0 \leq r_i \leq 1$ , it is convenient to pick time units for which  $0 \leq \lambda \leq 1$  (for example,  $\lambda \simeq 0.3$  is a good place to start). This means that when the do loop counts time with  $\Delta t = 1$  units, the time scale has been set as  $1/\lambda$ .

As an example, let us imagine a nucleus with a lifetime  $\tau = 10$  s. This corresponds to a decay parameter  $\lambda = 1/\tau = 0.1/\text{s}$ . So if we use  $\lambda = 0.1$  in the simulation, each time step would correspond to 1 s. Likewise, if we have a nucleus with a lifetime  $\tau = 10 \times 10^{-6}$  s, then this corresponds to a decay parameter  $\lambda = 1/\tau = 0.1/10^{-6}$  s. So if we use  $\lambda = 0.1$  in the simulation, each time step would correspond to  $10^{-6}$  of a second. The actual value of the decay rate  $\Delta N/\Delta t$  in particles per second would then be  $\Delta N/10^{-6}$ . However, unless you plan to compare your simulation to experimental data, you do not have to worry about the scale for time and can output  $\Delta N$  as the decay rate (it is actually the physics behind the slopes and relative magnitudes of the graphs that we want to show).

### 11.3 Decay Implementation and Visualization

Write your own program to simulate radioactive decay, using our sample program in Listing 11.1 as a guide (it is simple). You should obtain results like those on the right of Fig. 11.3.

**Listing 11.1:** Decay.java simulates spontaneous decay having decay whenever a random number is smaller than a normalized decay rate.

```

// Decay.java: Spontaneous decay simulation

import java.io.*;
import java.util.*;

public class Decay {
    static double lambda = 0.01;                                // Decay constant
    static int max = 1000, time_max = 500, seed = 68111;        // Params

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {

```

```

int atom, time, number, nloop;
double decay;

PrintWriter w =
    new PrintWriter(new FileOutputStream("decay.dat"), true);
number = nloop = max;                                // Initial value
Random r = new Random(seed);                         // Seed random generator
                                                     // Time loop
for ( time = 0; time <= time_max; time++ ) {           // Atom loop
    for ( atom = 1; atom <= number; atom++ ) {
        decay = r.nextDouble();
        if (decay < lambda) nloop--;
    }                                                       // An atom decays
    number = nloop;
    w.println( " " + time + " " + (double)number/max);
}
System.out.println("data stored in decay.dat");
}
}

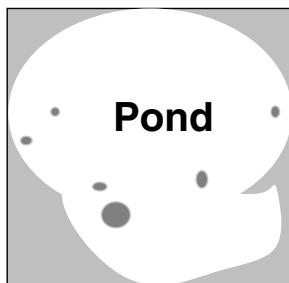
```

1. Plot the logarithm of the number left  $\ln N(t)$  and the logarithm of the decay rate  $\ln(\Delta N(t))$  versus time. Note that the simulation measures time in steps of  $\Delta t$  (generation number).
2. Check that you obtain what looks like exponential decay when you start with large values for  $N(0)$ , and that for small  $N(0)$  you get a stochastic process (the large  $N(0)$  simulation is also stochastic, it just does not look it).
3. Create two plots, one showing that the slopes of plots of  $N(t)$  versus  $t$  are *independent* of  $N(0)$ , and the another showing that the slope is proportional to  $\lambda$ .
4. Create a plot to show that, within the expected stochastic variations,  $\ln N(t)$  and  $\ln(\Delta N(t))$  are proportional.
5. Explain in your own words how a process that is spontaneous and random at its very heart leads to exponential decay.
6. How does your simulation show that the decay is exponential and not a power law like  $N = \beta t^{-\alpha}$ ?

#### 11.4

##### Integration by Stone Throwing (Problem)

Imagine yourself as a farmer walking to your furthermost field to add chemicals to a pond having an algae explosion. You get there, only to read the in-



**Fig. 11.4** Throwing stones in a pond as a technique for measuring its area. There is a tutorial of this on the CD where you can see the actual “splashes” (the dark spots) used in an integration.

structions and discover that you need to know the area of the pond to get the correct concentration. Your **problem** is to measure the area of this irregularly shaped pond with just the materials at hand [17].

## 11.5

### Integration by Rejection (Theory)

It is hard to believe that Monte Carlo techniques could be used to evaluate integrals. After all, we do not want to gamble on their values! While it is true that other methods are preferable for single and double integrals, when the number of integrations required gets large, Monte Carlo techniques are the best!

For our pond problem, we will use the *sampling* technique (Fig. 11.4):

1. Walk off a box that completely encloses the pond and remove any pebbles within the box.
2. Measure the lengths of the sides in natural units like *feet*. This tells you the area of the enclosing box  $A_{\text{box}}$ .
3. Grab a bunch of pebbles and throw them up in the air in random directions.
4. Count the number of splashes in the pond  $N_{\text{pond}}$  and the number of pebbles lying on the ground within your box  $N_{\text{box}}$ .
5. Assuming that you threw the pebbles uniformly and randomly, the number of pebbles falling into the pond should be proportional to the

area of the pond  $A_{\text{pond}}$ . You determine that area from the simple ratio

$$\begin{aligned} \frac{N_{\text{pond}}}{N_{\text{pond}} + N_{\text{box}}} &= \frac{A_{\text{pond}}}{A_{\text{box}}} \\ \Rightarrow A_{\text{pond}} &= \frac{N_{\text{pond}}}{N_{\text{pond}} + N_{\text{box}}} A_{\text{box}} \end{aligned} \quad (11.10)$$

### 11.5.1

#### Stone Throwing Implementation

Use sampling (Fig. 11.4) to perform a 2D integration and thereby determine  $\pi$ :

1. Imagine a circular pond centered at the origin and enclosed in a square of side 2.
2. We know the analytic result

$$\oint dA = \pi \quad (11.11)$$

3. Generate a sequence of random numbers  $\{r_i\}$  in  $[-1, 1]$ .
4. For  $i = 1$  to  $N$ , pick  $(x_i, y_i) = (r_{2i-1}, r_{2i})$ .
5. If  $x_i^2 + y_i^2 < 1$ , let  $N_{\text{pond}} = N_{\text{pond}} + 1$ , else let  $N_{\text{box}} = N_{\text{box}} + 1$ .
6. Use (11.10) to calculate the area and in this way  $\pi$ .

Try increasing  $N$  until you get  $\pi$  to three significant figures (we do not ask much; that is only slide-rule accuracy).

### 11.5.2

#### Integration by Mean Value (Math)

The standard Monte Carlo technique for integration is based on the *mean-value theorem* (presumably familiar from elementary calculus):

$$I = \int_a^b dx f(x) = (b - a) \langle f \rangle \quad (11.12)$$

The theorem states the obvious if you think of integrals as areas: the value of the integral of some function  $f(x)$  between  $a$  and  $b$  equals the length of the interval  $(b - a)$  times the average value of the function over that interval  $\langle f \rangle$  (Fig. 11.5). The integration algorithm uses Monte Carlo techniques to evaluate the mean in (11.12). With a sequence  $x_i$  of  $N$  uniform random numbers in  $[a, b]$ ,

we want to determine the *sample mean* by *sampling* the function  $f(x)$  at these points:

$$\langle f \rangle \simeq \frac{1}{N} \sum_{i=1}^N f(x_i) \quad (11.13)$$

This gives us the very simple integration rule:

$$\int_a^b dx f(x) \simeq (b-a) \frac{1}{N} \sum_{i=1}^N f(x_i) = (b-a) \langle f \rangle \quad (11.14)$$

Equation (11.14) looks much like our standard algorithm for integration (5.3), with the “points”  $x_i$  chosen randomly and with constant weights  $w_i = (b-a)/N$ . Because no attempt has been made to get the best answer for a given value of  $N$ , this is by no means an optimized way to evaluate integrals; but you will admit it is simple. If we let the number of samples of  $f(x)$  approach infinity  $N \rightarrow \infty$ , or we keep the number of samples finite and take the average of infinitely many runs, the laws of statistics assure us that, barring roundoff errors, (11.14) approaches the correct answer.

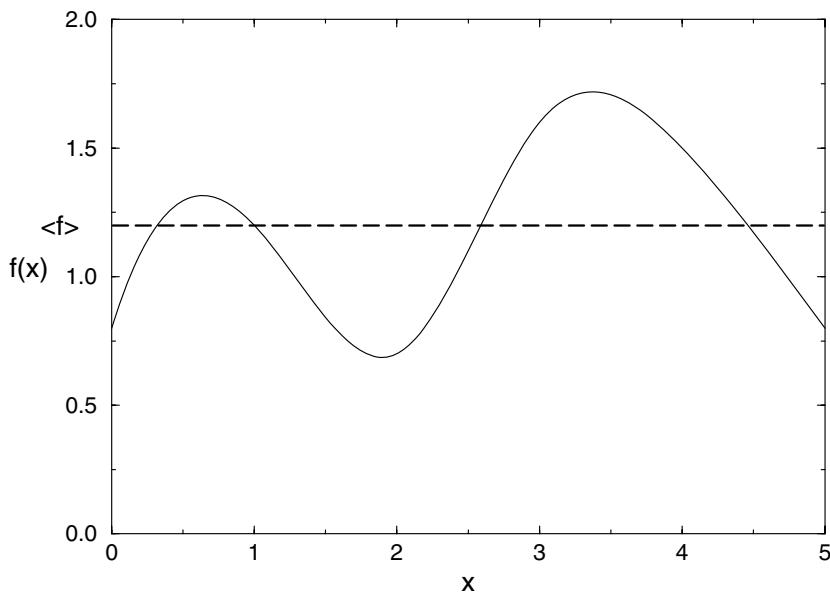
For those readers who are somewhat familiar with statistics, we remind you that the uncertainty in the value obtained for the integral  $I$  after  $N$  samples of  $f(x)$  is measured by the standard deviation  $\sigma_I$ . The standard deviation of the integrand  $f$  in the sampling is an intrinsic property of the function  $f(x)$ , that is, something we do not change by taking more samples. For normal distributions, the two are related by

$$\sigma_I \approx \frac{1}{\sqrt{N}} \sigma_f \quad (11.15)$$

This means that for large  $N$ , the error in the value of the integral always decreases as  $1/\sqrt{N}$ .

## 11.6 High-Dimensional Integration (Problem)

Let us say that we want to calculate some properties of a small atom such as magnesium with 12 electrons. To do that, we need to integrate some function over the three coordinates of each electron. This amounts to a  $3 \times 12 = 36$ -dimensional integral. If we use 64 points for each integration, this requires some  $64^{36} \simeq 10^{65}$  evaluations of the integrand. If the computer were fast and could evaluate the integrand a million times per second, this would take some  $10^{59}$  seconds, which is significantly longer than the age of the universe ( $\sim 10^{17}$  s).



**Fig. 11.5** The area under the curve  $f(x)$  is the same as that under the dashed line  $y = \langle f \rangle$ .

Your **problem** is to find a way to perform multidimensional integrations so that you are still alive to savor the answers. Specifically, evaluate the 10D integral

$$I = \int_0^1 dx_1 \int_0^1 dx_2 \cdots \int_0^1 dx_{10} (x_1 + x_2 + \cdots + x_{10})^2 \quad (11.16)$$

Check your numerical answer against the analytic one,  $\frac{155}{6}$ .

### 11.6.1

#### Multidimensional Monte Carlo

It is easy to generalize mean-value integration to many dimensions by picking random points in a multidimensional space. For example,

$$\int_a^b dx \int_c^d dy f(x, y) \simeq (b-a)(d-c) \frac{1}{N} \sum_i^N f(\mathbf{x}_i) = (b-a)(d-c) \langle f \rangle \quad (11.17)$$

### 11.6.2

#### Error in Multidimensional Integration (Assessment)

When we perform a multidimensional integration, the error in the Monte Carlo technique, being statistical, decreases as  $1/\sqrt{N}$ . This is valid even if the  $N$  points are distributed over  $D$  dimensions. In contrast, when we use these

same  $N$  points to perform a  $D$ -dimensional integration as  $D$  one-dimensional integrals, we use  $N/D$  points for each integration. For fixed  $N$ , this means that the number of points used for each integration decreases as the number of dimensions  $D$  increases, and so the error in each integration increases with  $D$ . Furthermore, the total error will be approximately  $N$  times the error in each integral. If we put these trends together and look at a particular integration rule, we would find that at a value of  $D \simeq 3\text{--}4$  the error in Monte Carlo integration is similar to that of conventional schemes. For larger values of  $D$ , the Monte Carlo method is more accurate!

### 11.6.3

#### **Implementation: 10D Monte Carlo Integration**

Use a built-in random-number generator to perform the 10-dimensional Monte Carlo integration in (11.16). Our program `Int10d.java` is available on the instructor's CD.

1. Conduct 16 trials and take the average as your answer.
2. Try sample sizes of  $N = 2, 4, 8, \dots, 8192$ .
3. Plot the absolute value of the error versus  $1/\sqrt{N}$  and try to identify linear behavior.

## 11.7

### **Integrating Rapidly Varying Functions (Problem) ⊖**

It is common in many physical applications to integrate a function with an approximately Gaussian dependence on  $x$ . The rapid falloff of the integrand means that our Monte Carlo integration technique would require an incredibly large number of integration points to obtain even modest accuracy. Your **problem** is to make Monte Carlo integration more efficient for rapidly varying integrands.

#### 11.7.1

##### **Variance Reduction ⊖ (Method)**

If the function being integrated never differs much from its average value, then the standard Monte Carlo mean-value method (11.14) should work well with a not too ridiculously large number of points. Yet for a function with a large *variance* (i.e., one that is not “flat”), many of the random evaluations of the function may occur where the function makes a slight contribution to the integral; this is, basically, a waste of time. The method can be improved by mapping the function  $f$  into a function  $g$  that has a smaller variance over the

interval. We indicate two methods here and refer you to the References (at the end of this book) for more details.

The first method is a *variance reduction* or *subtraction technique* in which we devise a flatter function on which to apply the Monte Carlo technique. Suppose we construct a function  $g(x)$  with the following properties on  $[a, b]$ :

$$|f(x) - g(x)| \leq \epsilon \quad \int_a^b dx g(x) = J \quad (11.18)$$

We now evaluate the integral of  $f(x) - g(x)$  and add the result to  $J$  to obtain the required integral

$$\int_a^b dx f(x) = \int_a^b dx \{f(x) - g(x)\} + J \quad (11.19)$$

If we are clever enough to find a simple  $g(x)$  that makes the variance of  $f(x) - g(x)$  less than that of  $f(x)$ , and that we can integrate analytically, we obtain more accurate answers in less time.

### 11.7.2

#### Importance Sampling (Method) ⊙

A second method to improve Monte Carlo integration is called *importance sampling* because it lets us sample the integrand in the most important regions. It derives from expressing the integral in the form

$$I = \int_a^b dx f(x) = \int_a^b dx w(x) \frac{f(x)}{w(x)} \quad (11.20)$$

If we now use  $w(x)$  as the *weighting function* or *probability distribution* for our random numbers, the integral can be approximated as

$$I = \left\langle \frac{f}{w} \right\rangle \simeq \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{w(x_i)} \quad (11.21)$$

The improvement with (11.21) is that a judicious choice of the weighting function  $w(x)$  makes  $f(x)/w(x)$  a rather constant function and consequently easier to integrate.

### 11.7.3

#### Implementation: Nonuniform Randomness ⊙

In order for  $w(x)$  to be the weighting function for random numbers over  $[a, b]$ , we want it with the properties

$$\int_a^b dx w(x) = 1 \quad (w(x) > 0) \quad d\mathcal{P}(x \rightarrow x + dx) = w(x)dx \quad (11.22)$$

where  $d\mathcal{P}$  is the probability of obtaining an  $x$  in the range  $x \rightarrow x + dx$ . For the uniform distribution over  $[a, b]$ ,  $w(x) = 1/(b - a)$ .

- **Inverse Transform/Change of Variable Method:** Let us consider a change of variables that take our original integral  $I$  (11.20) to the form

$$I = \int_a^b dx f(x) = \int_0^1 dW \frac{f[x(W)]}{w[x(W)]} \quad (11.23)$$

Our aim is to make this transformation such that there are equal contributions from all parts of the range in  $W$ , that is, we want to use a uniform sequence of random numbers for  $W$ . To determine the new variable, we start with  $u(r)$ , the uniform distribution over  $[0, 1]$ ,

$$u(r) = \begin{cases} 1 & \text{for } 0 \leq r \leq 1 \\ 0 & \text{otherwise} \end{cases} \quad (11.24)$$

We want to find a mapping  $r \leftrightarrow x$  or probability function  $w(x)$  for which probability is conserved:

$$w(x)dx = u(r)dr \quad \Rightarrow \quad w(x) = \left| \frac{dr}{dx} \right| u(r) \quad (11.25)$$

This means that even though  $x$  and  $r$  are related by some (possibly) complicated mapping,  $x$  is also random with the probability of  $x$  lying in  $x \rightarrow x + dx$  equal to that of  $r$  lying in  $r \rightarrow r + dr$ .

To find the mapping between  $x$  and  $r$  (the tricky part), we change variables to  $W(x)$  defined by the integral

$$W(x) = \int_{-\infty}^x dx' w(x') \quad (11.26)$$

We recognize  $W(x)$  as the (incomplete) integral of the probability density  $u(r)$  up to some point  $x$ . It is in this way another type of distribution function specifically, the integrated probability of finding a random number less than the value  $x$ . The function  $W(x)$  is on that account called a *cumulative distribution function* and can also be thought of as the area to the left of  $r = x$  on the plot of  $u(r)$  versus  $r$ . It follows immediately from the definition (11.26) that  $W(x)$  has the properties

$$W(-\infty) = 0 \quad W(\infty) = 1 \quad (11.27)$$

$$\frac{dW(x)}{dx} = w(x) \quad dW(x) = w(x)dx = u(r)dr \quad (11.28)$$

Consequently,  $W_i = \{r_i\}$  is a uniform sequence of random numbers and we invert (11.26) to obtain  $x$  values distributed with probability  $w(x)$ .

The crux of this technique is being able to invert (11.26) to obtain  $x = W^{-1}(r)$ . Let us look at some analytic examples to get a feel for these steps (numerical inversion is possible and frequent in realistic cases).

- **Uniform Weight Function  $w$ :** We start off with our old friend, the uniform distribution:

$$w(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases} \quad (11.29)$$

By following the rules this then leads to

$$W(x) = \int_a^x dx' \frac{1}{b-a} = \frac{x-a}{b-a} \quad (11.30)$$

$$\Rightarrow x = a + (b-a)W \quad \Rightarrow \quad W^{-1}(r) = a + (b-a)r \quad (11.31)$$

where  $W(x)$  is always taken as uniform. In this way we generate uniform random  $r : [0, 1]$  and uniform random  $x = a + (b-a)r : [a, b]$ .

- **Exponential Weight:** We want to generate points with the exponential distribution:

$$w(x) = \begin{cases} \frac{1}{\lambda} e^{-x/\lambda} & \text{for } x > 0 \\ 0, & \text{for } x < 0 \end{cases} \quad W(x) = \int_0^x dx' \frac{1}{\lambda} e^{-x'/\lambda} = 1 - e^{-x/\lambda}$$

$$\Rightarrow x = -\lambda \ln(1 - W) \equiv -\lambda \ln(1 - r) \quad (11.32)$$

In this way we generate uniform random  $r : [0, 1]$  and obtain  $x = -\lambda \ln(1 - r)$  distributed with an exponential probability distribution for  $x > 0$ .

Notice that our prescription (11.20)–(11.21) tells us to use  $w(x) = e^{-x/\lambda}/\lambda$  to remove the exponential-like behavior out of an integrand and place it into the weights and scaled points ( $0 \leq x_i \leq \infty$ ). Because the resulting integrand will vary less, it may be approximated better as a polynomial:

$$\int_0^\infty dx e^{-x/\lambda} f(x) \simeq \frac{\lambda}{N} \sum_{i=1}^N f(x_i) \quad x_i = -\lambda \ln(1 - r_i) \quad (11.33)$$

- **Gaussian (Normal) Distribution:**

We want to generate points with a normal distribution:

$$w(x') = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x'-\bar{x})^2/2\sigma^2} \quad (11.34)$$

This by itself is rather hard, but it is made easier by generating uniform distributions in angles and then using trigonometric relations to convert these

to a Gaussian distribution. But before doing that, we keep things simple by realizing that we can obtain (11.34) with mean  $\bar{x}$  and standard deviation  $\sigma$ , by scaling and a translation of a simpler  $w(x)$ :

$$w(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \quad x' = \sigma x + \bar{x} \quad (11.35)$$

We start by generalizing the statement of probability conservation for two different distributions (11.25) to two dimensions [9]:

$$p(x, y) dx dy = u(r_1, r_2) dr_1 dr_2 \quad \Rightarrow \quad p(x, y) = u(r_1, r_2) \left| \frac{\partial(r_1, r_2)}{\partial(x, y)} \right| \quad (11.36)$$

We recognize the term in vertical bars as the Jacobian determinant:

$$J = \left| \frac{\partial(r_1, r_2)}{\partial(x, y)} \right| \stackrel{\text{def}}{=} \frac{\partial r_1}{\partial x} \frac{\partial r_2}{\partial y} - \frac{\partial r_2}{\partial x} \frac{\partial r_1}{\partial y} \quad (11.37)$$

To specialize to a Gaussian distribution, we consider  $2\pi r$  as angles obtained from a uniform random distribution  $r$ , and  $x$  and  $y$  as Cartesian coordinates that will have a Gaussian distribution. The two are related by

$$x = \sqrt{-2 \ln r_1} \cos 2\pi r_2 \quad y = \sqrt{-2 \ln r_1} \sin 2\pi r_2 \quad (11.38)$$

The inversion of this mapping produces the Gaussian distribution

$$r_1 = e^{-(x^2+y^2)/2} \quad r_2 = \frac{1}{2\pi} \tan^{-1} \frac{y}{x} \quad J = -\frac{e^{-(x^2+y^2)/2}}{2\pi} \quad (11.39)$$

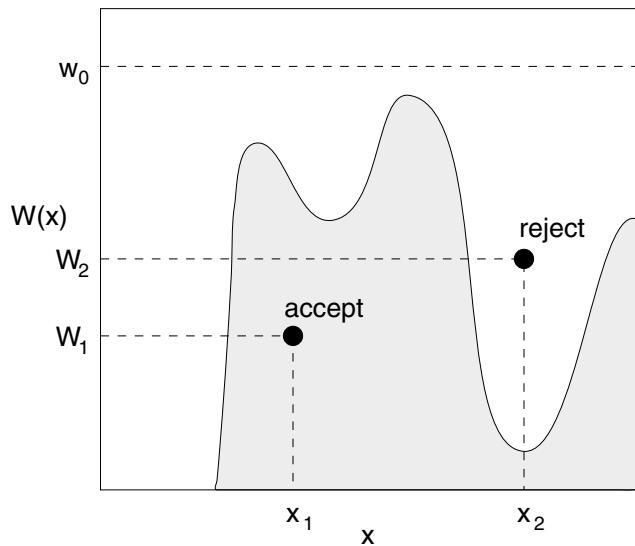
The solution to our problem is at hand. We use (11.38) with  $r_1$  and  $r_2$  uniform random distributions, and  $x$  and  $y$  will then be Gaussian random distributions centered around  $x = 0$ .

- **Alternate Gaussian Distribution:** The central limit theorem can be used to deduce a Gaussian distribution via a simple summation. The theorem states, under rather general conditions, that if  $\{r_i\}$  is a sequence of mutually independent random numbers, then the sum

$$x_N = \sum_{i=1}^N r_i \quad (11.40)$$

is distributed normally. This means that the generated  $x$  values have the distribution

$$P_N(x) = \frac{\exp \left[ -\frac{(x-\mu)^2}{2\sigma^2} \right]}{\sqrt{2\pi\sigma^2}} \quad \mu = N\langle r \rangle \quad \sigma^2 = N(\langle r^2 \rangle - \langle r \rangle^2) \quad (11.41)$$



**Fig. 11.6** The von Neumann rejection technique for generating random points with weight  $w(x)$ .

#### 11.7.4

##### von Neumann Rejection (Method)

A simple and ingenious method for generating random points with a probability distribution  $w(x)$  was deduced by von Neumann. This method is essentially the same as the rejection or sampling method used to guess the area of the pond; only now the pond is replaced by the weighting function  $w(x)$  and the arbitrary box around the lake by the arbitrary constant  $W_0$ . Imagine a graph of  $w(x)$  versus  $x$  (Fig. 11.6). Walk off your box by placing the line  $W = W_0$  on the graph, with the only condition being  $W_0 \geq w(x)$ . We next “throw stones” at this graph and count only those that fall into the  $w(x)$  pond. That is, we generate uniform distributions in  $x$  and  $y \equiv W$  with the maximum  $y$  value equal to the width of the box  $W_0$ :

$$(x_i, W_i) = (r_{2i-1}, W_0 r_{2i}) \quad (11.42)$$

We then reject all  $x_i$  that do not fall into the pond:

$$\text{if } W_i < w(x_i) \text{ accept} \quad \text{if } W_i > w(x_i) \text{ reject} \quad (11.43)$$

The  $x_i$  values so accepted will have the weighting  $w(x)$  (Fig. 11.6). The largest acceptance occurs where  $w(x)$  is large, in this case for midrange  $x$ .

In Unit 12, *Thermodynamic Simulations: The Ising Model*, we apply a variation of the rejection technique known as the *Metropolis algorithm*. That algorithm has now become the cornerstone of computation thermodynamics.

## 11.7.5

**Nonuniform Assessment** ◉

Use the von Neumann rejection technique to generate a normal distribution of standard deviation 1, and compare to the preceding *Box–Muller* method.

## 12

# Thermodynamic Simulations: Ising Model

This chapter applies the Ising model and the Metropolis algorithm to simulate magnetic materials. This is a more advanced application of the Monte Carlo techniques studied in Chap. 11, in which thermodynamic equilibrium is simulated. It is an important topic which may give readers the first true understanding of what is “dynamic” in thermodynamics.

Your **problem** is to see if a simple model can explain the thermal behavior and phase transitions of ferromagnets.

### 12.1

#### Statistical Mechanics (Theory)

Ferromagnetic materials contain finite-size *domains* in which, even in the absence of an external magnetic field, the spins of all the atoms are aligned in the same direction. When an external magnetic field is applied to these materials at low temperatures, the different domains align and the material becomes “magnetized.” Yet, as the temperature is raised, the magnetism decreases and then goes through a *phase transition* at the Curie temperature, beyond which all magnetization vanishes.

When we say that an object is *at* a temperature  $T$ , we mean that the object’s atoms are in thermodynamic equilibrium at temperature  $T$ . While this may be an equilibrium state, it is a dynamic one in which system’s energy is fluctuating as it exchanges energy with the environment (it is *thermodynamics* after all). However, each atom does have an average energy proportional to  $T$ .

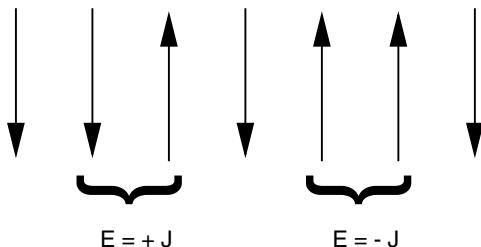
In the present problem we deal with the thermal properties of magnetized materials. The magnetism arises from the alignment of the spins of the atoms within domains. When the number of atoms is large, the problem is too big to solve completely, and so statistical methods are used to obtain average quantities (in most cases that is all we can measure, anyway). If the system is described microscopically by classical or quantum mechanics, then this method is called *statistical mechanics*.

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5



**Fig. 12.1** A 1D lattice of \$N\$ spins. The interaction energy \$V = \pm J\$ between nearest-neighbor pairs is shown for aligned and opposing spins.

Statistical mechanics starts with the elementary interactions among particles of a system and constructs the macroscopic thermodynamic properties such as temperature \$T\$ and internal energy \$U\$. The essential assumption is that all configurations of the system consistent with the constraints are possible. Because we have the temperature, volume, and number of particles fixed, we have a *canonical ensemble* or Boltzmann distribution. The energy \$E(\alpha\_j)\$ of a state \$\alpha\_j\$ in a canonical ensemble is not fixed, but rather is distributed with probabilities \$P(\alpha\_j)\$ according to the Boltzmann distribution:

$$P(\alpha_j) = \frac{e^{-E(\alpha_j)/kT}}{Z(T)} \quad Z(T) = \sum_{\alpha_j} e^{-E_j/kT} \quad (12.1)$$

where \$k\$ is Boltzmann's constant and \$T\$ is the temperature. The partition function \$Z(T)\$ in (12.1) is a weighted sum over states. Nonetheless, because we will be dealing with the *ratio* of probabilities, the \$Z(T)\$ factor cancels out, and we do not have to concern ourselves with it.

Notice that the Boltzmann distribution (12.1) does not require a thermal system to be in the state of lowest energy. Rather, it states that it is less likely for the system to have a high energy. Of course, as \$T \rightarrow 0\$, only the \$E = 0\$ state has a nonvanishing probability. For finite temperatures we expect the system's energy to have fluctuation on the order of \$kT\$.

## 12.2

### An Ising Chain (Model)

As our model, we consider \$N\$ magnetic dipoles fixed on the links of a linear chain (Fig. 12.1). (It is a straightforward generalization to handle 2- and 3D lattices, and indeed we do it as an exploration.) Because the particles are fixed, their positions and momenta are not dynamical variables, and we need worry only about their spins.

We assume that the particle at site  $i$  has spin  $s_i$ , which is either up or down:

$$s_i \equiv s_{z,i} = \pm \frac{1}{2} \quad (12.2)$$

Each possible configuration or state of the  $N$  particles is described by the quantum state vector

$$|\alpha_j\rangle = |s_1, s_2, \dots, s_N\rangle = \{\pm \frac{1}{2}, \pm \frac{1}{2}, \dots\} \quad j = 1, 2^N \quad (12.3)$$

Because the spin of each particle can assume any one of the *two* values, there are  $2^N$  different possible states of the  $N$  particles in the system. We do not need to concern ourselves with the symmetry of the wave function since fixed particles cannot be interchanged.

The energy of the system arises from the interaction of the spins with each other and with the external magnetic field  $B$ . We know from quantum mechanics that an electron's spin and magnetic moment are proportional to each other, so a "dipole-dipole" interaction is equivalent to a "spin-spin" interaction. We assume that each dipole interacts with the external magnetic field and with its nearest neighbor through the potential:

$$V_i = -J \mathbf{s}_i \cdot \mathbf{s}_{i+1} - g\mu_b \mathbf{s}_i \cdot \mathbf{B} \quad (12.4)$$

Here the constant  $J$  is called the *exchange energy* and is a measure of the strength of the spin-spin interaction. The constant  $g$  is the gyromagnetic ratio, that is, the proportionality constant between the angular momentum and magnetic moment. The constant  $\mu_b$  is the Bohr magneton, the unit for magnetic moments.

Because the  $2^N$  possible configurations of the spins become large even for small numbers of particles ( $2^{20} > 10^6$ ), the computer can examine all possible spin configurations only for small  $N$ . Realistic samples with  $\sim 10^{23}$  particles would be beyond imagination. Consequently, we apply a statistical approach based on the Boltzmann distribution (12.1), and assume that the statistics are valid even for moderate values of  $N$ . Just how large  $N$  must be for this to occur is one of the things we want you to discover with your simulations. In general, the answer depends on how good a description is required; for pedagogical purposes  $N \geq 200$  may appear statistical, with  $N \geq 2000$  more reliable.

The energy of the system to be used in the Boltzmann distribution (12.1) is the expectation value of the sum of  $V$  over the spins of the particles:

$$E(\alpha) = \langle \alpha | \sum_i V_i | \alpha \rangle = -J \sum_{i=1}^{N-1} s_i s_{i+1} - B \mu_b \sum_{i=1}^N s_i \quad (12.5)$$

An apparent paradox in the Ising model occurs when we turn off the external magnetic field and thereby eliminate a preferred direction in space. This

means that the average magnetization should vanish, even though the lowest energy state would have all spins aligned. The answer to this paradox is that the system with  $B = 0$  is unstable. Even if all the spins are aligned, there is nothing to stop a spontaneously reversal of all the spins. Indeed, natural magnetic materials have multiple finite domains with all the spins aligned, but with the different domains pointing in arbitrary directions. The instabilities in which a domains change directions are called *Bloch-wall transitions*. For simplicity, and to start, you may assume that  $B = 0$ , and just look at how the spins interact with each other. However, we recommend that you always include a very small external magnetic field in order to stabilize the simulation against spontaneous flipping of all the spins.

The equilibrium alignment of the spins depends critically on the sign of the exchange energy  $J$ . If  $J > 0$ , the lowest energy state will tend to have neighboring spins aligned. If the temperature is low enough, the ground state will be a *ferromagnet* with all spins aligned. If  $J < 0$ , the lowest energy state will tend to have neighbors with opposite spins. If the temperature is low enough, the ground state will be a *antiferromagnet* with alternating spins.

A simple model such as this has its limits. (Nonetheless, it is not hard to add improvements, such as longer range interactions, motion of the centers, higher multiplicity spin states, two and three dimensions, etc.) First, although the model is accurate in describing a system in thermal equilibrium, it is not quantitatively accurate in describing the *approach* to thermal equilibrium. On the one hand, nonequilibrium thermodynamics is a difficult subject in which the theory is not complete, while on the other hand, as part of our algorithm we postulate that only one spin gets flipped at a time, while a real magnetic material may flip many.

A fascinating aspect of magnetic materials is the existence of a critical temperature, the *Curie temperature*, above which the gross magnetization essentially vanishes. Below the Curie temperature the quantum state of the material has long-range order extending over macroscopic dimensions; above the Curie temperature there is only short-range order extending over atomic dimensions. Even though the 1D Ising model predicts realistic temperature dependences for the thermodynamic quantities, the model is too simple to support a phase transition. However, the 2D and 3D Ising models does support the Curie-temperature phase transition.

One of the most illuminating aspects of this simulation is the visualization showing that a system described by the Boltzmann distribution (12.1) does not have a single configuration. Rather, there is a continual and random interchange of thermal energy with the environment that leads to fluctuations in the total energy. Even at equilibrium, you should see the system fluctuating, with the fluctuations getting larger as the temperature rises.

### 12.2.1

#### Analytic Solutions

For very large numbers of particles, the thermodynamic properties of the 1D Ising model can be solved analytically [18]. The solution tells us that the average energy  $U$  is

$$\frac{U}{J} = -N \tanh \frac{J}{kT} = -N \frac{e^{J/kT} - e^{-J/kT}}{e^{J/kT} + e^{-J/kT}} = \begin{cases} N & kT \rightarrow 0 \\ 0 & kT \rightarrow \infty \end{cases} \quad (12.6)$$

The analytic results for the specific heat per particle and the magnetization are

$$C(kT) = \frac{1}{N} \frac{dU}{dT} = \frac{(J/kT)^2}{\cosh^2(J/kT)} \quad (12.7)$$

$$M(kT) = \frac{Ne^{J/kT} \sinh(B/kT)}{\sqrt{e^{2J/kT} \sinh^2(B/kT) + e^{-2J/kT}}}. \quad (12.8)$$

The **2D Ising model** has an analytic solution, but it was not an easy one to find [19,20]. Whereas the internal energy and heat capacity are found in terms of elliptic integrals, the spontaneous magnetization per particle has the rather simple form

$$M(T) = \begin{cases} 0 & T > T_c \\ \frac{(1+z^2)^{1/4}(1-6z^2+z^4)^{1/8}}{\sqrt{1-z^2}} & T < T_c \end{cases} \quad \begin{aligned} kT_c &\simeq 2.269185J \\ z &= e^{-2J/kT} \end{aligned} \quad (12.9)$$

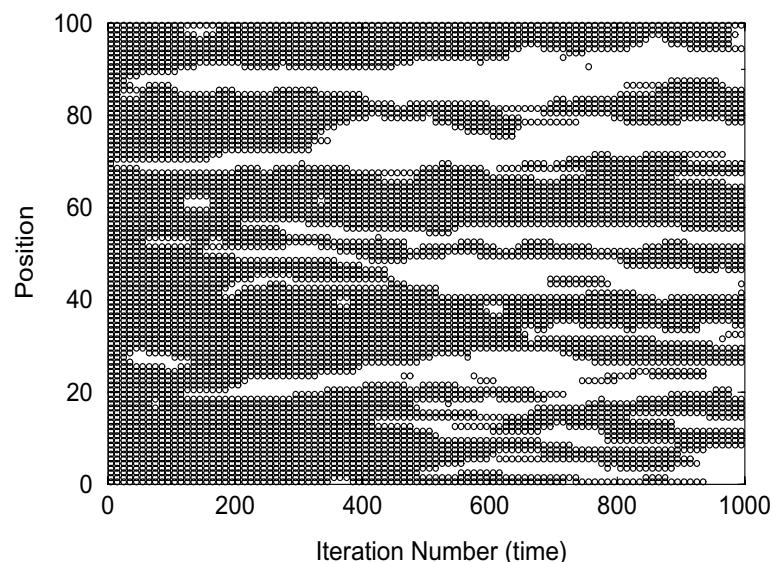
where the temperature is measured in units of the Curie temperature  $T_c$ , and  $z$  is a dimensionless variable.

### 12.3

#### The Metropolis Algorithm

We need an algorithm to evaluate the  $2^N$  sums that appear in the energy sum (12.5). This is analogous to a  $2^N$ -dimensional numerical integration, and we know from Section 11.6.2 that a Monte Carlo approach is best for high-dimensional integrations. Yet it would be a waste of time to generate all possible random configurations in a uniform manner because the Boltzmann factor essentially vanishes for those configurations whose energies are not close to the minimum energy. In other words, the majority of the terms we sum over hardly contribute at all, and it is quicker to restrict the sum somewhat to those terms which contribute the most.

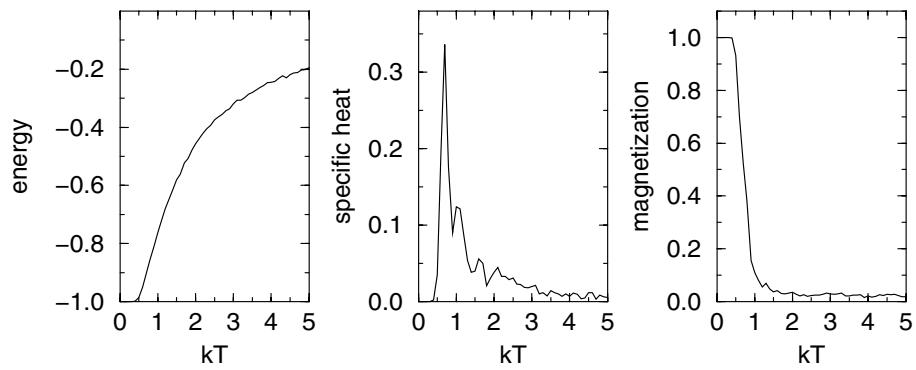
In their simulation of neutron transmission through matter, Metropolis, Rosenbluth, Teller, and Teller [21] invented an algorithm to improve the Monte



**Fig. 12.2** A 1D lattice of 100 spins aligned along the ordinate. Up spins are indicated by circles and down spins by blank spaces. The iteration number (“time”) dependence of the spins is shown along the abscissa. Even though the system starts with all up spins (a “cold” start) the system is seen to form domains as it equilibrates.

Carlo calculation of averages. This *Metropolis algorithm* is now a cornerstone of computational physics. The sequence of configurations it produces (a *Markov chain*) accurately simulates the fluctuations that occur during thermal equilibrium. The algorithm randomly changes the individual spins such that, on the average, the probability of a configuration occurring follows a Boltzmann distribution. (We do not find the proof of this trivial or particularly illuminating.)

The Metropolis algorithm is a combination of the variance reduction technique, discussed in Section 11.7.1, and the von Neumann rejection technique, discussed in Section 11.7.4. There we showed how to make Monte Carlo integration more efficient by sampling random points predominately where the integrand is large, and how to generate random points with an arbitrary probability distribution. Now we would like have spins flip randomly, have a system that can reach any configuration in a finite number of steps (*ergodic sampling*), and have a distribution of energies described by a Boltzmann distribution. (A direct Monte Carlo simulation would take prohibitively long times.) In a sense, the Metropolis algorithm is a simulated version of the annealing process used to make sword blades hard and flexible. Let us say that we are making a blade for a sword and are hammering away at it while it is red hot to get its shape just right. At this high a temperature there is a lot of internal motion and not the long-range order needed for a hard blade. So, as



**Fig. 12.3** Simulation results for the energy, specific heat, and magnetization of a 1D lattice of 100 spins as a function of temperature.

part of the process, we *anneal* the blade; that is, we heat and slow cool it in order to reduce brittleness and increase strength. Too rapid a cooling would not permit long-range equilibration (and ordering) of the blade, which would lead to brittleness.

Use of the Metropolis algorithm is done via a number of steps. We start with a fixed temperature and an initial configuration for the spins and apply the algorithm until thermal equilibrium is reached (equilibration). Then the algorithm generates the statistical fluctuations about equilibrium from which we deduce the thermodynamic quantities such as the internal energy  $U(T)$  and the magnetization  $M(T)$ . After that, the temperature is changed and the whole process is repeated in order to deduce the  $T$  dependence for the thermodynamic quantities. It is the accuracy of the temperature dependences that provides the convincing evidence for the algorithm. Because the possible  $2^N$  configurations can be a very large number, the amount of computer time needed can be very long, even though the program is simple. Typically, a small number  $\simeq 10N$  of iterations is adequate for equilibration.

Here is an outline of the Metropolis algorithm:

1. Start with an arbitrary spin configuration  $\alpha_k = \{s_1, s_2, \dots, s_N\}$ .
2. Generate a trial configuration  $\alpha_{k+1}$ :
  - (a) Pick particle  $i$  randomly.
  - (b) Reverse  $i$ 's spin direction.
3. Calculate the energy  $E(\alpha_{\text{tr}})$  of the trial configuration.
4. If  $E(\alpha_{\text{tr}}) \leq E(\alpha_k)$ , accept by setting  $\alpha_{k+1} = \alpha_{\text{tr}}$ .
5. If  $E(\alpha_{\text{tr}}) > E(\alpha_k)$ , accept with relative probability  $\mathcal{P} = \exp(-\Delta E/kT)$ :

(a) Choose a uniform random  $r_j$   $0 \leq r_i \leq 1$ .

$$(b) \alpha_{k+1} = \begin{cases} \alpha_{\text{tr}} & \text{if } \mathcal{P} \geq r_j \text{ (accept)} \\ \alpha_k & \text{if } \mathcal{P} < r_j \text{ (reject)} \end{cases}$$

The heart of this algorithm is its generation of a random spin configuration  $\alpha_j$  (12.3) with probability

$$\mathcal{P}(\alpha_j) \propto e^{-E(\alpha_j)/kT} \quad (12.10)$$

The technique is a variation of von Neumann rejection (stone throwing of Section 11.4) in which a random *trial* configuration is either accepted or rejected depending upon the value of the Boltzmann factor. Explicitly, the ratio of probabilities for a trial configuration of energy  $E_t$  to that of an initial configuration of energy  $E_i$  is

$$\frac{\mathcal{P}_{\text{tr}}}{\mathcal{P}_i} = e^{-\Delta E/kT} \quad \Delta E = E_{\text{tr}} - E_i \quad (12.11)$$

If the trial configuration has a lower energy ( $\Delta E \leq 0$ ), the relative probability will be greater than one and we accept the trial configuration as the new initial configuration with no further ado. However, if the trial configuration has a higher energy ( $\Delta E > 0$ ), we do not reject out of hand because the system has moved away from its lowest energy state. Instead, we accept it with relative probability  $\mathcal{P}_{\text{tr}}/\mathcal{P}_i = \exp(-\Delta E/kT) < 1$ . To accept a configuration with a probability, we pick a uniform random number between 0 and 1, and if the probability is greater than this number, we accept the trial configuration; if the probability is smaller than the chosen random number, we reject it. (You can remember which way this goes by letting  $E_t \rightarrow \infty$ , in which case the probability  $\rightarrow 0$  and nothing is accepted.) When the trial configuration is rejected, the next configuration is identical to the preceding one.

The key aspect of the Metropolis algorithm is that the weight given to a trial configuration depends on how far it is from the minimum-energy configuration. Those configurations that stray far from the minimum-energy configuration are deemphasized but not completely discarded. By permitting  $\Delta E > 0$ , we are permitting the system to go “uphill” for a while. This deviation away from a direct path to the minimum-energy configuration permits the algorithm to get away from a local minimum and instead find a global one. Its success relies on it not being too quick in “cooling” to the minimum-energy configuration; for this reason the algorithm is sometimes called *simulated annealing*.

How do you start? One possibility is to start with random values of the spins (a “hot” start). Another possibility (Fig. 12.2) is to start with all spins parallel or antiparallel (a “cold start” for positive and negative  $J$ , respectively).

In general, one tries to remove the importance of the starting configuration by letting the calculation “run a while” ( $\simeq 10N$  rearrangements) before calculating the equilibrium thermodynamic quantities. You should get similar results for hot, cold or arbitrary starts, and by taking their average, you remove some of the statistical fluctuations.

### 12.3.1

#### **Metropolis Algorithm Implementation**

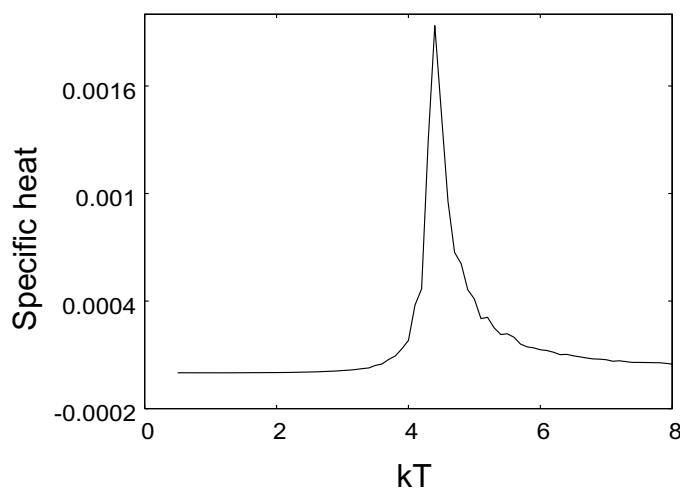
1. Write a program that implements the Metropolis algorithm, that is, that produces a new configuration  $\alpha_{k+1}$  from the present configuration  $\alpha_k$ . (Alternatively, use the program `Ising.java`.)
2. Make the key data structure in your program an array  $s[N]$  containing the values of  $s_i$ . For debugging, print out + and – to give the spins at each lattice point and the trial number.
3. The value for the exchange energy  $J$  fixes the scale for energy. Keep it fixed at  $J = 1$ . (You may also wish to study antiferromagnets with  $J = -1$ , but first examine ferromagnets whose domains are easier to understand.)
4. The thermal energy  $kT$  is in units of  $J$  and is the independent variable. Use  $kT = 1$  for debugging.
5. Use periodic boundary conditions on your chain to minimize end effects. This means that the chain is a circle with the first and last spins adjacent to each other.
6. Try  $N \simeq 20$  for debugging and larger values for production runs.
7. Use the printout to check that the system equilibrates for
  - (a) a totally ordered initial configuration (cold start) and
  - (b) a random initial configuration (hot start).

Your cold start simulation should resemble Fig. 12.2.

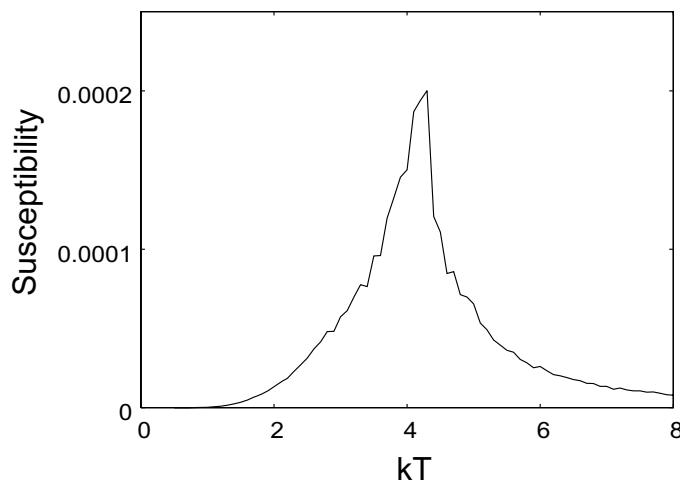
### 12.3.2

#### **Equilibration (Assessment)**

1. Watch a chain of  $N$  atoms attain thermal equilibrium when in contact with a heat bath. At high temperatures, or for small numbers of atoms, you should see large fluctuations, while at lower temperatures you should see smaller fluctuations.



**Fig. 12.4** The specific heat of a 3D Ising lattice as a function of temperature.



**Fig. 12.5** The susceptibility of a 3D Ising lattice as a function of temperature.

2. The largest  $kT$  may be unstable for  $b = 0$  because the system can absorb enough energy to flip all its spin.
3. Note how at thermal “equilibrium” the system is still quite dynamic with spins flipping all the time. It is the energy exchange that determines the thermodynamic properties.
4. You may well find that simulations at small  $kT$  (say that  $kT \simeq 0.1$  for  $N = 200$ ) are slow to equilibrate. Higher  $kT$  values equilibrate faster, yet have larger fluctuations.

5. Observe the formation of domains and the effect they have on the total energy. Regardless of the direction of a spin within a domain, the atom-atom interactions are attractive and so contribute negative amounts to the energy of the system when aligned. However, the  $\uparrow\downarrow$  or  $\downarrow\uparrow$  interactions between domains contribute positive energy. Therefore, you should expect a more negative energy at lower temperatures where there are larger and fewer domains.
6. Make a graph of average domain size versus temperature.

### 12.3.3

#### Thermodynamic Properties (Assessment)

For a given spin configuration  $\alpha_j$ , the energy and magnetization are given by

$$E_j = -J \sum_{i=1}^{N-1} s_i s_{i+1} \quad \mathcal{M}_j = \sum_{i=1}^N s_i \quad (12.12)$$

At high temperatures we expect a random assortment of spins and so a vanishing magnetization. At low temperature we expect  $\mathcal{M}$  to approach  $N/2$  as all the spins get aligned. Although the specific heat (Fig. 12.4) can be computed from the elementary definition

$$C = \frac{1}{N} \frac{dU}{dT} \quad (12.13)$$

doing a numerical differentiation of a fluctuating variable is not expected to be accurate. A better way is to first calculate the fluctuations in energy occurring during  $M$  trials

$$U_2 = \frac{1}{M} \sum_{t=1}^M (E_t)^2 \quad (12.14)$$

and then determine the specific heat from the energy fluctuations:

$$C = \frac{1}{N^2} \frac{U_2 - \langle U \rangle^2}{kT^2} = \frac{1}{N^2} \frac{\langle E^2 \rangle - \langle E \rangle^2}{kT^2} \quad (12.15)$$

1. Extend your program to calculate the internal energy  $U$  and the magnetization  $\mathcal{M}$  for the chain. Incorporate the fact that you do not have to recalculate entire sums for each new configuration because only one spin changes.
2. Make sure to wait for your system to equilibrate before you calculate thermodynamic quantities. (You can check that  $U$  is fluctuating about its average.) Your results should resemble those shown in Fig. 12.3.

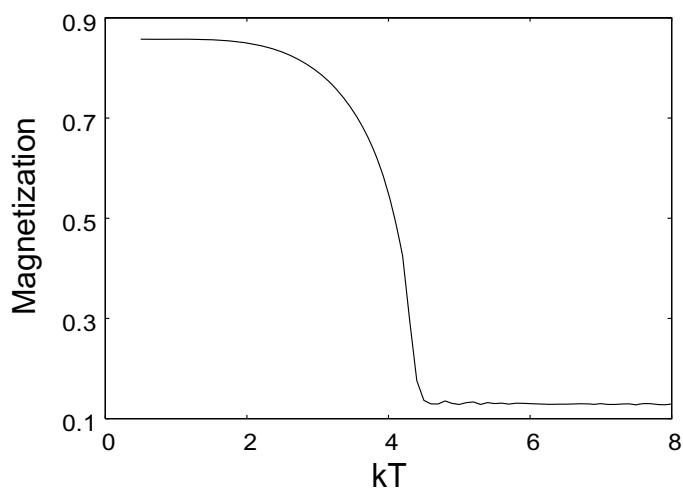


Fig. 12.6 The magnetization of a 3D Ising lattice as a function of temperature.

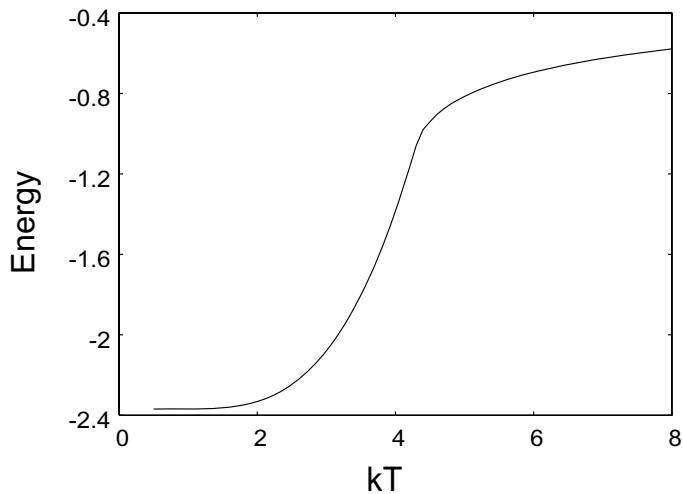


Fig. 12.7 The energy of a 3D Ising lattice as a function of temperature.

3. Reduce the statistical fluctuations by running the simulation a number of times with different seeds, and taking the average of results.
4. The simulations you run for small  $N$  may be realistic but may not agree with statistical mechanics, which assumes  $N \simeq \infty$  (you may assume that  $N \simeq 2000$  is close to infinity). Check that agreement with the analytic results for the thermodynamic limit is better for large  $N$  than small  $N$ .

5. Check that the simulated thermodynamic quantities are independent of initial conditions (within statistical uncertainties). In practice, your cold and hot start results should agree.
6. Make a plot of the internal energy  $U$  as a function of  $kT$  and compare to the analytic result (12.6).
7. Make a plot of the magnetization  $M$  as a function of  $kT$  and compare to the analytic result. Does this agree with how you expect a heated magnet to behave?
8. Compute the fluctuations of the energy  $U_2$  (12.14), and the specific heat  $C$  (12.15). Make a graph of your simulated specific heat compared to the analytic result (12.7).

#### 12.3.4

##### Beyond Nearest Neighbors and 1D (Exploration)

- Extend the model so that the spin–spin interaction (12.4) extends to next-nearest neighbors as well as nearest neighbors. For the ferromagnetic case, this should lead to more binding and less fluctuation because we have increased the couplings among spins and thus increased the thermal inertia.
- Extend the model so that the ferromagnetic spin–spin interaction (12.4) extends to nearest neighbors in two, and for the truly ambitious, then three dimensions (the code `Ising3D.java` is available for instructors). Continue using periodic boundary conditions and keep the number of particles small, at least to start [17].

1. Form a square lattice and place  $\sqrt{N}$  spins on each side.
2. Examine the mean energy and magnetization as the system equilibrates.
3. Is the temperature dependence of the average energy qualitatively different from the 1D model?
4. Identify domains in the printout of spin configurations for small  $N$ .
5. Once your system appears to be behaving properly, calculate the heat capacity and magnetization of the 2D Ising model with the same technique used for the 1D model. Use a total number of particles  $100 \leq N \leq 2000$ .
6. Look for a phase transition from an ordered to unordered configuration by examining the heat capacity and magnetization as a function of temperature. The heat should diverge at the phase transition (you may get only a peak), while the magnetization should vanish above the Curie temperature (Figs. 12.4–12.7).

**Listing 12.1:** Ising.java implements the Metropolis algorithm for a 1D Ising chain.

```
// Ising.java: 1D Ising model with Metropolis algorithm

import java.io.*; // Location of PrintWriter
import java.util.*; // Location of Random

public class Ising {
    public static int N = 1000; // Number of atoms
    public static double B = 1.; // Magnetic field
    public static double mu = .33; // Magnetic moment of atom
    public static double J = .20; // Exchange energy
    public static double k = 1.; // Boltzman constant
    public static double T = 100000000.; // Seed random generator

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {

        Random randnum = new Random(500767); // Seed random generator
        PrintWriter q = new PrintWriter(new FileOutputStream ("ising.dat"), false);

        int i, j, M = 5000; // Number of spin flips
        double[] state = new double[N];
        double[] test = state;
        double ES = energy(state), p, ET; // State, test's energy
                                         // Set initial state
        for ( i=0 ; i < N ; i++ ) state[i] = -1.; // Change state and test
        for ( j=1 ; j <= M ; j++ ) {
            test = state;
            i = (int)(randnum.nextDouble()*(double)N); // Flip random atom
            test[i] *= -1.;
            ET = energy(test);
            p = Math.exp((ES-ET)/(k*T)); // Test trial state
            if ( p >= randnum.nextDouble() ) {
                state = test;
                ES = ET;
            }
            q.println(ES); // Output energy to file
        }
        q.close();
    }

    public static double energy (double[] S) { // Method calc energy
        double FirstTerm = 0., SecondTerm = 0.; // Sum of energy
        int i;
        for ( i=0 ; i <= (N-2) ; i++ ) FirstTerm += S[i]*S[i + 1];
        FirstTerm *= -J;
        for ( i=0 ; i <= (N-1) ; i++ ) SecondTerm += S[i];
        SecondTerm *= -B*mu;
        return (FirstTerm + SecondTerm);
    }
}
```

## 13

# Computer Hardware Basics: Memory and CPU

In this chapter we discuss hardware aspects that are important for high performance computing (HPC). While it may not seem that what we do in this book can be called HPC, history keeps showing that what is HPC today will be on everyone's desktop or laptop in less than a decade. Indeed, the last 30 years have seen a computer progression: scalar → superscalar → vector → parallel, and these are topics that we will talk about here. More recent developments, such as programming for multicore computers, cell computers, and field programmable gate accelerators, will be discussed in future books.

In HPC, you generally modify or "tune" your program to take advantage of a computer's architecture. Often the real problem is to determine which parts of your program get used the most and to decide whether they would run significantly faster if you modified them to take advantage of a computer's architecture. In this chapter we mainly discuss the theory of a high-performance computer's memory and central processor design. In Chap. 13, we concentrate on how you determine the most numerically intensive parts of your program, and how specific hardware features affect them.

Your **problem** is to make a numerically intensive program run faster, but not by porting it to a faster computer. We assume that you are already running your programs on a scientific computer, and so need to know how to make better use of it. In this chapter we discuss the hardware of high-performance computers, while in Chap. 14 we apply that knowledge to your problem.

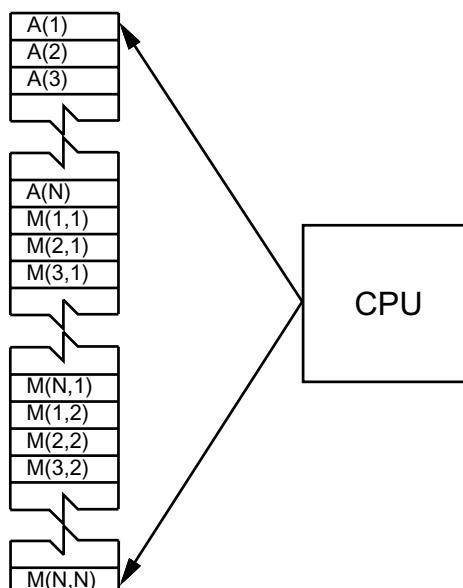
### 13.1

#### High-Performance Computers (CS)

By definition, supercomputers are the fastest and most powerful computers available, and are the superstars of the high-performance class of computers. At this instant, "supercomputers" almost always refer to parallel machines. Unix workstations and modern personal computers (PCs), which are small enough in size and cost to be used by a small group or an individual, yet powerful enough for large-scale scientific and engineering applications, can

also be high-performance computers. We define high-performance computers as machines with good balance among the following major elements:

- multistaged (pipelined) functional units
- multiple central processing units (parallel machines)
- fast, central registers
- very large and fast memories
- very fast communication among functional units
- vector or array processors
- software that integrates the above effectively.



**Fig. 13.1** The logical arrangement of CPU and memory showing a Fortran array,  $A(N)$ , and matrix  $M(N, N)$  loaded into memory.

### 13.1.1 Memory Hierarchy

An idealized model of computer architecture is a CPU sequentially executing a stream of instructions and reading from a continuous block of memory. To illustrate, in Fig. 13.1 we see a vector  $A[]$  and an array  $M[] []$  loaded in memory and about to be processed. The real world is more complicated than

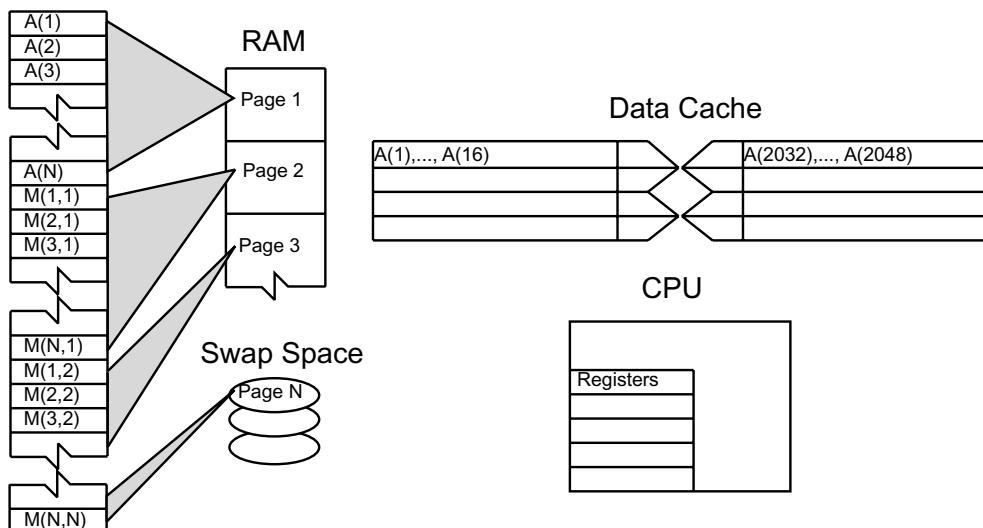
this. First, matrices are not stored in blocks, but rather in linear order. For instance, in Fortran it would be in *column-major* order:

```
M(1,1) M(2,1) M(3,1) M(1,2) M(2,2) M(3,2) M(1,3) M(2,3)
M(3,3),
```

while in Java and C it would be in *row-major* order:

```
M(0,0) M(0,1) M(0,2) M(1,0) M(1,1) M(1,2) M(2,0) M(2,1)
M(2,2).
```

Second, the values for the matrix elements may not even be in the same physical memory (Fig. 13.2). Some may be in RAM, some on the disk, some in cache, and some in the CPU.



**Fig. 13.2** The elements of a computer's memory architecture in the process of handling matrix storage.

To give some of these words more meaning, in Fig. 13.3 we show a simple model of the complex memory architecture of a high-performance computer. This hierarchical arrangement arises from an effort to balance speed and cost, with fast, expensive memory, supplemented by slow, less expensive memory. The memory architecture may include the following elements:

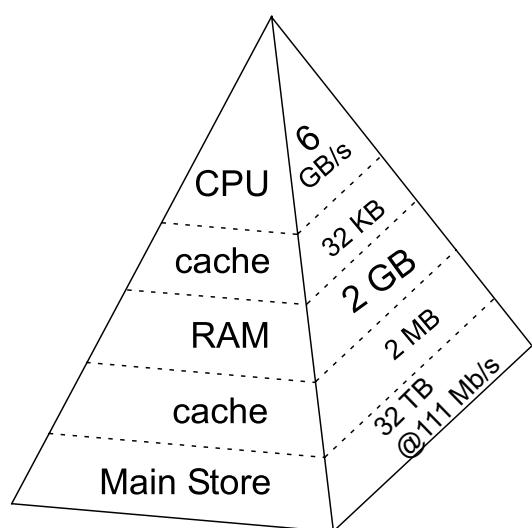
**CPU:** Central processing unit, the fastest part of the computer. The CPU consists of a number of very high-speed memory units called *registers*.

ters, containing the *instructions* sent to the hardware to do things like fetch, store, and operate on data. There are usually separate registers for instructions, addresses, and *operands* (current data). In many cases the CPU also contains some specialized parts for accelerating the processing of floating point numbers, something that used to be done with a separate piece of hardware.

**Cache:** A small, very fast bit of memory also called the *high-speed buffer* holds instructions, addresses, and data in their passage between the very fast CPU registers and the slower main RAM memory. This is seen in the next level down the pyramid in Fig. 13.3. The main memory is also called *dynamic RAM* (DRAM), while the cache is *static RAM* (SRAM). If the cache is used properly, it eliminates the need for the CPU to wait for data to be fetched from memory.

**Cache and data lines:** The data transferred to and from the cache or CPU are grouped into *cache lines* or *data lines*. The time it takes to bring data from memory into cache is called *latency*.

**RAM:** Random access memory or central memory is in the middle memory hierarchy in Fig. 13.3. RAM can be accessed directly (i.e., in random order), and it can be accessed quickly (i.e., without mechanical devices). It is where your program resides while it is being processed.



**Fig. 13.3** Typical memory hierarchy for a single-processor high-performance computer (B = bytes, K, M, G, T = kilo, mega, giga, terra).

**Pages:** Central memory is organized into *pages*, which are blocks of memory of fixed length. The operating system labels and organizes its memory pages much like we do the pages of books; they are numbered and kept track of with a *table of contents*. Typical page sizes are from 4–16 kB.

**Hard disk:** Finally, at the bottom of the memory pyramid is the permanent storage on magnetic disks or optical devices. Although disks are very slow compared to RAM, they can store vast amounts of data and sometimes compensate for their slower speeds by using a cache of their own, the *paging storage controller*.

**Virtual memory:** True to its name, this is a part of memory you will not find in our figures because it is *virtual*. It acts like RAM, but resides on the disk.

When we speak of “fast” and “slow” memory we are using a time scale set by the clock in the CPU. To be specific, if your computer has a clock speed or cycle time of 1 ns, this means that it could perform a billion operations per second, if it could get its hands on the needed data quickly enough (typically, more than 10 cycles are needed to execute a single instruction). While it usually takes 1 cycle to transfer data from the cache into the CPU, the other memory is much slower, and so you can speed your program up by not having the CPU wait for transfers among different levels of memory. Compilers try to do this for you, but their success is affected by your programming style.

As shown in Fig. 13.2 for our example, virtual memory permits your program to use more pages of memory than will physically fit into RAM at one time. A combination of operating system and hardware *maps* this virtual memory into pages with typical lengths of 4–16 kB. Pages not currently in use are stored in the slower memory on the hard disk and brought into fast memory only when needed. The separate memory location for this switching is known as *swap space* (Fig. 13.2). Observe that when the application accesses the memory location for  $M[i][j]$ , the number of the page of memory holding this address is determined by the computer, and the location of  $M[i][j]$  within this page is also determined. A *page fault* occurs if the needed page resides on the disk rather than in RAM. In this case the entire page must be read into memory while the least-recently used page in RAM is swapped onto the disk.

Thanks to virtual memory, it is possible to run programs on small computers that otherwise would require larger machines (or extensive reprogramming). The price you pay for virtual memory is an order-of-magnitude slowdown of your program’s speed when virtual memory is actually invoked. But this may be cheap compared to the time you would have to spend to rewrite

your program so it fits into RAM, or the money you would have to spend to buy a computer with enough RAM for your problem.

Virtual memory also allows *multitasking*, the simultaneous loading into memory of more programs than will physically fit into RAM. Although the ensuing switching among applications uses computing cycles, by avoiding long waits while an application is loaded into memory, multitasking increases the total throughout and permits an improved computing environment for users. For example, it is multitasking that permits windows system to provide us with multiple windows. Even though each window application uses a fair amount of memory, only the single application currently receiving input must actually reside in memory; the rest are *paged out* to disk. This explains why you may notice a slight delay when switching to an idle window; the pages for the now-active program are being placed into RAM and simultaneously the least-used application still in memory is paged out.

## 13.2

### The Central Processing Unit

How does the CPU get to be so fast? Often, it employs *prefetching* and *pipelining*; that is, it has the ability to prepare for the next instruction before the current one has finished. It is like an assembly line or a bucket brigade in which the person filling the buckets at one end of the line does not wait for each bucket to arrive at the other end before filling another bucket. In this same way a processor fetches, reads, and decodes an instruction while another instruction is executing. Consequently, even though it may take more than one cycle to perform some operations, it is possible for data to be entering and leaving the CPU on each cycle. To illustrate, consider how the operation  $c = (a + b) / (d * f)$  is handled (see Tab. 13.1):

**Tab. 13.1** Computation of  $c = (a + b) / (d * f)$ .

Arithmetic unit	Step 1	Step 2	Step 3	Step 4
A1	Fetch $a$	Fetch $b$	Add	—
A2	Fetch $d$	Fetch $f$	Multiply	—
A3	—	—	—	Divide

Here the pipelined arithmetic units A1 and A2 are simultaneously doing their jobs of fetching and operating on operands, yet arithmetic unit A3 must wait for the first two units to complete their tasks before it gets something to do (during which time the other two sit idle).

## 13.2.1

**CPU Design: RISC**

*RISC* is an acronym for Reduced Instruction Set Computer (also called super-scalar). It is a design philosophy for the CPU's architecture developed for high-performance computers and now used broadly. It increases the arithmetic speed of the CPU by decreasing the number of instructions the CPU must follow.

To understand RISC we contrast it with *CISC*, Complex Instruction Set Computers. In the late 1970s, processor designers began to take advantage of Very Large Scale Integration *VLSI* which allowed the placement of hundreds of thousands of devices on a single CPU chip. Much of the space on these early chips was dedicated to *microcode* programs written by the chip designers and containing machine language instructions that set the operating characteristics of the computer. There were over 1000 instructions available, and many were similar to higher level programming languages such as *Pascal* and *Forth*. The price paid for the large number of complex instructions was slow speed, with a typical instruction taking more than 10 clock cycles. Furthermore, a 1975 study by Alexander and Wortman of the *XLP* compiler of the IBM System/360 showed that 10 low-level machine instructions accounted for 80% of the use, while some 30 low-level instructions accounted for 99% of the use.

The RISC philosophy is to have just a small number of instructions available at the chip level, but to have the regular programmer's high-level language, such as Fortran or C, translate them into efficient machine instructions for a particular computer's architecture. This simpler scheme is cheaper to design and produce, lets the processor run faster, and uses the space saved on the chip by cutting down on microcode to increase arithmetic power. Specifically, RISC increases the number of internal CPU registers, thus making it possible to obtain longer pipelines (cache) for the data flow, a significantly lower probability of memory conflict, and some instruction-level parallelism.

The theory behind this philosophy for RISC design is the simple equation describing the execution time of a program

$$CPU\ time = \# \text{ instructions} \times \text{cycles/instruction} \times \text{cycle time} \quad (13.1)$$

Here *CPU time* is the time required by a program; *# instructions* is the total number of machine-level instructions the program requires (sometimes called the *path length*); *cycles/instruction* is the number of CPU clock cycles each instruction requires; and *cycle time* is the actual time it takes for 1 CPU cycle. After viewing (13.1) we can understand the CISC philosophy which tries to reduce *CPU time* by reducing *# instructions*, as well as the RISC philosophy which tries to reduce *CPU time* by reducing the *cycles per instruction* (preferably to one). For RISC to achieve an increase in performance requires a greater

decrease in cycle time and cycles/instruction than the increase in the number of instructions.

In summary, the elements of RISC are:

[**Single-cycle execution**] for most machine-level instructions.

[**Small instruction set**] of less than 100 instructions.

[**Register-based instructions**] operating on values in registers, with memory access confined to load and store to and from registers.

[**Many registers**,] usually more than 32.

[**Pipelining**,] that is, concurrent processing of several instructions.

[**High level compilers**] to improve performance.

### 13.2.2

#### CPU Design: Vector Processor

Often the most demanding part of a scientific computation involves matrix operations. On a classic (von Neumann) scalar computer, the addition of two vectors of physical length 99 to form a third ultimately requires 99 sequential additions (see Tab. 21.1):

**Tab. 13.2** Computation of matrix  $[C] = [A] + [B]$ .

Step 1	Step 2	...	Step 99
$c(1) = a(1) + b(1)$	$c(2) = a(2) + b(2)$	...	$c(99) = a(99) + b(99)$

There is actually much behind-the-scene work here. For each element  $i$  there is the *fetch* of  $a(i)$  from its location in memory, the *fetch* of  $b(i)$  from its location in memory, the addition of the numeric values of these two elements in a CPU register, and the *storage* in memory of the sum into  $c(i)$ . This fetching uses up time and is wasteful in the sense that the computer is being told again and again to do the same thing.

When we speak of a computer doing “vector” processing, we mean that there are hardware components that perform mathematical operations on entire rows or columns of “matrices” as opposed to individual elements. (This hardware also can handle single-subscripted matrices, that is, mathematical vectors.) In *vector* processing of  $[A] + [B] = [C]$ , the successive fetching and additions of the elements of  $A$  and  $B$  get grouped together and overlaid, and

$Z \simeq 64\text{--}256$  elements (the *section size*) are processed with one command (see Tab. 13.3):

**Tab. 13.3** Vector processing of matrix  $[A] + [B] = [C]$ .

Step 1	Step 2	Step 3	...	Step Z
$c(1) = a(1) + b(1)$				
	$c(2) = a(2) + b(2)$			
		$c(3) = a(3) + b(3)$		
			...	
				$c(Z) = a(Z) + b(Z)$

Depending on the array size, this vector processing may speed up the processing of vectors by a factor of about 10. If all  $Z$  elements were truly processed in the same step, then the speedup would be  $\sim 64\text{--}256$ .

Vector processing probably had its heyday during the time when computer manufacturers produced large mainframe computers designed for the scientific and military community. These computers had proprietary hardware and software, and were often so expensive that only corporate or military laboratories could afford them. While the Unix and then PC revolutions have nearly eliminated these large vector machines, some do exist, as well as PCs that use vector processing in their video cards. Who is to say what the future may hold in store?

## 14

# High-Performance Computing: Profiling and Tuning

Recall from Chap. 13 that your **problem** is to make a numerically intensive program run faster by better using your high-performance computer. By running the short implementations given in this chapter you may discover how to do that. In the process you will experiment with your computer's memory and experience some of the concerns, techniques, and rewards of high-performance computing (HPC).

In HPC, you generally modify or "tune" your program to take advantage of a computer's architecture (discussed in Chap. 13). Often the real problem is to determine which parts of your program get used the most and to decide whether they would run significantly faster if you modified them to take advantage of a computer's architecture. In this chapter we concentrate on how you determine the most numerically intensive parts of your program, and how specific hardware features affect them.

Be warned, there is a negative side to high-performance computing. Not only does it take hard work and time to tune a program, but as you optimize a program for a specific piece of hardware and its special software features, you make your program less portable and probably less readable. One school of thought says it is the compiler's, and not the scientist's, job to worry about computer architecture, and it is old-fashioned for you to tune your programs. Yet many computational scientists who run large and complex programs on a variety of machines frequently obtain a 300–500% speedup when they tune their programs for the CPU and memory architecture of a particular machine. You, of course, must decide whether it is worth the effort for the problem at hand; for a program run only once, it is probably not, for an essential tool used regularly, it probably is.

### 14.1

#### Rules for Optimization (Theory)

The type of optimization often associated with *High-Performance* or *Numerically Intensive* computing is one in which sections of a program are rewritten and reorganized in order to increase the program's speed. The overall value

of doing this, especially as computers have become so fast and so available, is often a subject of controversy between computer scientists and computational scientists. Both camps would agree that using the optimization options of the compilers is a good idea. However, computational scientists tend to run large codes with large amounts of data in order to solve real-world problems, and often believe that you cannot rely on the compiler to do all the optimization, especially when you end up with time on your hands waiting for the computer to finish executing your program.

#### 14.1.1

##### **Programming for Virtual Memory (Method)**

While paging makes little appear big, you pay a price because your program's run time increases with each page fault. If your program does not fit into RAM all at once, it will run significantly slower. If virtual memory is shared among multiple programs that run simultaneously, they all cannot have the entire RAM at once, and so there will be memory access *conflicts*, in which case the performance of all programs suffer.

The basic rules for programming for virtual memory are as follows.

1. Do not waste your time worry about reducing the amount of memory used (the *working set size*) unless your program is large. In that case, take a global view of your entire program and optimize those parts that contain the largest arrays.
2. Avoid page faults by organizing your programs to successively perform its calculations on subsets of data, each fitting completely into RAM.
3. Avoid simultaneous calculations in the same program to avoid competition for memory and consequent page faults. Complete each major calculation before starting another.
4. Group data elements close together in memory blocks if they are going to be used together in calculations.

#### 14.1.2

##### **Optimizing Programs; Java vs. Fortran/C**

Many of the optimization techniques developed for Fortran and C are also relevant for Java applications. Yet while Java is a good language for scientific programming and is the most universal and portable of languages, at present Java code runs slower than Fortran or C code, and does not work well if you use MPI for parallel computing (see Chap. 21). In part, this is a consequence of the Fortran and C compilers having been around longer and thereby having been better refined to get the most out of a computer's hardware, and, in part,

this is also a consequence of Java being designed for portability and not speed. Since modern computers are so fast, whether a program takes 1 s or 3 s usually does not matter much, especially in comparison to the hours or days of *your* time that it might take to modify a program for different computers. However, you may want to convert the code to C (whose command structure is very similar to Java) if you are running a computation that takes hours or days to complete and will be doing it many times.

Especially when asked to, Fortran and C compilers will look at your entire code as a single entity and rewrite it for you so that it runs faster. (The rewriting is at a fairly basic level, so there is not much use in your studying the compiler's output as a way of improving your programming skills.) In particular, Fortran and C compilers are very careful with the accessing of arrays in memory. They also are careful with keeping the cache lines full so as not to keep the CPU waiting with nothing to do.

There is no fundamental reason why a program written in Java cannot be compiled to produce an equally efficient code, and indeed such compilers are being developed and becoming available. However, such code is optimized for a particular computer architecture and so are not portable. In contrast, the class file produced by Java is designed to be interpreted or recompiled by the *Java Virtual Machine* (just another program). When you change from a Unix to a Windows computer, to illustrate, the Java Virtual Machine program changes, but the byte code that runs via the Virtual Machine stays the same. This is the essence of Java's portability.

In order to improve the performance of Java, many computers and browsers now run a *Just-In-Time* (JIT) Java compiler. If a JIT is present, the Java Virtual Machine feeds your byte code `Prog.class` to the JIT so that it can be recompiled into native code explicitly tailored to the machine on which you are running. Although there is an extra step involved here, the total time it takes to run your program is usually 10–30 times faster with the JIT, as compared to line-by-line interpretation. Because the JIT is an integral part of the Java Virtual Machine on each operating system, this usually happens automatically.

In the exercises below you will investigate techniques to optimize both a Fortran and a Java program. In the process you will compare the speeds of Fortran *versus* Java for the same computation. If you run your Java code on a variety of machines (easy to do with Java), you should also be able to compare the speed of one computer to another. Note that you can do this exercise even if you do not know Fortran.

## 14.1.3

**Good and Bad Virtual Memory Use (Experiment)**

To see the effect of using virtual memory, run these simple pseudocode examples on your computer. Use a command such as `time` to measure the time being used for each example. These examples call functions `force12` and `force21`. You should write these functions and make them have significant memory requirements for both local and global variables.

**BAD Program, Too Simultaneous**

```
for j = 1, n; {
    for i = 1, n; {
        f12(i,j) = force12( pion(i), pion(j) )           // Fill f12
        f21(i,j) = force21( pion(i), pion(j) )           // Fill f21
        ftot = f12(i,j) + f21(i,j)                         // Fill ftot
    }
}
```

You see that each iteration of the `for` loop requires the data and code for all the functions as well as access to all elements of the matrices and arrays. The working set size of this calculation is the sum of the sizes of the arrays `f12(N,N)`, `f21(N,N)`, and `pion(N)` plus the sums of the sizes of the functions `force12` and `force21`.

A better way to perform the same calculation is to break the calculation into separate components:

**GOOD Program, Separate Loops**

```
for j = 1, n; {
    for i = 1, n; {
        f12(i,j) = force12( pion(i), pion(j) )           // Fill just f12
    }
} for j = 1, n; {
    for i = 1, n; {
        f21(i,j) = force21( pion(i), pion(j) )           // Second nest
                                                       // Fill just f21
    }
} for j = 1, n; {
    for i = 1, n; {
        ftot = f12(i,j) + f21(i,j)                         // Third nest
                                                       // Compute ftot
    }
}
```

Here the separate calculations are independent and the *working set size*, that is, the amount of memory used, is reduced. However, you do pay the additional overhead costs associated with creating extra `for` loops. Because the working set size of the first `for` loop is the sum of the sizes of the arrays `f12(N,N)` and `pion(N)`, and of the function `force12`, we have approximately half the previous size. The size of the last `for` loop is the sum of the sizes for the two

arrays. The working set size of the entire program is the larger of the working set sizes for the different `for` loops.

As an example of the need to group data elements close together in memory or Common blocks if they are going to be used together in calculations, consider the following code:

### BAD Program, Discontinuous Memory

```
Common zed, ylt(9), part(9), zpart1(50000), zpart2(50000), med2(9)
for j = 1, n; {
    ylt(j) = zed * part(j) / med2(9)           // Discontinuous variables
```

Here the variables `zed`, `ylt`, and `part` are used in the same calculations and are adjacent in memory because the programmer grouped them together in Common (global variables). Later, when the programmer realized that the array `med2` was needed, it got tacked onto the end of Common. All the data comprising the variables `zed`, `ylt`, and `part` fit into one page, but the `med2` variable is on a different page because the large array `zpart2(50000)` separates it from the other variables. In fact, the system may be forced to make the entire 4-kB page available in order to fetch the 72 bytes of data in `med2`. While it is difficult for the Fortran or C programmer to assure the placement of variables within page boundaries, you will improve your chances by grouping data elements together:

### GOOD Program, Continuous Memory

```
Common zed, ylt(9), part(9), med2(9), zpart1(50000), zpart2(50000)
for j = 1, n; {
    ylt(j) = zed * part(j) / med2(j)           // Continuous variables
```

#### 14.1.4

### Experimental Effects of Hardware on Performance

We now return to our **problem** of making a numerically intensive program run faster. In this section you conduct an experiment in which you run a complete program in several languages, and on as many computers as you can get your hands on. In this way you explore how a computer's architecture and software affect a program's performance.

Even if you do not know (or care) what is going on inside of a program, some optimizing compilers are smart and caring enough to figure it out for you, and then go about rewriting your program for improved performance. You control how completely the compiler does this when you add on *optimization options* to the compile command:

```
> f90 -O tune.f90
```

Here the `-O` turns on optimization (`O` is the capital letter “oh,” not zero). The actual optimization that gets turned on often differs from compiler to compiler. Fortran and C compilers often have a bevy of such options and directives that lets you truly customize the resulting compiled code. Sometimes optimization options do make the code run faster, sometimes not, and sometimes the faster-running code gives the wrong answers (but does so quickly).

Because computational scientists often spend a good fraction of their time running compiled codes, the compiler options tend to get quite specialized. As a case in point, most compilers provide a number of levels of optimization for the compiler to attempt (there are no guarantees with these things). Although the speedup obtained depends upon the details of the program, higher levels may give greater speedup, as well as a concordant greater risk of being wrong. Some **Forte/Sun** Fortran compiler options, which are rather standard, include the following:

- O Use default optimization level (-O3)
- O1 Provide minimum statement-level optimizations
- O2 Enable basic block-level optimizations
- O3 Add loop unrolling and global optimizations
- O4 Add automatic inlining of routines from same source file
- O5 Attempt aggressive optimizations (with profile feedback)

For the **Visual Fortran (Compaq, Intell)** compiler under windows, options are entered as `/optimize` and for optimization are as follows:

- `/optimize:0` Disable most optimizations
- `/optimize:1` Local optimizations in source program unit
- `/optimize:2` Global optimization, includes `/optimize:1`
- `/optimize:3` Additional global optimizations; speed at cost of code size:  
loop unrolling, instruction scheduling,  
branch code replication, padding arrays for cache
- `/optimize:4` Interprocedure analysis, inlining small procedures
- `/optimize:5` activates loop transformation optimizations

The **gnu compilers** `gcc`, `g77`, `g90` accept `-O` options, as well as specialized ones that include the following:

-malign-double	align doubles on 64-bit boundaries
-ffloat-store	when using IEEE 854 extended precision
-fforce-mem, -fforce-addr	improved loop optimization
-fno-inline	do not compile statement functions inline
-ffast-math	try non-IEEE handling of floats
-funsafe-math-optimizations	speeds up but incorrect results possible
-fno-trapping-math	assume no floating point traps generated
-fstrength-reduce	makes some loops faster
-frerun-cse-after-loop	
-fexpensive-optimizations	
-fdelayed-branch	
-fschedule-insns	
-fschedule-insns2	
-fcaller-saves	
-funroll-loops	unroll iterative do loops
-funroll-all-loops	unroll DO WHILE loops

### 14.1.5

#### Java versus Fortran/C

The various `tune` programs solve the matrix eigenvalue problem

$$\mathbf{H}\mathbf{c} = E\mathbf{c} \quad (14.1)$$

for the eigenvalues  $E$  and eigenvectors  $\mathbf{c}$  of a Hamiltonian matrix  $\mathbf{H}$ . Here the individual Hamiltonian matrix elements are assigned the values

$$H_{ij} = \begin{cases} i, & \text{for } i = j, \\ 0.3^{|i-j|}, & \text{for } i \neq j, \end{cases} = \begin{bmatrix} 1 & 0.3 & 0.09 & 0.027 & \dots \\ 0.3 & 2 & 0.3 & 0.9 & \dots \\ 0.09 & 0.3 & 3 & 0.3 & \dots \\ \ddots & & & & \end{bmatrix} \quad (14.2)$$

Because the Hamiltonian is almost diagonal, the eigenvalues should be close to the values of the diagonal elements and the eigenvectors should be close to  $N$ -dimensional unit vectors. For the present problem, the  $H$  matrix has dimension  $N \times N \simeq 2000 \times 2000 = 4,000,000$ , which means that matrix manipulations should take enough time for you to see the effects of optimization. If your computer has a large supply of central memory, you may need to make the matrix even larger to see what happens when a matrix does not all fit into RAM.

The solution to (14.1) is found via a variation of the *power* or *Davidson method*. We start off with an arbitrary first guess for the eigenvector  $\mathbf{c}$ , which

we represent as the unit, column vector:<sup>1</sup>

$$\mathbf{c}_0 \simeq \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad (14.3)$$

Because the  $\mathbf{H}$  matrix is nearly diagonal with the diagonal element that increase as we move along the diagonal, this guess should be close to the eigenvector with the smallest eigenvalue. We next calculate the energy corresponding to this eigenvector,

$$E = \frac{\mathbf{c}_0^\dagger \mathbf{H} \mathbf{c}_0}{\mathbf{c}_0^\dagger \mathbf{c}_0} \quad (14.4)$$

where  $\mathbf{c}_0^\dagger$  is the row vector adjoint of  $\mathbf{c}_0$ . The heart of the algorithm is the guess that an improved eigenvector has the  $k$ th component

$$\mathbf{c}_1|_k \simeq \mathbf{c}_0|_k + \frac{[\mathbf{H} - EI]\mathbf{c}_0|_k}{E - H_{k,k}} \quad (14.5)$$

where  $k$  ranges over the length of the eigenvector. If repeated, this method converges to the eigenvector with the smallest eigenvalue. It will be the smallest eigenvalue since it gets the largest weight (smallest denominator) in (14.5) each time. For the present case, six places of precision in the eigenvalue is usually obtained after 11 iterations.

- Vary the variable `err` in `tune` that controls precision and note how it affects the number of iterations required.
- Try some variations on the initial guess for the eigenvector (14.5) and see if you can get the algorithm to converge to some of the other eigenvalues.
- Keep a table of your execution times versus technique.
- Compile and execute `tune.f90`, and record the run time. On Unix systems, the compiled program will be placed in the file `a.out`. From a Unix shell, the compilation, timing, and execution can all be done with the commands:

<code>&gt; f90 tune.f90</code>	Fortran compilation
<code>&gt; cc -lm tune.c</code>	C compilation, gcc also likely
<code>&gt; time a.out</code>	Execution

Here the compiled Fortran program is given the (default) name `a.out` and the `time` command gives you the execution (`user`) time and `system` time in seconds to execute `a.out`.

<sup>1</sup> Note that the codes refer to the eigenvector  $c_0$  as `coef`.

- As we indicated in Section 14.1.4, you can ask the compiler to produce a version of your program optimized for speed by including the appropriate compiler option:

```
> f90 -O tune.f90
```

Execute and time the optimized code, checking that it still gives the same answer, and note any speed up in your journal.

- Try out optimization options up to the highest levels and note the run time and accuracy obtained. Usually `-O3` is pretty good, especially for as simple a program as `tune` with only a main method. With only one program unit, we would not expect `-O4` or `-O5` to be an improvement over `-O3`. However, we do expect `-O3`, with its loop unrolling, to be an improvement over `-O2`.
- The program `tune4` does some *loop unrolling* (we will explore that soon). To see the best we can do with Fortran, record the time for the most optimized version of `tune4.f90`.
- The program `Tune.java` is the Java equivalent of the Fortran program `tune.f90`. It is given in Listing 14.1.
- To find an idea of what `Tune.java` does (and give you a feel for how hard life is for the poor computer), assume `ldim = 2` and work through one iteration of `Tune` *by hand*. Assume that the iteration loop has converged, follow the code to completion, and write down the values assigned to the variables.
- Compile and execute `Tune.java`. You do not have to issue the `time` command since we built a timer into the Java program (however there is no harm in trying it). Check that you still get the same answer as you did with Fortran, and note how much longer it takes with Java.
- Try the `-O` option with the Java compiler and note if the speed changes (since this just inlines methods, it should not affect our one-method program).
- You might be surprised how much slower is Java than Fortran and that the Java optimizer does not seem to do much good. To see what the actual Java byte code does, invoke the Java profiler with the command

```
> javap -c Tune
```

This should produce a file `java.prof` for you to look at with an editor. Look at it and see if you agree with us that scientists have better things to do with their time than understand such files!

- We now want to perform a little experiment in which we see what happens to performance as we fill up the computer's memory. In order for this experiment to be reliable, it is best for you to *not* be sharing the computer with any other users. On Unix systems, the `who -a` command will show you the other users (we leave it up to you to figure out how to negotiate with them).

**Listing 14.1:** `Tune.java` is meant to be a numerically intensive enough so as to show the results of various types of optimizations. The program solves the eigenvalue problem iteratively for a nearly diagonal Hamiltonian matrix using a variation of the power method.

```
// Tune.java: eigenvalue solution for performance tuning

public class Tune {

    public static void main(String[] argv) {

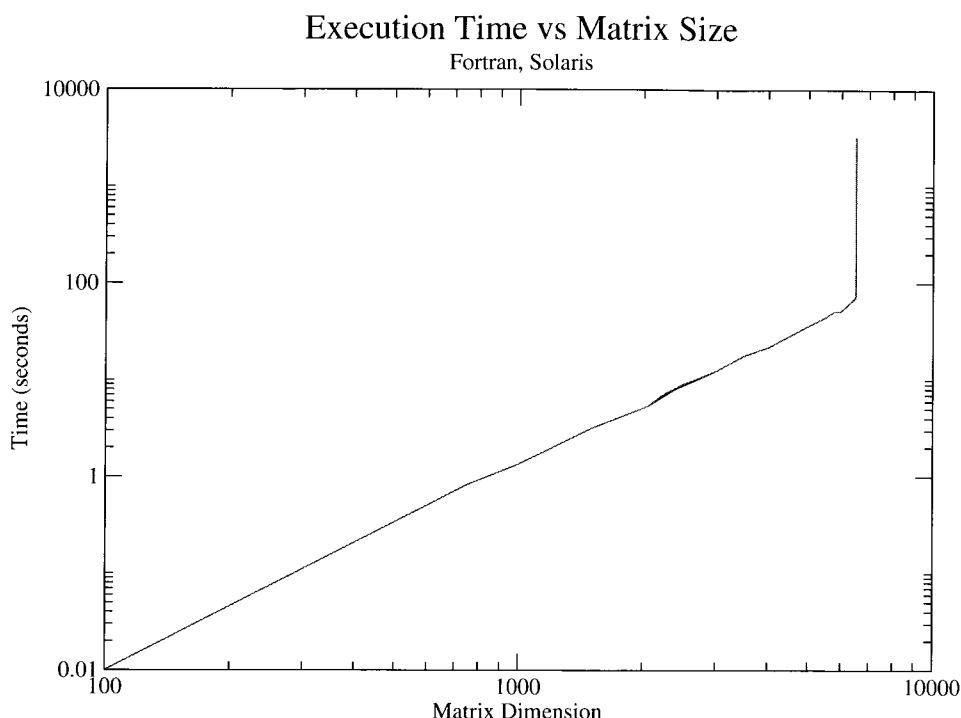
        final int Ldim = 2051;
        int i, j, iter = 0;
        double [][] ham = new double [Ldim] [Ldim];
        double [] coef = new double [Ldim];
        double [] sigma = new double [Ldim];
        double err, ener, ovlp, step = 0., time;

        time = System.currentTimeMillis();           // Initialize time
                                                    // Init matrix & vector
        for ( i = 1; i <= Ldim-1; i++ ) {
            for ( j=1; j <= Ldim-1; j++ ) {
                if (Math.abs(j-i) >10) ham[j][i] = 0. ;
                else ham[j][i] = Math.pow(0.3, Math.abs(j-i));
            }
            ham[i][i] = i ;
            coef[i] = 0.;
        }
        coef[1] = 1.;
        err = 1.;
        iter = 0 ;
                                                    // Start iteration
        while (iter < 15 && err > 1.e-6) {
            iter = iter + 1;
            ener = 0. ;
            ovlp = 0.;                                // Compute E & normalize
            for ( i = 1; i <= Ldim-1; i++ ) {
                ovlp = ovlp + coef[i]*coef[i] ;
                sigma[i] = 0. ;
                for (j=1; j <= Ldim-1; j++)
                    sigma[i] = sigma[i]+coef[j]*ham[j][i];
                ener = ener + coef[i]*sigma[i] ;
            }
            ener = ener/ovlp;
            for ( i = 1; i <= Ldim-1; i++ ) {
                coef[i] = coef[i]/Math.sqrt(ovlp) ;
                sigma[i] = sigma[i]/Math.sqrt(ovlp) ;
            }
            err = 0.;                                // Update
            for ( i = 2; i <= Ldim-1; i++ ) {
                step = (sigma[i] - ener*coef[i])/(ener-ham[i][i]) ;
                coef[i] = coef[i] + step ;
                err = err + step*step ;
            }
            err = Math.sqrt(err) ;
            System.out.println
        }
    }
}
```

```

        ("iter, ener, err " + iter + ", " + ener + ", " + err);
    }
    time = (System.currentTimeMillis() - time)/1000; // Elapsed t
    System.out.println("time = " + time + "s");
}
}

```



**Fig. 14.1** Running time versus matrix size for eigenvalue search using `tune.f90`. Note Fortran's execution time is proportional to the matrix size squared.

- To obtain some idea of what aspect of our little program is making it so slow, compile and run `Tune.java` for the series of matrix sizes `ldim = 10, 100, 250, 500, 750, 1025, 2500, and 3000`. You may get an error message that Java is out of memory at 3000. This is because you have not turned on the use of virtual memory. In Java, the memory allocation pool for your program is called the *heap* and it is controlled by the `-Xms` and `-Xmx` options to the Java interpreter `java`:

`-Xms256m`

Set initial heap size to 256 Mbytes

`-Xmx512m`

Set maximum heap size to 512 Mbytes

- Make a graph of the run time versus matrix size. It should be similar to Figs. 14.1 and 14.2. However, if there are more than one user on your computer

while you run, you may get erratic results. You will note that as our matrix gets larger than  $\sim 1000 \times 1000$  in size, the curve has a sharp increase in slope with execution time, in our case increasing like the *third* power of the dimension. Since the number of elements to compute increases like the *second* power of the dimension, something else is happening here. It is a good guess that the additional slowdown is due to page faults in accessing memory. In particular, accessing 2D arrays, with their elements scattered all through memory, can be very slow.

- Repeat the previous experiment with `tune.f90` that gauges the effect of increasing the `ham` matrix size, only now do it for `ldim = 10, 100, 250, 500, 1025, 3000, 4000, 6000, ...`. You should get a graph like ours. Although our implementation of Fortran has automatic virtual memory, its use will be exceedingly slow, especially for this problem (possibly 50-fold increase in time!). So if you submit your program and you get nothing on the screen (though you can hear the disk spin or see it flash busy), then you are probably in the virtual memory regime. If you can, let the program run for one or two iterations, kill it, and then scale your run time to the time it would have taken for a full computation.
  - To test our hypothesis that the access of the elements in our 2D array `ham [i] [j]` is slowing down the program, we have modified `Tune.java` into `Tune4.java` (Listing 14.2).

**Listing 14.2:** Tune4.java is similar to Tune.java in Listing 14.1, but does some loop unrolling by explicitly writing out two steps of a for loop (which is why the loop can proceed in steps of two). This results in better memory access and consequently faster execution.

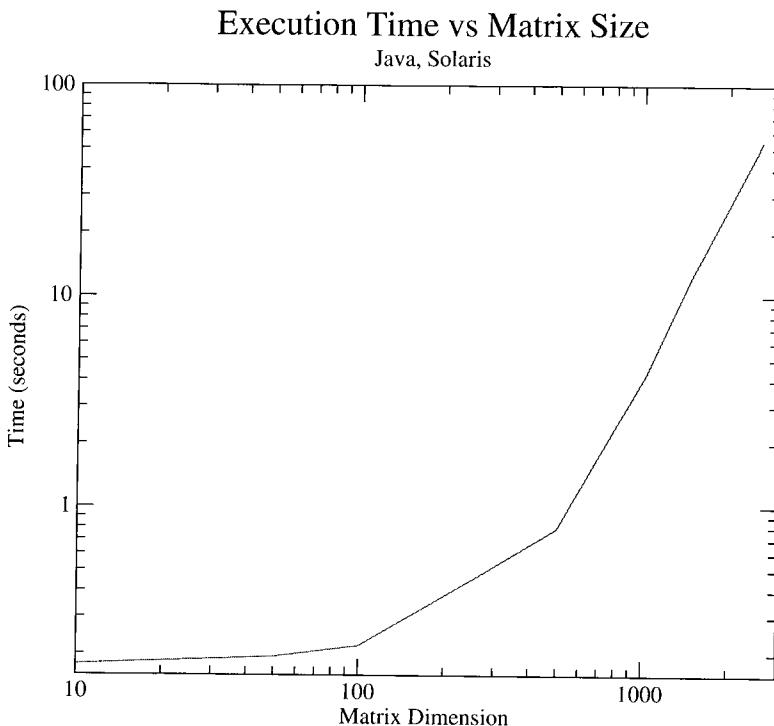
```

for ( i=1; i < Ldim-1; i++ ) {
    ham[i][i] = i ;
    coef[i] = 0. ;
    diag[i] = ham [i][i];
}
coef[1] = 1. ;
err = 1. ;
iter = 0 ;
while (iter < 15 && err > 1.e-6) {
    iter = iter + 1;
    // compute current energy & normalize
    ener = 0. ;
    ovlp1 = 0. ;
    ovlp2 = 0. ;
    for ( i=1; i <= Ldim-2; i = i + 2 ) {
        ovlp1 = ovlp1 + coef[i]*coef[i] ;
        ovlp2 = ovlp2 + coef[i+1]*coef[i+1] ;
        t1 = 0. ;
        t2 = 0. ;
        for ( j=1; j <= Ldim-1; j++ ) {
            t1 = t1 + coef[j]*ham[j][i];
            t2 = t2 + coef[j]*ham[j][i+1];
        }
        sigma[i] = t1;
        sigma[i + 1] = t2;
        ener = ener + coef[i]*t1 + coef[i+1]*t2 ;
    }
    ovlp = ovlp1 + ovlp2 ;
    ener = ener/ovlp;
    fact = 1./Math.sqrt(ovlp);
    coef[1] = fact*coef[1];
    err = 0. ;
    // Update & error norm
    for ( i = 2; i <= Ldim-1; i++ ) {
        t = fact*coef[i];
        u = fact*sigma[i]-ener*t;
        step = u/(ener-diag[i]) ;
        coef[i] = t + step ;
        err = err + step*step ;
    }
    err = Math.sqrt(err) ;
    System.out.println
        ("iter, ener, err "+iter+", " + ener + ", " + err);
}
time = (System.currentTimeMillis() - time)/1000;
System.out.println("time = " + time + "s"); // Elapsed time
}
}

```

- Look at Tune4.java and note where the nested `for` loop over `i` and `j` now takes step of  $\Delta i = 2$  rather than unit steps in `Tune.java`. If things work as expected, the better memory access of `Tune4.java` should cut the runtime nearly in half. Compile and execute `Tune4.java`. Record your answer in your table.

- In order to cut the number of calls to the 2D array in half, we employed a technique known as *loop unrolling* in which we explicitly wrote out some of the lines of code which, otherwise, would be executed implicitly as the `for` loop went through all the values for its counters. This is not as clear a piece of code as before, but, evidently, it permits the compiler to produce a faster executable. To check that `Tune` and `Tune4` actually do the same thing, assume `lDim = 4` and run through one iteration of `Tune4.java` by hand. Hand in your manual trial.



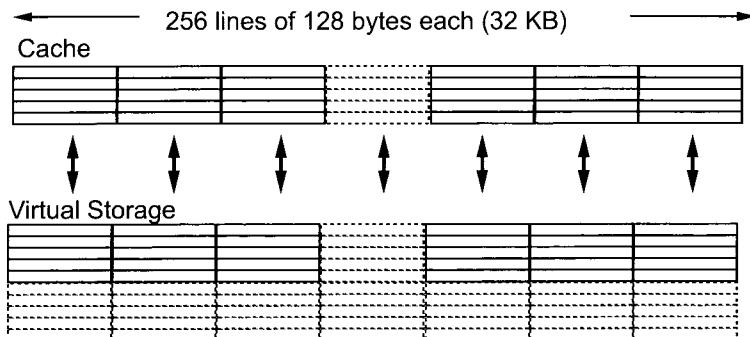
**Fig. 14.2** Running time versus matrix size for eigenvalue search using `Tune.java`. Note how much faster is Fortran (Fig. 14.1), and that for large sizes, the Java execution time varies as the third power of the matrix size. The extra power of size (compared to Fortran) arises from the time spent on reading the matrix elements into and out of memory, something that the Fortran program has been optimized to do rather well.

## 14.2

### Programming for Data Cache (Method)

Data caches are small, very fast memory used as temporary storage between the ultrafast CPU registers and the fast main memory. They have grown in importance as high-performance computers have become more prevalent. On systems that use a data cache, this may well be the single most important programming consideration; continually referencing data that are not in the cache (*cache misses*) may lead to an order-of-magnitude increase in CPU time.

As indicated in Figs. 13.2 and 14.3, the data cache holds a copy of some of the data in memory. The basics are the same for all caches but the sizes are manufacturer dependent. When the CPU tries to address a memory location, the *cache manager* checks to see if the data are in cache. If they are not, the manager reads the data from memory into cache and then the CPU deals with the data directly in cache. The cache manager's view of RAM is shown in Fig. 14.3.



**Fig. 14.3** The cache manager's view of RAM. Each 128-byte cache line is read into one of four lines in cache.

When considering how some matrix operation uses memory, it is important to consider the *stride* of that operation, that is, the number of array elements that get stepped through as an operation repeats. For instance, summing the diagonal elements of a matrix to form the trace

$$\text{Tr}A = \sum_{i=1}^N a(i, i) \quad (14.6)$$

involves a large stride because the diagonal elements are stored far apart for large  $N$ . However, the sum

$$c(i) = x(i) + x(i + 1) \quad (14.7)$$

has stride 1 because adjacent elements of  $x$  are involved. The basic rule in programming for cache is

- Keep the stride low, preferably at 1, which in practice means.
- Vary the leftmost index first on Fortran arrays.
- Vary the rightmost index first on Java and C arrays.

#### 14.2.1

##### **Exercise 1: Cache Misses**

We have said a number of times that your program will be slowed down if the data it needs is in virtual memory and not in RAM. Likewise, your program will also be slowed down if the data required by the CPU is not cache. For high-performance computing, you should write programs that keep as much of the data being processed as possible in cache. To do this you should recall that Fortran matrices are stored in successive memory locations with the row index varying most rapidly (column-major order), while Java and C matrices are stored in successive memory locations with the column index varying most rapidly (row-major order). While it is difficult to isolate the effects of cache from other elements of the computer's architecture, you should now estimate its importance by comparing the time it takes to step through matrix elements row by row, to the time it takes to step through matrix elements column by column.

By actually running on machines available to you, check that these two simple codes with the same number of arithmetic operations will take significantly different times to run because one of them must make large jumps through memory with the memory locations addressed not yet read into cache:

##### **Sequential Column and Row References**

```
for j = 1, 9999; {
    x(j) = m(1,j)
} // Sequential column reference
```

```
for j = 1, 9999; {
    x(j) = m(j,1)
} // Sequential row reference
```

#### 14.2.2

##### **Exercise 2: Cache Flow**

Test the importance of cache flow on your machine by comparing the time it takes to run these two simple programs. Run for increasing column size `idim` and compare the times for loop A versus those for loop B. A computer with very small caches may be most sensitive to stride.

**GOOD f90, BAD Java/C Program; Minimum, Maximum Stride**

```
Dimension Vec(idim, jdim) // Loop A
for j = 1, jdim; {
    for i = 1, idim; {
        Ans = Ans + Vec(i, j) * Vec(i, j) // Stride 1 fetch (f90)
    }
}
```

**BAD f90, GOOD Java/C Program; Maximum, Minimum Stride**

```
Dimension Vec(idim, jdim) // Loop B
for i = 1, idim; {
    for j = 1, jdim; {
        Ans = Ans + Vec(i, j) * Vec(i, j) // Stride jdim fetch (f90)
    }
}
```

Loop A steps through the matrix `vec` in column order. Loop B steps in row order. By changing the size of the columns (the rightmost index for Fortran), we change the size of the step (*stride*) we take through memory in Fortran. Both loops take us through all elements of the matrix, but the stride is different. By increasing the stride in any language, we use fewer elements already present in cache, require additional swapping and loading of cache, and thereby slow down the whole process.

## 14.2.3

**Exercise 3: Large Matrix Multiplication**

As you increase the dimension of the arrays in your program, memory use increases geometrically, and at some point you should be concerned about efficient memory use. The penultimate example of memory usage is large matrix multiplication:

$$[C] = [A] \times [B] \quad (14.8)$$

This involves all the concerns with the different kinds of memory. The natural way to code (14.8) follows from the definition of matrix multiplication:

$$c_{ij} = \sum_{k=1}^N a_{ik} \times b_{kj} \quad (14.9)$$

The sum is over a row of *A* times a column of *B*.

Try out these two codes on your computer. In Fortran, the first code has *B* with stride 1, but *C* with stride *N*. This is cured in the second code by performing the initialization in another loop. In Java and C, the problems are

reversed. On one of our machines , we found a factor of 100 difference in CPU times even though the number of operations is the same!

### BAD f90, GOOD Java/C Program; Maximum, Minimum Stride

```
for i = 1, N; {
    for j = 1, N; {
        c(i, j) = 0.                                // Row
                                                // Column
        for k = 1, N; {
            c(i, j) = c(i, j) + a(i, k) * b(k, j) // Initialize
                                                // Accumulate sum
        }
    }
}
```

### GOOD f90, BAD Java/C Program; Minimum, Maximum Stride

```
for j = 1, N; {                                // Initialization
    for i = 1, N; {
        c(i, j) = 0.0
    }
    for k = 1, N; {
        for i = 1, N; {
            c(i, j) = c(i, j) + a(i, k)*b(k, j)
        }
    }
}
```

## 15

# Differential Equations Applications

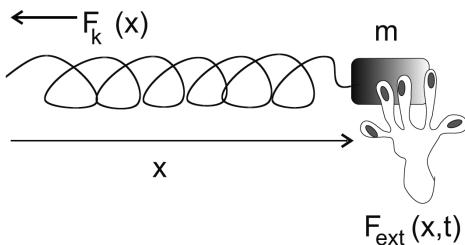
Part of the fascination of computational physics is that we can solve most any differential equation rather easily. Furthermore, while most traditional (read analytic) treatments of oscillations are limited to small displacements about equilibrium, where the restoring forces are linear, we will exceed those limits and look at a range of forces and the motion they cause. In Unit I we look at oscillators that may be harmonic for certain parameter values, but then become anharmonic when the oscillations get large, or for other choices of parameters. We let the reader/instructor decide which oscillator (or both) to study. We start off without any external, time-dependent forces, and spend some time examining how well various differential-equation solvers work. We then include time-dependent forces in order to investigate resonances and beats.

In Chapter 19 we make a related study of the realistic pendulum, and its chaotic behavior. Some special properties of nonlinear equations are discussed in Chapter 26 on solitons. In Unit II we examine how to solve the simultaneous ODEs that arise in projectile and planetary motion.

### 15.1

#### UNIT I. Free Nonlinear Oscillations

**Problems:** In Fig. 15.1 we show a mass  $m$  that is attached to a spring that exerts a restoring force toward the origin, as well as a hand that exerts a time-dependent external force on the mass. We are told that the restoring force exerted by the spring is nonharmonic, that is, not simply proportional to displacement from equilibrium, but we are not given details as to how this is nonharmonic. Your **problem** is to solve for the motion of the mass as a function of time. You may assume that the motion is constrained to one dimension.



**Fig. 15.1** A mass  $m$  attached to a spring with restoring force  $F_k(x)$ , and with an external agency (hand) subjecting the mass to a time-dependent driving force.

## 15.2 Nonlinear Oscillator (Theory)

This is a problem in classical mechanics for which Newton's second law provides us with the equation of motion

$$F_k(x) + F_{\text{ext}}(x, t) = m \frac{d^2x}{dt^2} \quad (15.1)$$

where  $F_k(x)$  is the restoring force exerted by the spring and  $F_{\text{ext}}(x, t)$  is the external force. Equation (15.1) is the differential equation we must solve for arbitrary forces. Because we are not told just how the springs depart from being linear, we are free to try out some different models. As our first model, we try a potential that is a harmonic oscillator for small displacements  $x$ , but also contains a perturbation that introduces a nonlinear term to the force for large  $x$  values:

$$V(x) \simeq \frac{1}{2}kx^2 (1 - \frac{2}{3}\alpha x) \quad (15.2)$$

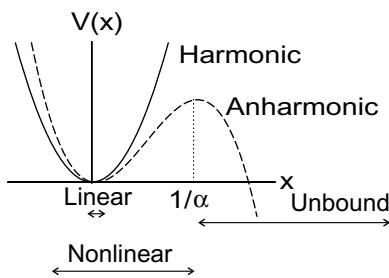
$$\Rightarrow F_k(x) = -\frac{dV(x)}{dx} = -kx(1 - \alpha x) = m \frac{d^2x}{dt^2} \quad (15.3)$$

where we have left off the time-dependent external force. This is the second-order ODE we need to solve. If  $\alpha x \ll 1$ , we should have essentially harmonic motion.

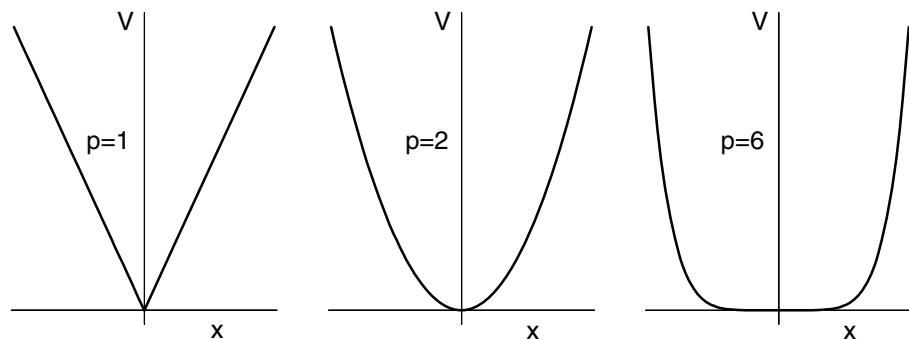
We can understand the basic physics of this model by looking at Fig. 15.2. As long as  $x < 1/\alpha$ , there will be a *restoring force* and the motion will be periodic, even though it is only harmonic (linear) for small-amplitude oscillations. Yet, as the amplitude of oscillation gets larger, there will be an asymmetry in the motion to the right and left of the equilibrium position. And if  $x > 1/\alpha$ , the force becomes repulsive and the mass "rolls" down the potential hill.

As a second model of a nonlinear oscillator, we assume that the spring's potential function is proportional to some arbitrary, *even* power  $p$  of the displacement  $x$  from equilibrium:

$$V(x) = \frac{1}{p}kx^p \quad (p \text{ even}) \quad (15.4)$$



**Fig. 15.2** The potential of an harmonic oscillator (solid curve) and of an oscillator with an anharmonic correction (dashed curve).



**Fig. 15.3** The shape of the potential energy function  $V(x) \propto |x|^p$  for different  $p$  values. The linear and nonlinear labels refer to restoring force derived from these potentials.

We require an even  $p$  to ensure that the force,

$$F_k(x) = -\frac{dV(x)}{dx} = -kx^{p-1} \quad (15.5)$$

contains an odd power of  $p$ , which guarantees that it is a *restoring force* even for negative  $x$ . We display the characteristics of this potential in Fig. 15.3 for  $p = 2$ , the harmonic oscillator, and for  $p = 6$ , which is nearly a square well with the mass moving almost freely until it hits the wall at  $x \approx \pm 1$ . Regardless of the  $p$  value, the motion will be periodic, but it will be harmonic only for  $p = 2$ . Newton's law (15.1) gives the second-order ODE we need to solve

$$F_{\text{ext}}(x, t) - kx^{p-1} = m \frac{d^2x}{dt^2} \quad (15.6)$$

### 15.3

#### Math: Types of Differential Equations

The background material in this section is presented to avoid confusion over semantics. The well-versed student may want to skim or skip it.

- **Order:** A general form for a *first-order* differential equation is

$$\frac{dy}{dt} = f(t, y) \quad (15.7)$$

where the “order” refers to the degree of the derivative on the LHS. The derivative or force function  $f(t, y)$  on the RHS, is arbitrary. For instance, even if  $f(t, y)$  is a nasty function of  $y$  and  $t$  such as

$$\frac{dy}{dt} = -3t^2y + t^9 + y^7 \quad (15.8)$$

this is still first order in the derivative. A general form for a *second-order* differential equation is

$$\frac{d^2y}{dt^2} + \lambda \frac{dy}{dt} = f(t, \frac{dy}{dt}, y) \quad (15.9)$$

The derivative function  $f$  on the RHS is arbitrary and may involve any power of the first derivative as well. To illustrate,

$$\frac{d^2y}{dt^2} + \lambda \frac{dy}{dt} = -3t^2 \left( \frac{dy}{dt} \right)^4 + t^9 y(t) \quad (15.10)$$

is a second-order differential equation as is Newton’s law (15.1).

In the differential equations (15.7) and (15.9), the time  $t$  is the *independent* variable and the position  $y$  is the *dependent* variable. This means that we are free to vary the time at which we want a solution, but not the value of the solution  $y$  at that time. Note that we usually use the symbol  $y$  or  $Y$  for the dependent variable, but that this is just a symbol. In some applications we use  $y$  to describe a position that is an independent variable instead of  $t$ .

- **Ordinary and Partial:** Differential equations such as (15.1) and (15.7) are *ordinary* differential equations because they contain only *one* independent variable, in these cases  $t$ . In contrast, an equation such as the Schrödinger equation,

$$i \frac{\partial \psi(\mathbf{x}, t)}{\partial t} = -\frac{1}{2m} \left[ \frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial^2 \psi}{\partial z^2} \right] + V(\mathbf{x})\psi(\mathbf{x}, t) \quad (15.11)$$

(where we have set  $\hbar = 1$ ) contain several independent variables, and this makes it a *partial differential equations (PDE)*. The partial derivative symbol  $\partial$  is used to indicate that the dependent variable  $\psi$  depends simultaneously on several independent variables. In the early parts of this book we limit ourselves to ordinary differential equations (ODEs). In Chaps. 23–27 we examine a variety of partial differential equations (PDEs).

- **Linear and Nonlinear:** Part of the liberation of computational science is that we are no longer limited to solving *linear equations*. A *linear* equation is

one in which only the first power of  $y$  or  $d^n y / d^n t$  appears; a *nonlinear* equation may contain higher powers. For example

$$\frac{dy}{dt} = g^3(t)y(t) \quad (\text{linear}) \quad (15.12)$$

$$\frac{dy}{dt} = \lambda y(t) - \lambda^2 y^2(t) \quad (\text{nonlinear}) \quad (15.13)$$

An important property of linear equations is the *law of linear superposition* which lets us add solutions together to form new ones. As a case in point, if  $A(t)$  and  $B(t)$  are solutions of the linear equation in (15.12), then

$$y(t) = \alpha A(t) + \beta B(t) \quad (15.14)$$

is also a solution for arbitrary values of the constants  $\alpha$  and  $\beta$ . In contrast, even if we were clever enough to guess that the solution of the nonlinear equation (15.12) is

$$y(t) = \frac{a}{1 + be^{-\lambda t}} \quad (15.15)$$

(which you can verify by substitution), we would go amiss if we tried to obtain a more general solution by adding together two such solutions

$$y_1(t) = \frac{a}{1 + be^{-\lambda t}} + \frac{a'}{1 + b'e^{-\lambda t}} \quad (15.16)$$

(which you can confirm by substitution).

- **Initial and Boundary Conditions:** The general solution of a first-order differential equation always contains one arbitrary constant. A general solution of a second-order differential equation contains two such constants, and so forth. For any specific problem, these constants are fixed by the *initial conditions*. For a first-order equation the sole initial condition is usually the position  $y(t)$  at some time. For a second-order equation the two initial conditions are usually position and velocity at some time. Regardless of how powerful a computer you use, the mathematical fact still remains and you must know the initial conditions in order to solve the problem.

In addition to initial conditions, it is possible to further restrict solutions of differential equations. One such way is by *boundary conditions* or *values* that constrain the solution to have fixed values at the boundaries of the solution space. Problems of this sort are called *eigenvalue problems*. They are so demanding that solutions do not always exist, and even when they do exist, they may require a trial-and-error *search* to find them. In Unit II on ODE eigenvalues, we discuss how to extend the techniques of the present unit to boundary-value problems.

### 15.4

#### Dynamical Form for ODEs (Theory)

A standard form for ODEs, which has found use both in numerical analysis [9] and in classical dynamics [22–24], is to express ODEs of *any order* as  $N$  simultaneous, first-order ODEs:

$$\begin{aligned}\frac{dy^{(0)}}{dt} &= f^{(0)}(t, y^{(i)}) \\ \frac{dy^{(1)}}{dt} &= f^{(1)}(t, y^{(i)}) \\ &\quad \ddots \\ \frac{dy^{(N-1)}}{dt} &= f^{(N-1)}(t, y^{(i)})\end{aligned}\quad (15.17) \quad (15.18) \quad (15.19)$$

where the  $y^{(i)}$  dependence of  $f$  indicates that it may depend on all the components of  $y$ , but not on the derivatives  $dy^{(i)}/dt$ . These equations can be expressed more compactly by use of the  $N$ -dimensional vectors  $\mathbf{y}$  and  $\mathbf{f}$ :

$$d\mathbf{y}(t)/dt = \mathbf{f}(t, \mathbf{y}) \quad (15.20)$$

$$\mathbf{y} = \begin{pmatrix} y^{(0)}(t) \\ y^{(1)}(t) \\ \ddots \\ y^{(N-1)}(t) \end{pmatrix} \quad \mathbf{f} = \begin{pmatrix} f^{(0)}(t, \mathbf{y}) \\ f^{(1)}(t, \mathbf{y}) \\ \ddots \\ f^{(N-1)}(t, \mathbf{y}) \end{pmatrix}$$

The utility of such compact notation is that we can study the properties of the ODEs, as well as develop algorithms to solve them, by dealing with the single equation (15.20) without having to worry about the individual equations. To see how this works, let us express a second-order differential equation, namely, Newton's law,

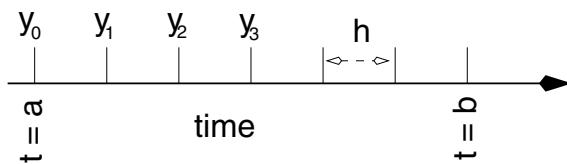
$$\frac{d^2x}{dt^2} = \frac{1}{m}F\left(t, \frac{dx}{dt}, x\right) \quad (15.21)$$

in the standard dynamical form (15.20). The rule is that the RHS may not contain any explicit derivatives, although individual components of  $y^{(i)}$  may represent derivatives. To pull this off, we define the position  $x$  as the dependent variable  $y^{(0)}$  and the velocity  $dx/dt$  as the dependent variable  $y^{(1)}$ :

$$y^{(0)}(t) \stackrel{\text{def}}{=} x(t) \quad y^{(1)}(t) \stackrel{\text{def}}{=} \frac{dx}{dt} = \frac{dy^{(0)}}{dt} \quad (15.22)$$

The second-order ODE (15.21) is now two simultaneous, first-order ODEs,

$$\frac{dy^{(0)}}{dt} = y^{(1)}(t) \quad \frac{dy^{(1)}}{dt} = \frac{1}{m}F(t, y^{(0)}, y^{(1)}) \quad (15.23)$$



**Fig. 15.4** The steps of length  $h$  taken in solving a differential equation. The solution starts at time  $t = a$  and is integrated to  $t = b$ .

This expresses the acceleration (the second derivative in (15.21)) as the first derivative of the velocity [ $y^{(2)}$ ]. These equations are now in the standard form (15.20) with the derivative or force function  $f$  having the two components

$$f^{(0)} = y^{(1)}(t) \quad f^{(1)} = \frac{1}{m}F(t, y^{(0)}, y^{(1)}) \quad (15.24)$$

where  $F$  may be an explicit function of time, as well as of position and velocity.

To be even more specific, we apply these definitions to our spring problem (15.6) to obtain the coupled first-order equations:

$$\frac{dy^{(0)}}{dt} = y^{(1)}(t) \quad \frac{dy^{(1)}}{dt} = \frac{1}{m} [F_{\text{ext}}(x, t) - ky^{(0)}(t)^{p-1}] \quad (15.25)$$

where  $y^{(0)}(t)$  is the position of the mass at time  $t$ , and  $y^{(1)}(t)$  is its velocity. In the standard form, the components of the force/derivative function, and the initial conditions are

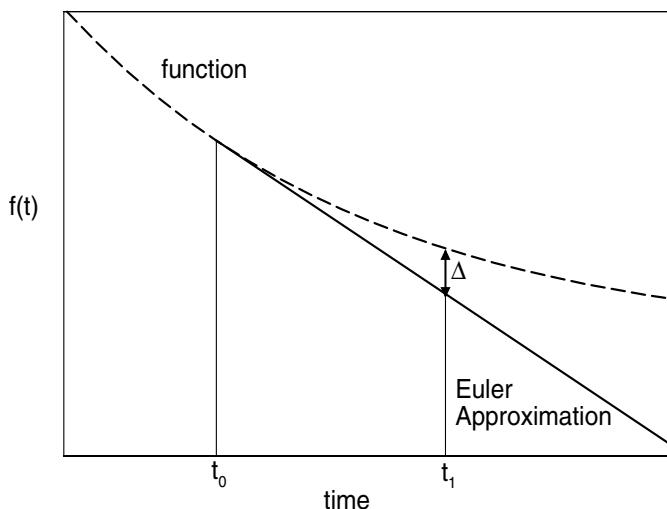
$$\begin{aligned} f^{(0)}(t, \mathbf{y}) &= y^{(1)}(t) & f^{(1)}(t, \mathbf{y}) &= \frac{1}{m} [F_{\text{ext}}(x, t) - k(y^{(0)})^{p-1}] \\ y^{(0)}(0) &= x_0 & y^{(1)}(0) &= v_0 \end{aligned} \quad (15.26)$$

Breaking a second-order differential equation into two first-order ones is not just an arcane mathematical maneuver. In classical dynamics it occurs when transforming the single Newtonian equation of motion involving position and acceleration, (15.1), into two *Hamiltonian* equations involving position and momentum:

$$\frac{dp_i}{dt} = F_i \quad m \frac{dy_i}{dt} = p_i \quad (15.27)$$

## 15.5 ODE Algorithms

The classic way to solve a differential equation is to start with the known initial value of the dependent variable,  $y_0 \equiv y(t = 0)$ , and then use the derivative



**Fig. 15.5** Euler's algorithm for the forward integration of a differential equation by one time step. The linear extrapolation is seen to cause the error  $\Delta$ .

function  $f(t, y)$  to find an approximate value for  $y$  at a small step  $\Delta t = h$  forward in time, that is,  $y(t = h) \equiv y_1$ . Once you can do that, you solve the ODE for all  $t$  values by just continuing stepping to large times, one  $h$  at a time (Fig. 15.4).<sup>1</sup>

It is simplest if the time steps used throughout the integration remain constant in size, and that is mostly what we shall do. Industrial-strength algorithms, such as the one we discuss in Section 15.5.2, adapt the step size by making  $h$  larger in those regions where  $y$  varies slowly (this speeds up the integration and cuts down on the roundoff error), and make  $h$  smaller in those regions where  $y$  varies rapidly (this provides better precision).

Error is always a concern when integrating differential equations because derivatives require small differences, and small differences are prone to subtractive cancellations. In addition, because our stepping procedure for solving the differential equation is a continuous extrapolation of the initial conditions with each step building on a previous extrapolation, this is somewhat like a castle built on sand; in contrast to interpolation, there are no tabulated values on which to anchor your solution.

<sup>1</sup> To avoid confusion, notice that  $y^{(n)}$  is the  $n$ th component of the  $y$  vector, while  $y_n$  is the value of  $y$  after  $n$  time steps. (Yes, there is a price to pay for elegance in notation.)

### 15.5.1

#### Euler's Rule

Euler's rule (Fig. 15.5) is a simple algorithm to integrate the differential equation (15.7) by one step. One simply substitutes the forward difference algorithm for the derivative:

$$\frac{dy(t)}{dt} \simeq \frac{\mathbf{y}(t_{n+1}) - \mathbf{y}(t_n)}{h} = \mathbf{f}(t, \mathbf{y}) \quad (15.28)$$

$$\Rightarrow \mathbf{y}_{n+1} \simeq \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n) \quad (15.29)$$

where  $y_n \stackrel{\text{def}}{=} y(t_n)$  is the value of  $y$  at time  $t_n$ . We know from our discussion of differentiation that the error in the forward difference algorithm is  $\mathcal{O}(h^2)$ , and so this too is the error in Euler's rule.

To indicate the simplicity of this algorithm, we apply it to our oscillator problem for the first time step:

$$y_1^{(0)} = x_0 + v_0 h \quad y_1^{(1)} = v_0 + h \frac{1}{m} [F_{\text{ext}}(t=0) + F_k(t=0)] \quad (15.30)$$

Compare these to the projectile equations familiar from the first-year physics:

$$x = x_0 + v_0 h + \frac{1}{2}ah^2 \quad v = v_0 + ah \quad (15.31)$$

We see that in (15.30) the acceleration does not contribute to the distance covered (no  $h^2$  term), yet it does contribute to the velocity here (and so will contribute belatedly to the distance in the next time step). This is clearly a simple algorithm that requires very small  $h$  values to obtain precision, yet using small values for  $h$  increases the number of steps and the accumulation of the round-off error, which may lead to instability. (Instability is often a problem when you integrate a  $y(t)$  which decreases as the integration proceeds, analogous to upward recursion of spherical Bessel functions. In that case, and if you have a linear problem, you are best off integrating *inward* from large times to small times and then scaling the answer to agree with the initial conditions.) Although we do not recommend Euler's algorithm for general use, it is commonly used to start some of the more sophisticated algorithms.

### 15.5.2

#### Runge–Kutta Algorithm

Although no one algorithm will be good for solving all ODEs, the fourth-order Runge–Kutta algorithm `rk4`, or its extension with adaptive step size, `rk45`, has proven to be robust and capable of industrial-strength work. Although `rk4` is our recommended standard method, we derive the simpler `rk2` and just give the results for `rk4`.

The Runge–Kutta algorithm for integrating a differential equation is based upon the formal integral of our differential equation:

$$\frac{dy}{dt} = f(t, y) \quad \Rightarrow \quad y(t) = \int f(t, y) dt \quad (15.32)$$

$$\Rightarrow \quad y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t, y) dt \quad (15.33)$$

**Listing 15.1:** `rk2.java` solves an ODE with RHS given by the method `f()` using a second-order Runge–Kutta algorithm. Note that the method `f()`, which you will need to change for each problem, is kept separate from the algorithm, which is best not to change.

```
// rk2.java : Runge–Kutta 2nd order ODE solver

import java.io.*;
public class Rk2 {
    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
        PrintWriter w =
            new PrintWriter(new FileOutputStream("rk2.dat"), true);
        double h, t, a = 0., b = 10.; // Endpoints
        double y[] = new double[2], ydumb[] = new double[2];
        double fReturn[] = new double[2];
        double k1[] = new double[2];
        double k2[] = new double[2];
        int i, n=100;
        y[0] = 3.; y[1] = -5.; // Initialize
        h = (b-a)/n;
        t = a;
        System.out.println("rk2 t=" + t + " , x = " + y[0] + " , v = " + y[1]);
        w.println(t + " " + y[0] + " " + y[1]); // Output to file
                                                // Loop over time
        while (t < b) {
            if ((t + h) > b) h = b - t; // The last step
            f(t, y, fReturn); // Evaluate RHS's and return fReturn
            k1[0] = h*fReturn[0]; // Compute function values
            k1[1] = h*fReturn[1];
            for (i=0; i <= 1; i++) ydumb[i] = y[i] + k1[i]/2;
            f(t + h/2, ydumb, fReturn);
            k2[0] = h*fReturn[0];
            k2[1] = h*fReturn[1];
            for (i=0; i <= 1; i++) y[i] = y[i] + k2[i];
            t = t + h;
            System.out.println("rk2 t=" + t + " , x = " + y[0] + " , v = " + y[1]);
            w.println(t + " " + y[0] + " " + y[1]); // Output to file
                                                // End while loop
        }
        // RHS FUNCTION of your choice
        // here
    public static void f(double t, double y[], double fReturn[])
}
```

```
{
  fReturn[0] = y[1];                                // RHS of first equation
  fReturn[1] = -100*y[0]-2*y[1] + 10*Math.sin(3*t);
}
```

To derive the second-order algorithm (`rk2` Listing 15.1), we expand  $f(t, y)$  in a Taylor series about the *midpoint* of the integration interval and retain two terms:

$$f(t, y) \simeq f(t_{n+1/2}, y_{n+1/2}) + (t - t_{n+1/2}) \frac{df}{dt}(t_{n+1/2}) + \mathcal{O}(h^2) \quad (15.34)$$

Because  $(t - t_{n+1/2})$  to any odd power is equally positive and negative over the interval  $t_n \leq t \leq t_{n+1}$ , the integral of  $(t - t_{n+1/2})$  vanishes in (15.33), and we have our algorithm:

$$\int f(t, y) dt \simeq f(t_{n+1/2}, y_{n+1/2})h + \mathcal{O}(h^3) \quad (15.35)$$

$$\Rightarrow y_{n+1} \simeq y_n + hf(t_{n+1/2}, y_{n+1/2}) + \mathcal{O}(h^3) \quad (\text{rk2}) \quad (15.36)$$

So, while `rk2` contains the same number of terms as Euler's rule, it obtains a higher level of precision by taking advantage of the cancellation of the  $\mathcal{O}(h)$  term (likewise, `rk4` has the integral of the  $(t - t_{n+1/2})$  and  $(t - t_{n+1/2})^3$  terms vanish). Yet the price for improved precision is having to evaluate the derivative function and  $y$  at the intermediate time  $t = t_n + h/2$ . And there is the rub, for we cannot use this same algorithm to determine  $y_{n+1/2}$ . The way out of this quandary is to use Euler's algorithm for  $y_{n+1/2}$ :

$$y_{n+1/2} \simeq y_n + \frac{1}{2}h \frac{dy}{dt} = y_n + \frac{1}{2}hf(t_n, y_n) \quad (15.37)$$

Putting the pieces all together gives the complete `rk2` algorithm:

$$\mathbf{y}_{n+1} \simeq \mathbf{y}_n + \mathbf{k}_2 \quad (\text{rk2}) \quad (15.38)$$

$$\mathbf{k}_2 = h\mathbf{f}(t_n + \frac{h}{2}\mathbf{y}_n + \frac{\mathbf{k}_1}{2}) \quad \mathbf{k}_1 = h\mathbf{f}(t_n, \mathbf{y}_n) \quad (15.39)$$

where we use bold font to indicate the vector nature of  $y$  and  $f$ . We see that the known derivative function  $\mathbf{f}$  is evaluated at the ends and midpoint of the interval, but that only the (known) initial value of the dependent variable  $\mathbf{y}$  is required. This makes the algorithm self-starting.

As an example of the use of `rk2`, we apply it to solve the ODE for our spring problem:

$$\begin{aligned} y_1^{(0)} &= y_0^{(0)} + hf^{(0)}(\frac{h}{2}, y_0^{(0)} + k_1) \simeq x_0 + h[v_0 + \frac{h}{2}F_k(0)] \\ y_1^{(1)} &= y_0^{(1)} + hf^{(1)}[\frac{h}{2}, y_0 + \frac{h}{2}f(0, y_0)] \simeq v_0 + \frac{h}{m} \left[ F_{\text{ext}}(\frac{h}{2}) + F_k(y_0^{(1)} + \frac{k_1}{2}) \right]. \end{aligned}$$

These equations say that the position  $y^{(0)}$  changes due to the initial velocity  $v_0$  and the force at time 0, while the velocity changes due to the external force at  $t = h/2$  and the internal force at two intermediate positions. We see that the position now has an  $h^2$  time dependence, which, at last, brings us up to the equation for projectile motion studied in first-year physics.

**Listing 15.2:** `rk4.java` solves an ODE with RHS given by the method `f()` using a fourth-order Runge–Kutta algorithm. Note that the method `f()`, which you will need to change for each problem, is kept separate from the algorithm, which is best not to change.

```
// rk4.java : 4th order Runge–Kutta ODE Solver for arbitrary y(t)

import java.io.*;

public class RK4 {

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {

        PrintWriter w = // Output to file
            new PrintWriter(new FileOutputStream("rk4.dat"), true);

        double h, t;
        double ydumb[] = new double[2];
        double y[] = new double[2];
        double fReturn[] = new double[2];
        double k1[] = new double[2];
        double k2[] = new double[2];
        double k3[] = new double[2];
        double k4[] = new double[2];
        double a = 0.; double b = 10.; // Endpoints
        int i, n = 100;

        y[0] = 3.; y[1] = -5.; // Initialize
        h = (b-a)/n;
        t = a;
        System.out.println("rk4_rhl t = " // Printout for initial step
            + t + ", x = " + y[0] + ", v = " + y[1]);
        w.println(t + " " + y[0] + " " + y[1]); // Output answer to file
                                                // Loop over time
        while (t < b) {
            if ((t + h) > b) h = b - t; // Last step
            f(t, y, fReturn); // Evaluate RHS's, return in fReturn
            k1[0] = h*fReturn[0]; // Compute function values
            k1[1] = h*fReturn[1];
            for (i=0; i <= 1; i++) ydumb[i] = y[i] + k1[i]/2;
            f(t + h/2, ydumb, fReturn);
            k2[0] = h*fReturn[0];
            k2[1] = h*fReturn[1];
            for (i=0; i <= 1; i++) ydumb[i] = y[i] + k2[i]/2;
            f(t + h/2, ydumb, fReturn);
            k3[0] = h*fReturn[0];
            k3[1] = h*fReturn[1];
            for (i=0; i <= 1; i++) ydumb[i] = y[i] + k3[i];
            f(t + h, ydumb, fReturn);
            k4[0] = h*fReturn[0];
```

```

k4[1] = h*fReturn[1];
for (i=0;i <= 1; i++)y[i]=y[i]+(k1[i]+2*(k2[i]+k3[i])+k4[i])/6;
t = t + h;
System.out.println("in Rk4, t = "
                     + t + " , x = " + y[0] + " , v = " + y[1]); // Printout
w.println(t + " " + y[0] + " " + y[1]); // File output
}
} // End while loop

// FUNCTION of your choice here
public static void f( double t, double y[], double fReturn[] )
{
    fReturn[0] = y[1]; // RHS 1st eq
    fReturn[1] = -100*y[0]-2*y[1] + 10*Math.sin(3*t); } // RHS 2nd
}

```

**rk4** The fourth-order Runge–Kutta method `rk4` (Listing 15.2) obtains  $\mathcal{O}(h^4)$  precision by approximating  $y$  as a Taylor series up to  $h^2$  (a parabola) at the midpoint of the interval. This approximation provides an excellent balance of power, precision, and programming simplicity. There are now four gradient ( $k$ ) terms to evaluate with four subroutine calls needed to provide a better approximation to  $f(t, y)$  near the midpoint. This is computationally more expensive than the Euler method, but its precision is much better, and the steps size  $h$  can be made larger. Explicitly, `rk4` requires the evaluation of four intermediate slopes, and these are approximated with the Euler algorithm:

$$\begin{aligned}
y_{n+1} &= y_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) & (15.40) \\
\mathbf{k}_1 &= h\mathbf{f}(t_n, \mathbf{y}_n) & \mathbf{k}_2 &= h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_1}{2}\right) \\
\mathbf{k}_3 &= h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_2}{2}\right) & \mathbf{k}_4 &= h\mathbf{f}(t_n + h, \mathbf{y}_n + \mathbf{k}_3)
\end{aligned}$$

**Listing 15.3:** `rk45.java` solves an ODE with RHS given by the method `f()` using a Runge–Kutta algorithm with adaptive step size that may yield fifth-order precision. Note that the method `f()`, which you will need to change for each problem, is kept separate from the algorithm, which is best not to change.

```

// rk45: Runge–Kutta–Fehlberg adaptive step size ODE solver

import java.io.*;

public class Rk45 {
    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
        PrintWriter w =
            new PrintWriter(new FileOutputStream("rk45_rhl.dat"), true);
        PrintWriter q =
            new PrintWriter(new FileOutputStream("rk45_exact.dat"), true);

        double h, t, s, s1, hmin, hmax, E, Eexact, error;
        double y[] = new double[2];
    }
}

```

```

double fReturn[] = new double[2];
double ydumb[] = new double[2];
double err[] = new double[2];
double k1[] = new double[2];
double k2[] = new double[2];
double k3[] = new double[2];
double k4[] = new double[2];
double k5[] = new double[2];
double k6[] = new double[2];
double Tol = 1.0E-8; // Error tolerance
double a = 0., b = 10.; // Endpoints
int i, j, flops, n = 20;

y[0] = 1.; y[1] = 0.; // Initialize
h = (b-a)/n; // Temp number of steps
hmin = h/64; hmax = h*64; // Min and max step sizes
t = a;
j = 0;
long timeStart = System.nanoTime();
System.out.println(" " + timeStart + " ");
flops = 0;
Eexact = 0.;
error = 0.;
double sum = 0.; // Loop over time
while (t < b) {
    if ((t + h) > b) h = b - t; // Last step
    f(t, y, fReturn); // evaluate both RHS's, return in fReturn
    k1[0] = h*fReturn[0];
    k1[1] = h*fReturn[1];
    for (i=0; i <= 1; i++) ydumb[i] = y[i] + k1[i]/4;
    f(t + h/4, ydumb, fReturn);
    k2[0] = h*fReturn[0];
    k2[1] = h*fReturn[1];
    for (i=0; i <= 1; i++) ydumb[i] = y[i]+3*k1[i]/32 + 9*k2[i]/32;
    f(t + 3*h/8, ydumb, fReturn);
    k3[0] = h*fReturn[0];
    k3[1] = h*fReturn[1];
    for (i=0; i <= 1; i++) ydumb[i] = y[i] + 1932*k1[i]/2197
        -7200*k2[i]/2197. + 7296*k3[i]/2197;
    f(t + 12*h/13, ydumb, fReturn);
    k4[0] = h*fReturn[0];
    k4[1] = h*fReturn[1];
    for (i=0; i <= 1; i++) ydumb[i] = y[i]+439*k1[i]/216 -8*k2[i]
        + 3680*k3[i]/513 -845*k4[i]/4104;
    f(t + h, ydumb, fReturn);
    k5[0] = h*fReturn[0];
    k5[1] = h*fReturn[1];
    for (i=0; i <= 1; i++) ydumb[i] = y[i] -8*k1[i]/27 + 2*k2[i]
        -3544*k3[i]/2565 + 1859*k4[i]/4104 -11*k5[i]/40;
    f(t + h/2, ydumb, fReturn);
    k6[0] = h*fReturn[0];
    k6[1] = h*fReturn[1];
    for (i=0; i <= 1; i++) err[i] = Math.abs( k1[i]/360
        - 128*k3[i]/4275 - 2197*k4[i]/75240 + k5[i]/50. +2*k6[i]/55 );
        // Accept step size
    if (err[0] < Tol || err[1] < Tol || h <= 2*hmin) {

```

```

        for ( i=0; i <= 1; i++ ) y[ i ] = y[ i ] + 25*k1[ i ]/216.
            + 1408*k3[ i ]/2565. + 2197*k4[ i ]/4104. - k5[ i ]/5.;
            t = t + h;
            j++ ;
    }
    if ( err[0]==0 || err[1]==0 ) s = 0;           // Trap division by 0
    else s = 0.84*Math.pow(Tol*h/err[0], 0.25);      // Reduce step
    if ( s < 0.75 && h > 2*hmin ) h /= 2.;          // Increase step
    else if ( s > 1.5 && 2* h < hmax ) h *= 2.;
    flops++;
    E = Math.pow(y[0], 6.) + 0.5*y[1]*y[1];
    Eexact = 1. ;
    error = Math.abs((E-Eexact)/Eexact);
    sum += error;
}
System.out.println(" < error> = " + sum/flops);
System.out.println("flops = " + flops + "");
long timeFinal = System.nanoTime();
System.out.println(" " + timeFinal + "");
System.out.println("Nanoseconds = " + (timeFinal-timeStart));
}

// Enter your own RHS here!
public static void f(double t, double y[], double fReturn[]) {
    fReturn[0] = y[1];
    fReturn[1] = -6.*Math.pow(y[0], 5.);
    return;
}
}

```

**rk45** (Listing 15.3) A variation of `rk4`, known as the Runge–Kutta–Fehlberg method, or `rk45` [25], automatically doubles the step size and tests to see how an estimate of the error changes. If the error is still within acceptable bounds, the algorithm will continue to use the larger step size and thus speed up the computation; if the error is too large, the algorithm will decrease the step size until an acceptable error is found. As a consequence of the extra information obtained in the testing, the algorithm obtains  $\mathcal{O}(h^5)$  precision, but often at the expense of extra computing time. Whether that extra time is recovered by being able to use a larger step size depends upon the application.

### 15.5.3

#### Assessment: rk2 v. rk4 v. rk45

While you are free to do as you please, we do *not* recommend that you write your own `rk4` method, unless you are very careful. We will be using `rk4` for some high precision work, and unless you get every fraction and method call just right, your `rk4` may appear to work well, but still not give all the precision that you should have. We do, regardless, recommend that you write your own `rk2`, as doing so will make it clearer as to how the Runge–Kutta

methods work, but without all the pain. We do give sample `rk2`, `rk4` and `rk45` codes on the CD, and list the latter two in Listings 15.2 and 15.3.

1. Write your own `rk2` method. Design your method for a general ODE; this means making the derivative function  $f(t, x)$  a separate method.
2. Use your `rk2` method as the basis for a program that solves the equation of motion (15.6) or (15.25). Be sure to use double precision to help control subtractive cancellation. Have your program output and plot both the position  $x(t)$  and velocity  $dx/dt$  as functions of time.
3. Once your ODE solver compiles and executes, do a number of things to check that it is working well and to help you pick a reasonable value for the step size  $h$ .
  - (a) Adjust the parameters in your potential so that it corresponds to a pure harmonic oscillator (set  $p = 2$  or  $\alpha = 0$ ). For this case we have an analytic result to compare with:
 
$$x(t) = A \sin(\omega_0 t + \phi) \quad v(t) = \omega_0 A \cos(\omega_0 t + \phi) \quad \omega_0 = \sqrt{k/m}.$$
  - (b) Pick values of  $k$  and  $m$  such that the period  $T = 2\pi/\omega$  is a nice number with which to work (something like  $T = 1$ ).
  - (c) Start with a step size  $h \simeq T/5$ , and make  $h$  smaller and smaller until the solution looks smooth, has a period that remains constant for a large number of periods, and which agrees with the analytic result. (As a general rule of thumb, we suggest that you start with  $h \simeq T/100$ , where  $T$  is a characteristic time for the problem at hand. Here we want you to start with a large  $h$  so that you can see how the solution improves.)
  - (d) Make sure that you have exactly the same initial conditions for the analytic and numeric solutions (zero displacement, nonzero velocity), and then plot the two together. It is good if you cannot tell them apart, yet that only ensures that there is  $\sim 2$  places of agreement.
  - (e) Try different initial velocities, and verify that a *harmonic* oscillator is *isochronous*, that is, its period does *not* change as the amplitude varies.
4. Now that you know you can get a good solution of an ODE with `rk2`, **compare** the solutions obtained with the `rk2`, `rk4`, and `rk45` solvers. You will find methods for all three on the CD, or you can try writing your own (but recall our caveat).

**Tab. 15.1** Comparison of ODE solvers for different equations.

Equation	Method	Initial $h$	No. of flops	Time (ms)	Relative error
(15.41)	rk4	0.01	1000	5.2	$2.2 \times 10^{-8}$
	rk45	1.00	72	1.5	$1.8 \times 10^{-8}$
(15.42)	rk4	0.01	227	8.9	$1.8 \times 10^{-8}$
	rk45	0.1	3143	36.7	$5.7 \times 10^{-11}$

5. Make a table of your comparisons similar to Tab. 15.1. There we compare `rk4` and `rk45` for the two equations,

$$2y y'' + y^2 - y'^2 = 0 \quad (15.41)$$

$$y'' + 6y^5 = 0 \quad (15.42)$$

with initial conditions  $(y(0), y'(0)) = (1, 1)$ . Equation (15.41) yields oscillations with variable frequency, and has an analytic solution with which to compare. Equation (15.42) corresponds to our standard potential (15.4), with  $p = 6$ . Although we have not tuned `rk45`, the table shows that by setting its tolerance parameter to a small enough number, `rk45` will obtain better precision than `rk4`, but that it requires  $\sim 10$  times more floating-point operations, and takes  $\sim 5$  times longer.

## 15.6

### Solution for Nonlinear Oscillations (Assessment)

Use your `rk4` program to study anharmonic oscillations by trying powers in the range  $p = 2\text{--}12$ , or anharmonic strengths in the range  $0 \leq \alpha x \leq 2$ . Do *not* include any time-dependent forces yet. Note that for large values of  $p$  you may need to decrease the step size  $h$  from the value used for the harmonic oscillator because the forces and accelerations get large near the turning points.

1. Check that, regardless of how nonlinear you make the force, the solution remains periodic with constant amplitude and period, for a given initial condition and value of  $p$  or  $\alpha$ . In particular, check that the maximum speed occurs at  $x = 0$  and that the minimum speed occurs at maximum  $x$ . This is all just a consequence of energy conservation.
2. Verify that different initial conditions do indeed lead to different periods (a *nonisochronous* oscillator).
3. Why do the shapes of the oscillations change for different  $p$ 's or  $\alpha$ 's?
4. Devise an algorithm to determine the period of the oscillation by recording times at which the mass passes through the origin. Note that, be-

cause the motion may be asymmetric, you must record at least four times.

5. Construct a graph of the deduced period as a function of initial energy.
6. Verify that as the initial energy approaches  $k/6\alpha^2$ , or as  $p$  gets large, the motion is oscillatory but not harmonic.
7. Verify that at  $E = k/6\alpha^2$ , the motion of the oscillator changes from oscillatory to translational. See how close you can get to this *separatrix*, and verify that at this energy a single oscillation takes an infinite time. (There is no separatrix for the power-law potential.)

#### 15.6.1

##### Precision Assessment: Energy Conservation

We have not explicitly built energy conservation into our ODE solvers. Nonetheless, unless you have explicitly included a frictional force, it follows from Newton's laws that energy must be a constant for all values of  $p$  or  $\alpha$ . That being so, the constancy of energy in your numerical solution is a demanding test.

1. Plot the potential energy  $PE(t) = V[x(t)]$ , the kinetic energy  $KE(t) = mv^2(t)/2$ , and the total energy  $E(t) = KE(t) + PE(t)$ , for hundreds of periods. Comment on the correlation between  $PE(t)$  and  $KE(t)$ , and how it depends on the potential's parameters.
2. Check the long-term *stability* of your solution by plotting

$$-\log_{10} \left| \frac{E(t) - E(t=0)}{E(t=0)} \right| \simeq \text{number places precision}$$

for a large number of periods. Because  $E$  should be independent of time, the numerator is the absolute error in your solution, and when divided by  $E(0)$ , becomes the relative error (say  $10^{-11}$ ). That being the case,  $-\log_{10}$  of the relative error gives you the approximate number of decimal places of precision in your solution. If you cannot achieve 11 or more places, then you need to decrease the value of  $h$  or look for bugs in your program.

3. Because a particle bound by a large  $p$  oscillator is essentially "free" most of the time, you should observe that the average of its kinetic energy over time exceeds its average potential energy. This is actually a physical explanation of the Virial theorem for a power-law potential:

$$\langle KE \rangle = \frac{p}{2} \langle PE \rangle \tag{15.43}$$

Verify that your solution satisfies the Viral theorem. (Those readers who have worked on the perturbed oscillator problem can use this relation to deduce an effective  $p$  value, which should be between 2 and 3.)

## 15.7

### Extensions: Nonlinear Resonances, Beats and Friction

**Problem:** So far, our oscillations have been rather simple. We have ignored friction and assumed that there are no external forces (hands) to influence the system's natural oscillations. Determine

1. how the oscillations change when friction is included,
2. how the resonances and beats of nonlinear oscillators differ from those of linear oscillators, and
3. how introducing friction affects resonances.

#### 15.7.1

##### Friction: Model and Implementation

The real world is full of friction, and it is not all bad. For while it may make it harder to pedal a bike through the wind, it also tends to stabilize oscillations. The simplest models for friction are *static*, *kinetic* or *sliding*, and *viscous* friction:

$$F_f^{(\text{static})} \leq -\mu_s N \quad F_f^{(\text{kinetic})} = -\mu_k N \frac{v}{|v|} \quad F_f^{(\text{viscous})} = -bv \quad (15.44)$$

Here  $N$  is the *normal force*,  $\mu$  and  $b$  are parameters, and  $v$  is the velocity. This model for static friction is clearly appropriate for objects at rest, while the model for kinetic friction is most appropriate for an object sliding on a dry surface. If the surface is lubricated, or if the object is moving through a viscous medium (like air), then a frictional force proportional to velocity is a better model.

1. Modify your code for harmonic oscillations to include the three types of friction modeled in (15.44), and observe how the motion changes from the frictionless case. Note that this means there should be *two* separate simulations, one including static plus kinetic friction, and another for viscous friction.
2. *Hint:* For the simulation with static and kinetic friction, each time the oscillator has  $v = 0$  you need to check that the restoring force exceeds the static force of friction. If not, the oscillation must end at that instant. Check that your simulation terminates at nonzero  $x$  values.

3. For your simulations with viscous friction, investigate the qualitative changes that occur for increasing  $b$  values:

<b>Underdamped</b>	$b < 2m\omega_0$	Oscillation within an decaying envelope
<b>Critically damped</b>	$b = 2m\omega_0$	Nonoscillatory, finite-time decay
<b>Over damped</b>	$b > 2m\omega_0$	Nonoscillatory, infinite-time decay.

### 15.7.2

#### Resonances and Beats: Model and Implementation

All stable physical systems will oscillate if displaced slightly from their rest positions. The frequency  $\omega_0$  with which such a system executes small oscillations about its rest positions is called its *natural frequency*. If an external, sinusoidal force is applied to this system, and if the frequency of the external force equals  $\omega_0$ , then a *resonance* may occur in which the oscillator absorbs energy from the external force, and the amplitude of oscillation increases with time. If the oscillator and the driving force remain in phase over time, then the amplitude will continue to grow, unless there is some mechanism, such as friction or nonlinearities, to limit the growth.

If the frequency of the driving force is close to the natural frequency of the oscillator, then a related phenomenon, known as *beating*, may occur. In this situation, the oscillator acquires an additional amplitude due to the external force which is slightly out of phase with the natural vibration of the oscillator. There then results either constructive or destructive interference between the two oscillations. If the frequency of the driver is very close to the natural frequency, then the resulting motion,

$$x \simeq x_0 \sin \omega t + x_0 \sin \omega_0 t = (2x_0 \cos \frac{\omega - \omega_0}{2}t) \sin \frac{\omega + \omega_0}{2}t \quad (15.45)$$

looks like the natural vibration of the oscillator at the average frequency  $\frac{\omega + \omega_0}{2}$ , yet with an amplitude  $2x_0 \cos \frac{\omega - \omega_0}{2}t$  that varies with the slow *beat frequency*  $\frac{\omega - \omega_0}{2}$ .

### 15.8

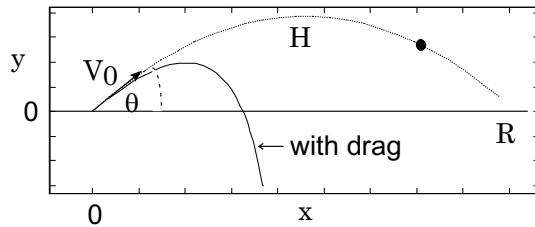
#### Implementation: Inclusion of Time-Dependent Force

To extend our simulation to include an external force,

$$F_{\text{ext}}(t) = F_0 \sin \omega t,$$

we need a force function  $f(t, y)$  with explicit time dependence.

1. Add the time-dependent external force to the space-dependent restoring force in your program (do not include friction yet).
2. Start by using a very large value for the magnitude of the driving force  $F_0$ . This should lead to *mode locking* (the 500-pound gorilla effect), where the system is overwhelmed by the driving force, and, after the transients die out, the system oscillates in phase with the driver regardless of its frequency.
3. Now lower  $F_0$  until it is close to the magnitude of the natural restoring force of the system. You need to have this near equality for beating to occur.
4. For a harmonic system, verify that the “beat frequency,” that is, the number of variations in intensity per unit time, equals the frequency difference  $(\omega - \omega_0)/2\pi$  in cycles per second. You will be able to do this only if  $\omega$  and  $\omega_0$  are close.
5. Once you have a value for  $F_0$  matched well with your system, make a series of runs in which you progressively increase the frequency of the driving force for the range  $\omega_0/10 \leq \omega \leq 10\omega_0$ .
6. Make of plot of the maximum amplitude of oscillation that occurs as a function of the frequency of the driving force.
7. Explore what happens when you make both linear and nonlinear systems resonate. If the nonlinear system is close to being harmonic, you should get beating instead of the blowup that occurs for the linear system. Beating occurs because the natural frequency changes as the amplitude increases, and thus the natural and forced oscillations fall out of phase. Yet once out of phase, the external force stops feeding energy into the system, and amplitude decreases. Yet with the decrease in amplitude, the frequency of the oscillator returns to its natural frequency, the driver and oscillator get back into phase, and the entire cycle repeats.
8. Investigate now, how the inclusion of viscous friction modifies the curve of amplitude versus driving-force frequency. You should find that friction broadens the curve.
9. Notice how the character of the resonance changes as the exponent  $p$  in the potential  $V(x) = k|x|^p/p$  is made larger and larger. At large  $p$ , the mass effectively “hits” the wall and falls out of phase with the driver.



**Fig. 15.6** The trajectory of a projectile fired with initial velocity  $V_0$  in the  $\theta$  direction. The solid curve includes air resistance.

### 15.9

#### UNIT II. Balls, not Planets, Fall Out of the Sky

**Problem:** Golf players and baseball outfielders claim that hit balls appear to fall straight down out of the sky at the end of their trajectories (the solid curve in Fig. 15.6). Your **problem** is to determine whether there is a simple physics explanation for this effect, or whether it is “all in the mind’s eye”. And while you are wondering why things fall out of the sky, see if you can use your new-found numerical tools to explain why planets do not fall out of the sky.

*There are two points to this problem. First, the physics of the motion of projectiles with drag and of planets are interesting and full of surprises. Second, these topics give us the opportunity to extend our ODE solver to two dimensions, where there may be more work for the computer, and there is little more work for us.*

### 15.10

#### Theory: Projectile Motion with Drag

Figure 15.6 shows the initial velocity  $V_0$  and inclination  $\theta$  for a projectile. If we ignore air resistance, the projectile has only the force of gravity acting on it and therefore has a constant acceleration  $g = 9.8 \text{ m/s}^2$  in the negative  $y$  direction. The analytic solutions to the equations of motion are

$$x(t) = V_{0x}t \quad y(t) = V_{0y}t - \frac{1}{2}gt^2 \quad (15.46)$$

$$v_x(t) = V_{0x} \quad v_y(t) = V_{0y} - gt \quad (15.47)$$

where  $(V_{0x}, V_{0y}) = V_0(\cos \theta, \sin \theta)$ . Solving for  $t$  as a function of  $x$ , and substituting it into the  $y(t)$  equation, shows that the trajectory is a parabola:

$$y = \frac{V_{0y}}{V_{0x}}x - \frac{g}{2V_{0x}^2}x^2 \quad (15.48)$$

Likewise, it is easy to show (dotted curve in Fig. 15.6) that without friction the range  $R = 2V_0^2 \sin \theta \cos \theta / g$ , and the maximum height  $H = V_0^2 \sin^2 \theta / (2g)$ .

The parabola of frictionless motion is symmetric about its midpoint, and so does not describe a ball dropping out of the sky. We want to determine if the inclusion of air resistance leads to trajectories that are much steeper at their ends than at their beginnings (solid curve in Fig. 15.6). The basic physics is Newton's second law in two dimensions for a frictional force  $\mathbf{f}^{(f)}$  opposing motion, and a vertical gravitational force  $-mg\hat{\mathbf{e}}_y$ :

$$\mathbf{f}^{(f)} - mg\hat{\mathbf{e}}_y = m \frac{d^2\mathbf{x}(t)}{dt^2} \quad (15.49)$$

$$\Rightarrow f_x^{(f)} = m \frac{d^2x}{dt^2} \quad f_y^{(f)} - mg = m \frac{d^2y}{dt^2}. \quad (15.50)$$

where the bold symbols indicate vector quantities.

The force of friction  $\mathbf{f}^{(f)}$  is not a basic force of nature with a universal form, but rather, it is an approximate model of the physics of viscous flow, with no one expression being accurate for all velocities. We know it always opposes motion, which means it is in a direction opposite to that of the velocity. A simple model for air resistance assumes that the frictional force is proportional to some power  $n$  of the projectile's speed [26, 27]:

$$\mathbf{f}^{(f)} = -k m |v|^n \frac{\mathbf{v}}{|v|} \quad (15.51)$$

where the  $-\mathbf{v}/|v|$  factor ensures that the frictional force is always in a direction opposite to that of the velocity. (If  $n$  is odd, then we may leave off the last factor; however, even  $n$  requires it to ensure the correct sign.) Though a frictional force proportional to a power of the velocity is more accurate of nature than a constant force, it is still a simplification. Indeed, physical measurements indicate that the power  $n$  varies with velocity, and so the most accurate model would be a numerical one that uses the empirical velocity dependence  $n(v)$ .

With a power law for friction, the equations of motion are

$$\frac{d^2x}{dt^2} = -k v_x^n \frac{v_x}{|v|} \quad \frac{d^2y}{dt^2} = -g - k v_y^n \frac{v_y}{|v|} \quad |v| = \sqrt{v_x^2 + v_y^2} \quad (15.52)$$

We shall consider three values for  $n$ , each of which represents a different model for the air resistance: (1)  $n = 1$  for low velocities; (2)  $n = 3/2$ , for medium velocities; and (3)  $n = 2$  for high velocities.

### 15.10.1

#### Simultaneous Second Order ODEs

Even though (15.52) are simultaneous, second-order ODEs, we can still use our regular ODE solver on them after expressing them in standard form

$$\frac{d\mathbf{Y}}{dt} = \mathbf{F}(t, \mathbf{Y}) \quad (\text{standard form}) \quad (15.53)$$

where we use uppercase  $\mathbf{Y}$  and  $\mathbf{F}$  to avoid confusion with the coordinates and frictional force. We pick  $\mathbf{Y}$  to be the 4D vector of dependent variables:

$$Y^{(0)} = x(t) \quad Y^{(1)} = \frac{dx}{dt} \quad Y^{(2)} = y(t) \quad Y^{(3)} = \frac{dy}{dt} \quad (15.54)$$

We express the equations of motion in terms of  $\mathbf{Y}$  to obtain the standard form:

$$\begin{aligned} \frac{dY^{(0)}}{dt} \left( \equiv \frac{dx}{dt} \right) &= Y^{(1)} & \frac{dY^{(1)}}{dt} \left( \equiv \frac{d^2x}{dt^2} \right) &= \frac{1}{m} f_x^{(f)}(\mathbf{y}) \\ \frac{dY^{(2)}}{dt} \left( \equiv \frac{dy}{dt} \right) &= Y^{(3)} & \frac{dY^{(3)}}{dt} \left( \equiv \frac{d^2y}{dt^2} \right) &= \frac{1}{m} f_y^{(f)}(\mathbf{y}) - g \end{aligned}$$

And now we just read off the components of the force function  $\mathbf{F}(t, \mathbf{Y})$ :

$$F^{(0)} = Y^{(1)} \quad F^{(1)} = \frac{1}{m} f_x^{(f)} \quad F^{(2)} = Y^{(3)} \quad F^{(3)} = \frac{1}{m} f_y^{(f)} - g.$$

### 15.10.2

#### Assessment

1. Modify your `rk4` program to solve the simultaneous ODEs for projectile motion, (15.52), for a frictional force with  $n = 1$ , and to plot the results.
2. Check that you get a plot similar to the dotted one shown in Fig. 15.6.
3. In general, it is not possible to compare analytic and numerical results to realistic problems because analytic expressions do not exist. However, we do know the analytic expressions for the frictionless case, and we may turn friction off in our numerical algorithm and then compare the two. This is not a guarantee that we have handled friction correctly, but it is a guarantee that we have something wrong if the comparison fails. Modify your program to also have it compute trajectories with the friction coefficient  $k = 0$ .
4. The model of friction (15.51) with  $n = 1$  is appropriate for low velocities. Modify your program to handle  $n = 3/2$  (medium-velocity friction) and  $n = 2$  (high-velocity friction). To make a realistic comparison, adjust the value of  $k$  for the latter two cases such that the initial force of friction,  $kV_0^n$ , is the same for all.
5. What is your conclusion about balls falling out of the sky?

### 15.11

#### Exploration: Planetary Motion

Newton's explanation of the motion of the planets in terms of a universal law of gravitation is one of the great achievements of science. He was able to prove that the planets traveled along elliptical paths with the sun at one vertex, and predicted periods of motion that agreed with observation. All Newton needed to postulate was that the force between a planet of mass  $m$  and the sun of mass  $M$  is given by

$$f = -\frac{GmM}{r^2} \quad (15.55)$$

Here  $r$  is the distance between the planet of mass  $m$  and sun of mass  $M$ ,  $G$  is a universal constant, and the minus sign indicates that the force is attractive and lies along the line connecting the planet and sun (Fig. 15.7). The hard part for Newton was solving the resulting differential equations, since he had to invent calculus to do it. Whereas the analytic solution is complicated, the numerical solution is not. Even for planets, the basic equation of motion is still

$$\mathbf{f} = m\mathbf{a} = m \frac{d^2\mathbf{x}}{dt^2} \quad (15.56)$$

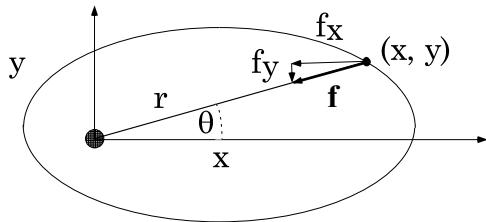
with the force (15.55) having components (Fig. 15.7):

$$f_x = f \cos \theta = f \frac{x}{r} \quad f_y = f \sin \theta = f \frac{y}{r} \quad (r = \sqrt{x^2 + y^2}) \quad (15.57)$$

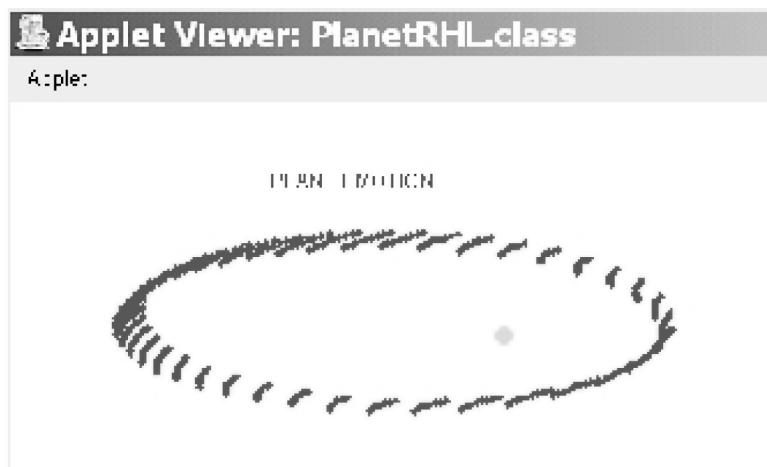
The equation of motion yields two simultaneous, second-order ODEs:

$$\frac{d^2x}{dt^2} = -GM \frac{x}{r^3} \quad \frac{d^2y}{dt^2} = -GM \frac{y}{r^3} \quad (15.58)$$

In Section 15.10.1 we described how to write simultaneous, second-order ODEs in the standard  $\text{rk4}$  form. The same method applies here, but without friction but with the gravitational force now having two, nonconstant components.



**Fig. 15.7** The gravitational force on a planet a distance  $r$  from the sun. The  $x$  and  $y$  components of the force are indicated.



**Fig. 15.8** The precession of a planet's orbit when the gravitational force  $\propto 1/r^4$ .

#### 15.11.1

##### Implementation: Planetary Motion

1. To keep the calculation simple, assume units such that  $GM = 1$ , and that the initial conditions are
 
$$x(0) = 0.5 \quad y(0) = 0 \quad v_x(0) = 0.0 \quad v_y(0) = 1.63.$$
2. Modify your ODE solver program to solve (15.58).
3. Make the number of time steps large enough so that you can see the planet's orbit repeat on top of itself.
4. You may need to make the time step small enough so that the orbit closes upon itself, as it should, and just repeats. This should be a nice ellipse.
5. Experiment with initial conditions until you obtain the ones that produce a circular orbit (a circle is a special case of an ellipse).
6. Once you have good precision, see the effect of progressively increasing the initial velocity until the orbit opens up and becomes a hyperbola.
7. Use the same initial conditions as produced the ellipse, but new investigate the effect of the power in (15.55) being  $1/r^4$  rather than  $1/r^2$ . You should find that the orbital ellipse now rotates or precesses (Fig. 15.8). In fact, as you should verify, even a slight variation from exactly an inverse square power law will cause the orbit to precess.

**15.11.1.1 Exploration: Restricted Three-Body Problem**

Extend the previous solution for planetary motion to one in which a satellite of tiny mass moves under the influence of two planets of equal mass  $M = 1$ . Consider the planets as rotating about their center of mass in circular orbits, and of such large mass that they are uninfluenced by the satellite. Assume that all motions remain in the  $x-y$  plane, and that units are such that  $G = 1$ .

**14****High-Performance Computing: Profiling and Tuning**

Recall from Chap. 13 that your **problem** is to make a numerically intensive program run faster by better using your high-performance computer. By running the short implementations given in this chapter you may discover how to do that. In the process you will experiment with your computer's memory and experience some of the concerns, techniques, and rewards of high-performance computing (HPC).

In HPC, you generally modify or "tune" your program to take advantage of a computer's architecture (discussed in Chap. 13). Often the real problem is to determine which parts of your program get used the most and to decide whether they would run significantly faster if you modified them to take advantage of a computer's architecture. In this chapter we concentrate on how you determine the most numerically intensive parts of your program, and how specific hardware features affect them.

Be warned, there is a negative side to high-performance computing. Not only does it take hard work and time to tune a program, but as you optimize a program for a specific piece of hardware and its special software features, you make your program less portable and probably less readable. One school of thought says it is the compiler's, and not the scientist's, job to worry about computer architecture, and it is old-fashioned for you to tune your programs. Yet many computational scientists who run large and complex programs on a variety of machines frequently obtain a 300–500% speedup when they tune their programs for the CPU and memory architecture of a particular machine. You, of course, must decide whether it is worth the effort for the problem at hand; for a program run only once, it is probably not, for an essential tool used regularly, it probably is.

**14.1****Rules for Optimization (Theory)**

The type of optimization often associated with *High-Performance* or *Numerically Intensive* computing is one in which sections of a program are rewritten and reorganized in order to increase the program's speed. The overall value

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5

of doing this, especially as computers have become so fast and so available, is often a subject of controversy between computer scientists and computational scientists. Both camps would agree that using the optimization options of the compilers is a good idea. However, computational scientists tend to run large codes with large amounts of data in order to solve real-world problems, and often believe that you cannot rely on the compiler to do all the optimization, especially when you end up with time on your hands waiting for the computer to finish executing your program.

#### 14.1.1 Programming for Virtual Memory (Method)

While paging makes little appear big, you pay a price because your program's run time increases with each page fault. If your program does not fit into RAM all at once, it will run significantly slower. If virtual memory is shared among multiple programs that run simultaneously, they all cannot have the entire RAM at once, and so there will be memory access *conflicts*, in which case the performance of all programs suffer.

The basic rules for programming for virtual memory are as follows.

1. Do not waste your time worry about reducing the amount of memory used (the *working set size*) unless your program is large. In that case, take a global view of your entire program and optimize those parts that contain the largest arrays.
2. Avoid page faults by organizing your programs to successively perform its calculations on subsets of data, each fitting completely into RAM.
3. Avoid simultaneous calculations in the same program to avoid competition for memory and consequent page faults. Complete each major calculation before starting another.
4. Group data elements close together in memory blocks if they are going to be used together in calculations.

#### 14.1.2 Optimizing Programs; Java vs. Fortran/C

Many of the optimization techniques developed for Fortran and C are also relevant for Java applications. Yet while Java is a good language for scientific programming and is the most universal and portable of languages, at present Java code runs slower than Fortran or C code, and does not work well if you use MPI for parallel computing (see Chap. 21). In part, this is a consequence of the Fortran and C compilers having been around longer and thereby having been better refined to get the most out of a computer's hardware, and, in part,

this is also a consequence of Java being designed for portability and not speed. Since modern computers are so fast, whether a program takes 1 s or 3 s usually does not matter much, especially in comparison to the hours or days of *your* time that it might take to modify a program for different computers. However, you may want to convert the code to C (whose command structure is very similar to Java) if you are running a computation that takes hours or days to complete and will be doing it many times.

Especially when asked to, Fortran and C compilers will look at your entire code as a single entity and rewrite it for you so that it runs faster. (The rewriting is at a fairly basic level, so there is not much use in your studying the compiler's output as a way of improving your programming skills.) In particular, Fortran and C compilers are very careful with the accessing of arrays in memory. They also are careful with keeping the cache lines full so as not to keep the CPU waiting with nothing to do.

There is no fundamental reason why a program written in Java cannot be compiled to produce an equally efficient code, and indeed such compilers are being developed and becoming available. However, such code is optimized for a particular computer architecture and so are not portable. In contrast, the class file produced by Java is designed to be interpreted or recompiled by the *Java Virtual Machine* (just another program). When you change from a Unix to a Windows computer, to illustrate, the Java Virtual Machine program changes, but the byte code that runs via the Virtual Machine stays the same. This is the essence of Java's portability.

In order to improve the performance of Java, many computers and browsers now run a *Just-In-Time* (JIT) Java compiler. If a JIT is present, the Java Virtual Machine feeds your byte code `Prog.class` to the JIT so that it can be recompiled into native code explicitly tailored to the machine on which you are running. Although there is an extra step involved here, the total time it takes to run your program is usually 10–30 times faster with the JIT, as compared to line-by-line interpretation. Because the JIT is an integral part of the Java Virtual Machine on each operating system, this usually happens automatically.

In the exercises below you will investigate techniques to optimize both a Fortran and a Java program. In the process you will compare the speeds of Fortran *versus* Java for the same computation. If you run your Java code on a variety of machines (easy to do with Java), you should also be able to compare the speed of one computer to another. Note that you can do this exercise even if you do not know Fortran.

## 14.1.3

**Good and Bad Virtual Memory Use (Experiment)**

To see the effect of using virtual memory, run these simple pseudocode examples on your computer. Use a command such as `time` to measure the time being used for each example. These examples call functions `force12` and `force21`. You should write these functions and make them have significant memory requirements for both local and global variables.

**BAD Program, Too Simultaneous**

```
for j = 1, n; {
    for i = 1, n; {
        f12(i,j) = force12( pion(i), pion(j) )           // Fill f12
        f21(i,j) = force21( pion(i), pion(j) )           // Fill f21
        ftot = f12(i,j) + f21(i,j)                         // Fill ftot
    }
}
```

You see that each iteration of the `for` loop requires the data and code for all the functions as well as access to all elements of the matrices and arrays. The working set size of this calculation is the sum of the sizes of the arrays `f12(N,N)`, `f21(N,N)`, and `pion(N)` plus the sums of the sizes of the functions `force12` and `force21`.

A better way to perform the same calculation is to break the calculation into separate components:

**GOOD Program, Separate Loops**

```
for j = 1, n; {
    for i = 1, n; {
        f12(i,j) = force12( pion(i), pion(j) )           // Fill just f12
    }
} for j = 1, n; {
    for i = 1, n; {
        f21(i,j) = force21( pion(i), pion(j) )           // Fill just f21
    }
} for j = 1, n; {
    for i = 1, n; {
        ftot = f12(i,j) + f21(i,j)                         // Compute ftot
    }
}
```

Here the separate calculations are independent and the *working set size*, that is, the amount of memory used, is reduced. However, you do pay the additional overhead costs associated with creating extra `for` loops. Because the working set size of the first `for` loop is the sum of the sizes of the arrays `f12(N,N)` and `pion(N)`, and of the function `force12`, we have approximately half the previous size. The size of the last `for` loop is the sum of the sizes for the two

arrays. The working set size of the entire program is the larger of the working set sizes for the different `for` loops.

As an example of the need to group data elements close together in memory or Common blocks if they are going to be used together in calculations, consider the following code:

#### BAD Program, Discontinuous Memory

```
Common zed, ylt(9), part(9), zpart1(50000), zpart2(50000), med2(9)
for j = 1, n; {
    ylt(j) = zed * part(j) / med2(9)           // Discontinuous variables
```

Here the variables `zed`, `ylt`, and `part` are used in the same calculations and are adjacent in memory because the programmer grouped them together in `Common` (global variables). Later, when the programmer realized that the array `med2` was needed, it got tacked onto the end of `Common`. All the data comprising the variables `zed`, `ylt`, and `part` fit into one page, but the `med2` variable is on a different page because the large array `zpart2(50000)` separates it from the other variables. In fact, the system may be forced to make the entire 4-kB page available in order to fetch the 72 bytes of data in `med2`. While it is difficult for the Fortran or C programmer to assure the placement of variables within page boundaries, you will improve your chances by grouping data elements together:

#### GOOD Program, Continuous Memory

```
Common zed, ylt(9), part(9), med2(9), zpart1(50000), zpart2(50000)
for j = 1, n; {
    ylt(j) = zed * part(j) / med2(j)           // Continuous variables
```

#### 14.1.4

#### Experimental Effects of Hardware on Performance

We now return to our **problem** of making a numerically intensive program run faster. In this section you conduct an experiment in which you run a complete program in several languages, and on as many computers as you can get your hands on. In this way you explore how a computer's architecture and software affect a program's performance.

Even if you do not know (or care) what is going on inside of a program, some optimizing compilers are smart and caring enough to figure it out for you, and then go about rewriting your program for improved performance. You control how completely the compiler does this when you add on *optimization options* to the compile command:

```
> f90 -O tune.f90
```

Here the `-O` turns on optimization (`O` is the capital letter “oh,” not zero). The actual optimization that gets turned on often differs from compiler to compiler. Fortran and C compilers often have a bevy of such options and directives that lets you truly customize the resulting compiled code. Sometimes optimization options do make the code run faster, sometimes not, and sometimes the faster-running code gives the wrong answers (but does so quickly).

Because computational scientists often spend a good fraction of their time running compiled codes, the compiler options tend to get quite specialized. As a case in point, most compilers provide a number of levels of optimization for the compiler to attempt (there are no guarantees with these things). Although the speedup obtained depends upon the details of the program, higher levels may give greater speedup, as well as a concordant greater risk of being wrong. Some **Forte/Sun** Fortran compiler options, which are rather standard, include the following:

- `-O` Use default optimization level (`-O3`)
- `-O1` Provide minimum statement-level optimizations
- `-O2` Enable basic block-level optimizations
- `-O3` Add loop unrolling and global optimizations
- `-O4` Add automatic inlining of routines from same source file
- `-O5` Attempt aggressive optimizations (with profile feedback)

For the **Visual Fortran (Compaq, Intell)** compiler under windows, options are entered as `/optimize` and for optimization are as follows:

- `/optimize:0` Disable most optimizations
- `/optimize:1` Local optimizations in source program unit
- `/optimize:2` Global optimization, includes `/optimize:1`
- `/optimize:3` Additional global optimizations; speed at cost of code size:  
loop unrolling, instruction scheduling,  
branch code replication, padding arrays for cache
- `/optimize:4` Interprocedure analysis, inlining small procedures
- `/optimize:5` activates loop transformation optimizations

The **gnu compilers** `gcc`, `g77`, `g90` accept `-O` options, as well as specialized ones that include the following:

-malign-double	align doubles on 64-bit boundaries
-ffloat-store	when using IEEE 854 extended precision
-fforce-mem, -fforce-addr	improved loop optimization
-fno-inline	do not compile statement functions inline
-ffast-math	try non-IEEE handling of floats
-funsafe-math-optimizations	speeds up but incorrect results possible
-fno-trapping-math	assume no floating point traps generated
-fstrength-reduce	makes some loops faster
-frerun-cse-after-loop	
-fexpensive-optimizations	
-fdelayed-branch	
-fschedule-insns	
-fschedule-insns2	
-fcaller-saves	
-funroll-loops	unroll iterative do loops
-funroll-all-loops	unroll DO WHILE loops

#### 14.1.5

##### Java versus Fortran/C

The various `tune` programs solve the matrix eigenvalue problem

$$\mathbf{H}\mathbf{c} = E\mathbf{c} \quad (14.1)$$

for the eigenvalues  $E$  and eigenvectors  $\mathbf{c}$  of a Hamiltonian matrix  $\mathbf{H}$ . Here the individual Hamiltonian matrix elements are assigned the values

$$H_{i,j} = \begin{cases} i, & \text{for } i = j, \\ 0.3^{|i-j|}, & \text{for } i \neq j, \end{cases} = \begin{bmatrix} 1 & 0.3 & 0.09 & 0.027 & \dots \\ 0.3 & 2 & 0.3 & 0.9 & \dots \\ 0.09 & 0.3 & 3 & 0.3 & \dots \\ \ddots & & & & \end{bmatrix} \quad (14.2)$$

Because the Hamiltonian is almost diagonal, the eigenvalues should be close to the values of the diagonal elements and the eigenvectors should be close to  $N$ -dimensional unit vectors. For the present problem, the  $H$  matrix has dimension  $N \times N \simeq 2000 \times 2000 = 4,000,000$ , which means that matrix manipulations should take enough time for you to see the effects of optimization. If your computer has a large supply of central memory, you may need to make the matrix even larger to see what happens when a matrix does not all fit into RAM.

The solution to (14.1) is found via a variation of the *power* or *Davidson method*. We start off with an arbitrary first guess for the eigenvector  $\mathbf{c}$ , which

we represent as the unit, column vector:<sup>1</sup>

$$\mathbf{c}_0 \simeq \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad (14.3)$$

Because the  $\mathbf{H}$  matrix is nearly diagonal with the diagonal element that increase as we move along the diagonal, this guess should be close to the eigenvector with the smallest eigenvalue. We next calculate the energy corresponding to this eigenvector,

$$E = \frac{\mathbf{c}_0^\dagger \mathbf{H} \mathbf{c}_0}{\mathbf{c}_0^\dagger \mathbf{c}_0} \quad (14.4)$$

where  $\mathbf{c}_0^\dagger$  is the row vector adjoint of  $\mathbf{c}_0$ . The heart of the algorithm is the guess that an improved eigenvector has the  $k$ th component

$$\mathbf{c}_1|_k \simeq \mathbf{c}_0|_k + \frac{[\mathbf{H} - EI]\mathbf{c}_0|_k}{E - H_{k,k}} \quad (14.5)$$

where  $k$  ranges over the length of the eigenvector. If repeated, this method converges to the eigenvector with the smallest eigenvalue. It will be the smallest eigenvalue since it gets the largest weight (smallest denominator) in (14.5) each time. For the present case, six places of precision in the eigenvalue is usually obtained after 11 iterations.

- Vary the variable `err` in `tune` that controls precision and note how it affects the number of iterations required.
- Try some variations on the initial guess for the eigenvector (14.5) and see if you can get the algorithm to converge to some of the other eigenvalues.
- Keep a table of your execution times versus technique.
- Compile and execute `tune.f90`, and record the run time. On Unix systems, the compiled program will be placed in the file `a.out`. From a Unix shell, the compilation, timing, and execution can all be done with the commands:

<code>&gt; f90 tune.f90</code>	Fortran compilation
<code>&gt; cc -lm tune.c</code>	C compilation, <code>gcc</code> also likely
<code>&gt; time a.out</code>	Execution

Here the compiled Fortran program is given the (default) name `a.out` and the `time` command gives you the execution (`user`) time and `system` time in seconds to execute `a.out`.

<sup>1</sup> Note that the codes refer to the eigenvector  $c_0$  as `coef`.

- As we indicated in Section 14.1.4, you can ask the compiler to produce a version of your program optimized for speed by including the appropriate compiler option:

```
> f90 -O tune.f90
```

Execute and time the optimized code, checking that it still gives the same answer, and note any speed up in your journal.

- Try out optimization options up to the highest levels and note the run time and accuracy obtained. Usually `-O3` is pretty good, especially for as simple a program as `tune` with only a main method. With only one program unit, we would not expect `-O4` or `-O5` to be an improvement over `-O3`. However, we do expect `-O3`, with its loop unrolling, to be an improvement over `-O2`.
- The program `tune4` does some *loop unrolling* (we will explore that soon). To see the best we can do with Fortran, record the time for the most optimized version of `tune4.f90`.
- The program `Tune.java` is the Java equivalent of the Fortran program `tune.f90`. It is given in Listing 14.1.
- To find an idea of what `Tune.java` does (and give you a feel for how hard life is for the poor computer), assume `ldim = 2` and work through one iteration of `Tune` by hand. Assume that the iteration loop has converged, follow the code to completion, and write down the values assigned to the variables.
- Compile and execute `Tune.java`. You do not have to issue the `time` command since we built a timer into the Java program (however there is no harm in trying it). Check that you still get the same answer as you did with Fortran, and note how much longer it takes with Java.
- Try the `-O` option with the Java compiler and note if the speed changes (since this just inlines methods, it should not affect our one-method program).
- You might be surprised how much slower is Java than Fortran and that the Java optimizer does not seem to do much good. To see what the actual Java byte code does, invoke the Java profiler with the command

```
> javap -c Tune
```

This should produce a file `java.prof` for you to look at with an editor. Look at it and see if you agree with us that scientists have better things to do with their time than understand such files!

- We now want to perform a little experiment in which we see what happens to performance as we fill up the computer's memory. In order for this experiment to be reliable, it is best for you to *not* be sharing the computer with any other users. On Unix systems, the `who -a` command will show you the other users (we leave it up to you to figure out how to negotiate with them).

**Listing 14.1:** `Tune.java` is meant to be a numerically intensive enough so as to show the results of various types of optimizations. The program solves the eigenvalue problem iteratively for a nearly diagonal Hamiltonian matrix using a variation of the power method.

```
// Tune.java: eigenvalue solution for performance tuning

public class Tune {

    public static void main(String[] argv) {

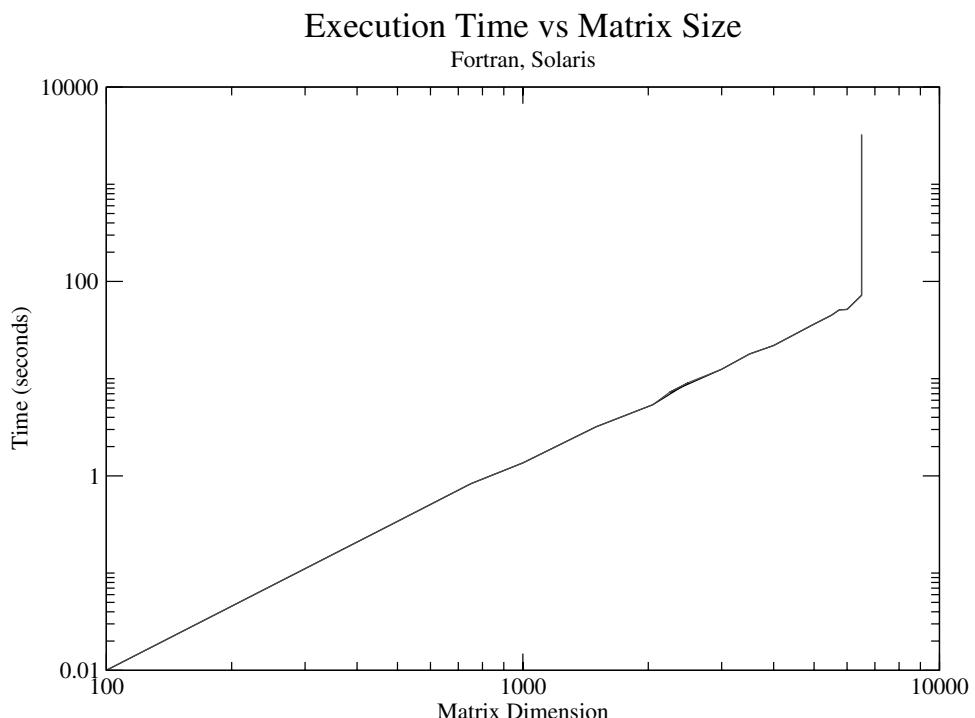
        final int Ldim = 2051;
        int i, j, iter = 0;
        double [][] ham = new double [Ldim] [Ldim];
        double [] coef = new double [Ldim];
        double [] sigma = new double [Ldim];
        double err, ener, ovlp, step = 0., time;

        time = System.currentTimeMillis(); // Initialize time
                                         // Init matrix & vector
        for ( i = 1; i <= Ldim-1; i++ ) {
            for ( j=1; j <= Ldim-1; j++ ) {
                if (Math.abs(j-i) >10) ham[j][i] = 0. ;
                else ham[j][i] = Math.pow(0.3, Math.abs(j-i));
            }
            ham[i][i] = i ;
            coef[i] = 0. ;
        }
        coef[1] = 1. ;
        err = 1. ;
        iter = 0 ;
                                         // Start iteration
        while (iter < 15 && err > 1.e-6) {
            iter = iter + 1;
            ener = 0. ;
            ovlp = 0. ;
                                         // Compute E & normalize
            for ( i = 1; i <= Ldim-1; i++ ) {
                ovlp = ovlp + coef[i]*coef[i] ;
                sigma[i] = 0. ;
                for (j= 1; j <= Ldim-1; j++)
                    sigma[i] = sigma[i]+coef[j]*ham[j][i];
                ener = ener + coef[i]*sigma[i] ;
            }
            ener = ener/ovlp;
            for ( i = 1; i <= Ldim-1; i++ ) {
                coef[i] = coef[i]/Math.sqrt(ovlp) ;
                sigma[i] = sigma[i]/Math.sqrt(ovlp) ;
            }
            err = 0. ;
                                         // Update
            for ( i = 2; i <= Ldim-1; i++ ) {
                step = (sigma[i] - ener*coef[i])/(ener-ham[i][i]) ;
                coef[i] = coef[i] + step ;
                err = err + step*step ;
            }
            err = Math.sqrt(err) ;
            System.out.println
        }
    }
}
```

```

        ("iter, ener, err " + iter + ", " + ener + ", " + err);
    }
    time = (System.currentTimeMillis() - time)/1000; // Elapsed t
    System.out.println("time = " + time + "s");
}
}

```



**Fig. 14.1** Running time versus matrix size for eigenvalue search using `tune.f90`. Note Fortran's execution time is proportional to the matrix size squared.

- To obtain some idea of what aspect of our little program is making it so slow, compile and run `Tune.java` for the series of matrix sizes `lDim = 10, 100, 250, 500, 750, 1025, 2500, and 3000`. You may get an error message that Java is out of memory at 3000. This is because you have not turned on the use of virtual memory. In Java, the memory allocation pool for your program is called the *heap* and it is controlled by the `-Xms` and `-Xmx` options to the Java interpreter `java`:

`-Xms256m`  
`-Xmx512m`

Set initial heap size to 256 Mbytes  
Set maximum heap size to 512 Mbytes

- Make a graph of the run time versus matrix size. It should be similar to Figs. 14.1 and 14.2. However, if there are more than one user on your computer

while you run, you may get erratic results. You will note that as our matrix gets larger than  $\sim 1000 \times 1000$  in size, the curve has a sharp increase in slope with execution time, in our case increasing like the *third* power of the dimension. Since the number of elements to compute increases like the *second* power of the dimension, something else is happening here. It is a good guess that the additional slowdown is due to page faults in accessing memory. In particular, accessing 2D arrays, with their elements scattered all through memory, can be very slow.

- Repeat the previous experiment with `tune.f90` that gauges the effect of increasing the `ham` matrix size, only now do it for `ldim = 10, 100, 250, 500, 1025, 3000, 4000, 6000, ...`. You should get a graph like ours. Although our implementation of Fortran has automatic virtual memory, its use will be exceedingly slow, especially for this problem (possibly 50-fold increase in time!). So if you submit your program and you get nothing on the screen (though you can hear the disk spin or see it flash busy), then you are probably in the virtual memory regime. If you can, let the program run for one or two iterations, kill it, and then scale your run time to the time it would have taken for a full computation.
  - To test our hypothesis that the access of the elements in our 2D array `ham [i][j]` is slowing down the program, we have modified `Tune.java` into `Tune4.java` (Listing 14.2).

**Listing 14.2:** `Tune4.java` is similar to `Tune.java` in Listing 14.1, but does some loop unrolling by explicitly writing out two steps of a `for` loop (which is why the loop can proceed in steps of two). This results in better memory access and consequently faster execution.

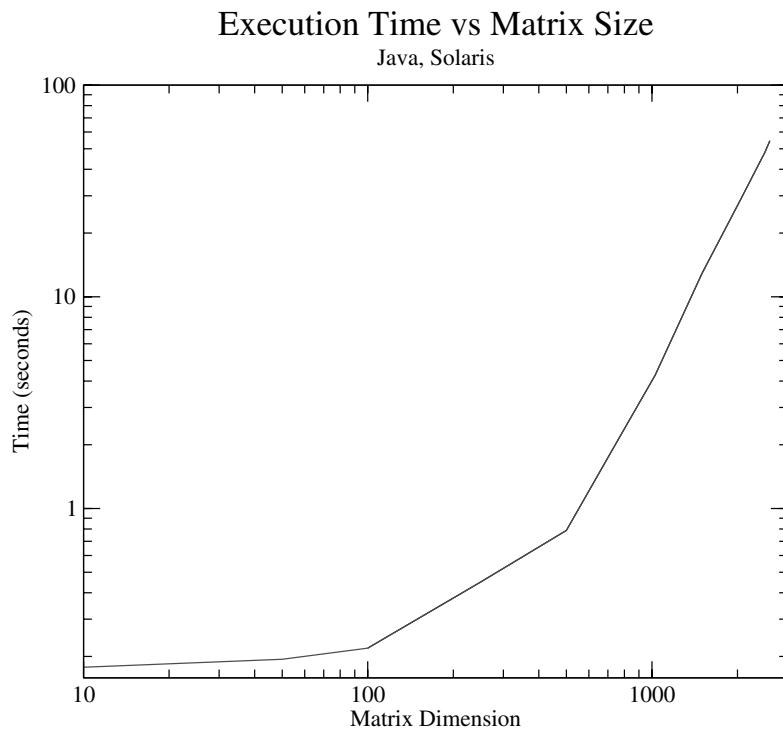
```

    for ( i=1; i < Ldim-1; i++ ) {
        ham[i][i] = i ;
        coef[i] = 0. ;
        diag[i] = ham [i][i];
    }
    coef[1] = 1. ;
    err = 1. ;
    iter = 0 ;
    while (iter < 15 && err > 1.e-6) {
        iter = iter + 1;
        // compute current energy & normalize
        ener = 0. ;
        ovlp1 = 0. ;
        ovlp2 = 0. ;
        for ( i= 1; i <= Ldim-2; i = i + 2 ) {
            ovlp1 = ovlp1 + coef[i]*coef[i] ;
            ovlp2 = ovlp2 + coef[i+1]*coef[i+1] ;
            t1 = 0. ;
            t2 = 0. ;
            for ( j=1; j <= Ldim-1; j++ ) {
                t1 = t1 + coef[j]*ham[j][i];
                t2 = t2 + coef[j]*ham[j][i+1];
            }
            sigma[i] = t1;
            sigma[i + 1] = t2;
            ener = ener + coef[i]*t1 + coef[i+1]*t2 ;
        }
        ovlp = ovlp1 + ovlp2 ;
        ener = ener/ovlp;
        fact = 1./Math.sqrt(ovlp);
        coef[1] = fact*coef[1];
        err = 0. ;
        // Update & error norm
        for ( i = 2; i <= Ldim-1; i++ ) {
            t = fact*coef[i];
            u = fact*sigma[i]-ener*t;
            step = u/(ener-diag[i]) ;
            coef[i] = t + step ;
            err = err + step*step ;
        }
        err = Math.sqrt(err) ;
        System.out.println
            ("iter, ener, err "+iter+", " + ener + ", " + err);
    }
    time = (System.currentTimeMillis() - time)/1000;
    System.out.println("time = " + time + "s"); // Elapsed time
}
}

```

- Look at `Tune4.java` and note where the nested `for` loop over `i` and `j` now takes step of  $\Delta i = 2$  rather than the unit steps in `Tune.java`. If things work as expected, the better memory access of `Tune4.java` should cut the runtime nearly in half. Compile and execute `Tune4.java`. Record your answer in your table.

- In order to cut the number of calls to the 2D array in half, we employed a technique known as *loop unrolling* in which we explicitly wrote out some of the lines of code which, otherwise, would be executed implicitly as the `for` loop went through all the values for its counters. This is not as clear a piece of code as before, but, evidently, it permits the compiler to produce a faster executable. To check that `Tune` and `Tune4` actually do the same thing, assume `ldim = 4` and run through one iteration of `Tune4.java` *by hand*. Hand in your manual trial.



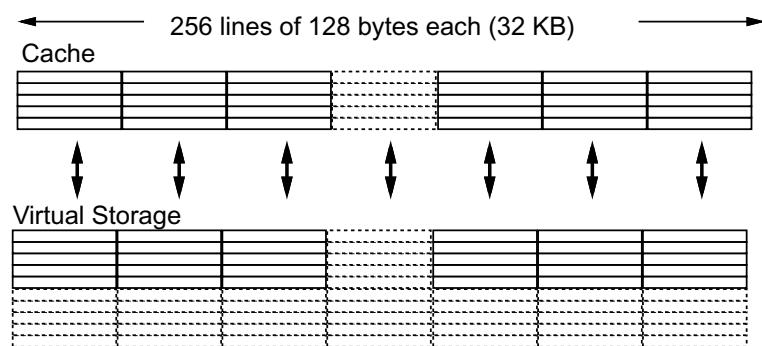
**Fig. 14.2** Running time versus matrix size for eigenvalue search using `Tune.java`. Note how much faster is Fortran (Fig. 14.1), and that for large sizes, the Java execution time varies as the third power of the matrix size. The extra power of size (compared to Fortran) arises from the time spent on reading the matrix elements into and out of memory, something that the Fortran program has been optimized to do rather well.

## 14.2

### Programming for Data Cache (Method)

Data caches are small, very fast memory used as temporary storage between the ultrafast CPU registers and the fast main memory. They have grown in importance as high-performance computers have become more prevalent. On systems that use a data cache, this may well be the single most important programming consideration; continually referencing data that are not in the cache (*cache misses*) may lead to an order-of-magnitude increase in CPU time.

As indicated in Figs. 13.2 and 14.3, the data cache holds a copy of some of the data in memory. The basics are the same for all caches but the sizes are manufacturer dependent. When the CPU tries to address a memory location, the *cache manager* checks to see if the data are in cache. If they are not, the manager reads the data from memory into cache and then the CPU deals with the data directly in cache. The cache manager's view of RAM is shown in Fig. 14.3.



**Fig. 14.3** The cache manager's view of RAM. Each 128-byte cache line is read into one of four lines in cache.

When considering how some matrix operation uses memory, it is important to consider the *stride* of that operation, that is, the number of array elements that get stepped through as an operation repeats. For instance, summing the diagonal elements of a matrix to form the trace

$$\text{Tr}A = \sum_{i=1}^N a(i,i) \quad (14.6)$$

involves a large stride because the diagonal elements are stored far apart for large  $N$ . However, the sum

$$c(i) = x(i) + x(i+1) \quad (14.7)$$

has stride 1 because adjacent elements of  $x$  are involved. The basic rule in programming for cache is

- Keep the stride low, preferably at 1, which in practice means.
- Vary the leftmost index first on Fortran arrays.
- Vary the rightmost index first on Java and C arrays.

#### 14.2.1

##### **Exercise 1: Cache Misses**

We have said a number of times that your program will be slowed down if the data it needs is in virtual memory and not in RAM. Likewise, your program will also be slowed down if the data required by the CPU is not cache. For high-performance computing, you should write programs that keep as much of the data being processed as possible in cache. To do this you should recall that Fortran matrices are stored in successive memory locations with the row index varying most rapidly (column-major order), while Java and C matrices are stored in successive memory locations with the column index varying most rapidly (row-major order). While it is difficult to isolate the effects of cache from other elements of the computer's architecture, you should now estimate its importance by comparing the time it takes to step through matrix elements row by row, to the time it takes to step through matrix elements column by column.

By actually running on machines available to you, check that these two simple codes with the same number of arithmetic operations will take significantly different times to run because one of them must make large jumps through memory with the memory locations addressed not yet read into cache:

##### **Sequential Column and Row References**

```
for j = 1, 9999; {
    x(j) = m(1,j)                                // Sequential column reference
```

```
for j = 1, 9999; {
    x(j) = m(j,1)                                // Sequential row reference
```

#### 14.2.2

##### **Exercise 2: Cache Flow**

Test the importance of cache flow on your machine by comparing the time it takes to run these two simple programs. Run for increasing column size `idim` and compare the times for loop A versus those for loop B. A computer with very small caches may be most sensitive to stride.

**GOOD f90, BAD Java/C Program; Minimum, Maximum Stride**

```
Dimension Vec(idim, jdim) // Loop A
for j = 1, jdim; {
    for i = 1, idim; {
        Ans = Ans + Vec(i, j) * Vec(i, j) // Stride 1 fetch (f90)
    }
}
```

**BAD f90, GOOD Java/C Program; Maximum, Minimum Stride**

```
Dimension Vec(idim, jdim) // Loop B
for i = 1, idim; {
    for j = 1, jdim; {
        Ans = Ans + Vec(i, j) * Vec(i, j) // Stride jdim fetch (f90)
    }
}
```

Loop A steps through the matrix `Vec` in column order. Loop B steps in row order. By changing the size of the columns (the rightmost index for Fortran), we change the size of the step (*stride*) we take through memory in Fortran. Both loops take us through all elements of the matrix, but the stride is different. By increasing the stride in any language, we use fewer elements already present in cache, require additional swapping and loading of cache, and thereby slow down the whole process.

**14.2.3****Exercise 3: Large Matrix Multiplication**

As you increase the dimension of the arrays in your program, memory use increases geometrically, and at some point you should be concerned about efficient memory use. The penultimate example of memory usage is large matrix multiplication:

$$[C] = [A] \times [B] \quad (14.8)$$

This involves all the concerns with the different kinds of memory. The natural way to code (14.8) follows from the definition of matrix multiplication:

$$c_{ij} = \sum_{k=1}^N a_{ik} \times b_{kj} \quad (14.9)$$

The sum is over a row of *A* times a column of *B*.

Try out these two codes on your computer. In Fortran, the first code has *B* with stride 1, but *C* with stride *N*. This is cured in the second code by performing the initialization in another loop. In Java and C, the problems are

reversed. On one of our machines, we found a factor of 100 difference in CPU times even though the number of operations is the same!

#### BAD f90, GOOD Java/C Program; Maximum, Minimum Stride

```
for i = 1, N; {  
    for j = 1, N; {  
        c(i, j) = 0.  
        for k = 1, N; {  
            c(i, j) = c(i, j) + a(i, k) * b(k, j)  
        }  
    }  
}
```

#### GOOD f90, BAD Java/C Program; Minimum, Maximum Stride

```
for j = 1, N; {  
    for i = 1, N; {  
        c(i, j) = 0.0  
    }  
    for k = 1, N; {  
        for i = 1, N; {  
            c(i, j) = c(i, j) + a(i, k)*b(k, j)  
        }  
    }  
}
```

## 16

**Quantum Eigenvalues via ODE Matching**

*In this chapter we examine how a differential-equation solver may be combined with a search algorithm to solve the eigenvalue problem. This problem requires us to solve the bound-state eigenvalue problem for the 1D, time dependent Schrödinger equation. Even though this equation is an ODE, which we know how to solve quite well by now, the extra requirement that we need to solve for bound states makes this an eigenvalue problem. Specifically, the bound-state requirement imposes a boundary conditions on the form of the solution, which, in turn, means that a solution will exist only for certain energies, the eigenenergies or eigenvalues.*

*If this all sounds a bit much for you now, rest assured that you do not need to understand all the physics behind these statements. What we want, is for you to get experience with the technique of conducting a numerical search for the eigenvalue, in conjunction with solving an ODE numerically. This is how one solve the numerical, ODE eigenvalue problem. In Section 29.1.1, we discuss how to solve the equivalent, but more advanced, momentum-space eigenvalue problem as a matrix problem. In Chap. 25 we study the related problem of the motion of a quantum wave packet confined to a potential well. Further discussions of the numerical bound-state problem are found in [28, 29].*

**Problem:** We want to determine whether the rules of quantum mechanics are applicable inside of a nucleus. More specifically, you are told that nuclei contain neutrons and protons (“nucleons”) with mass  $mc^2 \simeq 940$  MeV, and that a nucleus has a size of about 2 fm.<sup>1</sup> Your explicit **problem** is to see if these experimental facts are compatible, first, with quantum mechanics and, second, with the observation that there is a typical spacing of several million electron volts (MeV) between the ground and excited states in nuclei.

<sup>1</sup> A fm, or fermi, equals  $10^{-13}$  cm =  $10^{-15}$  m, and  $\hbar c \simeq 197.32$  MeV fm.

### 16.1

#### Theory: The Quantum Eigenvalue Problem

Quantum mechanics describes phenomena that occur at atomic or subatomic scales (an elementary particle is subatomic). It is a statistical theory in which the probability that a particle is located at point  $x$  is  $\mathcal{P} = |\psi(x)|^2 dx$ , where  $\psi(x)$  is called the *wavefunction*. If a particle of energy  $E$  moving in one dimension experiences a potential  $V(x)$ , its wave function is determined by the ODE known as the time-independent Schrödinger equation.<sup>2</sup>

$$\frac{-\hbar^2}{2m} \frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x). \quad (16.1)$$

In addition, when our problem tells us that the particle is “bound,” we are being told that it is confined to some finite region of space. The mathematical expression of confinement is that the probability of finding the particle over all of space is 1:

$$\int_{-\infty}^{\infty} dx |\psi(x)|^2 = 1 \quad (16.2)$$

Yet the only way to have a  $\psi(x)$  with finite integral is to have it decay exponentially as  $x \rightarrow \pm\infty$ :

$$\psi(x) \rightarrow \begin{cases} e^{-x} & \text{for } x \rightarrow +\infty \\ e^{+x} & \text{for } x \rightarrow -\infty \end{cases} \quad (16.3)$$

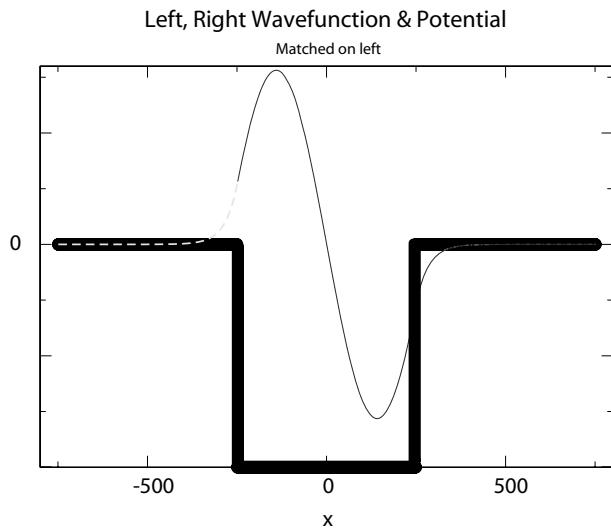
In summary, although it is straightforward to solve the ODE (16.1) with the techniques we have learned so far, we must also require that the solution  $\psi(x)$  simultaneously satisfies the boundary conditions (16.3). This extra condition turns the ODE problem into an *eigenvalue problem* that has solutions (*eigenvalues*) only for a certain values of the energy  $E$ . The ground state energy corresponds to the smallest (most negative) eigenvalue. The ground state wave function (eigenfunction), which we must determine in order to find its energy, must be nodeless and even (symmetric) about  $x = 0$ . The excited states have higher (less negative) energies, and wave functions that may be odd (antisymmetric).

##### 16.1.1

#### Model: Nucleon in a Box

The numerical methods we describe are capable of handling the most realistic potential shapes. Yet to make a connection with the standard textbook case,

<sup>2</sup> The time-dependent equation requires the solution of a partial differential equation, as discussed in Chap. 25.



**Fig. 16.1** Computed wave function and square well potential. The wave function computed by integration in from the right is matched to the one computed by integration in from the left (dashed) at a point near the left edge of the well. Note how the wave function decays rapidly outside of the well.

and to permit some analytic checking, we will use a simple model in which the potential  $V(x)$  in (16.1) is a finite square well (Fig. 16.1):

$$V(x) = \begin{cases} -V_0 = -83 \text{ MeV} & \text{for } |x| \leq a = 2 \text{ fm} \\ 0 & \text{for } |x| > a = 2 \text{ fm} \end{cases} \quad (16.4)$$

A depth of 83 MeV and radius of 2 fm are typical for nuclei, and therefore this problem will be solved with energies in millions of electron volts and lengths in fermis. With this potential the Schrödinger equation (16.1) becomes

$$\frac{d^2\psi(x)}{dx^2} + \frac{2m}{\hbar^2}(E + V_0)\psi(x) = 0 \quad \text{for } |x| \leq a \quad (16.5)$$

$$\frac{d^2\psi(x)}{dx^2} + \frac{2m}{\hbar^2}E\psi(x) = 0 \quad \text{for } |x| > a \quad (16.6)$$

To evaluate the ratio of constants here, we insert  $c^2$ , the speed-of-light squared, into the both numerator and the denominator [30, Appendix A.1]:

$$\frac{2m}{\hbar^2} = \frac{2mc^2}{(\hbar c)^2} \simeq \frac{2 \times 940 \text{ MeV}}{(197.32 \text{ MeV fm})^2} = 0.0483 \text{ MeV}^{-1}\text{fm}^{-2} \quad (16.7)$$

## 16.1.2

**Algorithm: Eigenvalues via ODE Solver + Search**

The solution of the eigenvalue problem combines the numerical solution of the ordinary differential equation (16.5) with a trial-and-error search for a wave function that satisfies the boundary conditions (16.3). This is done in several steps:

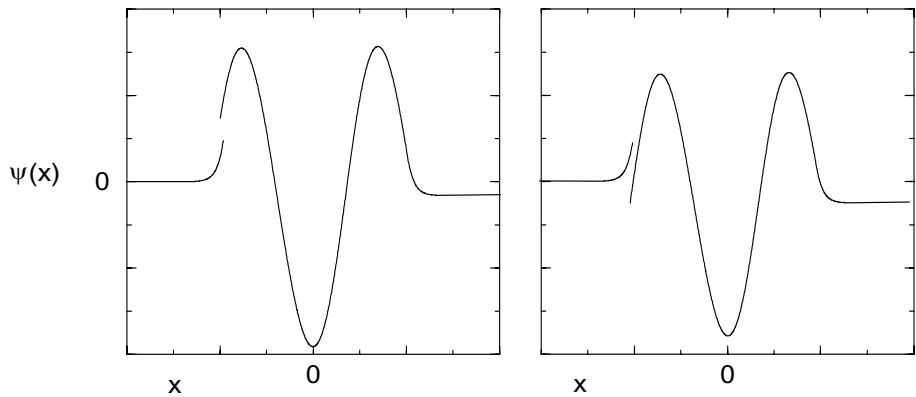
1. Start on the very far *left*, that is, at  $x = -X_{\max} \simeq -\infty$ , where  $X_{\max} \gg a$ , the width of the potential. We assume that the wave function there satisfies the LH boundary condition:

$$\psi_L(x = -X_{\max}) = e^{+x(-X_{\max})} = e^{-X_{\max}}$$

2. Use your favorite ODE solver to step  $\psi_L(x)$  in toward the origin (to the right) from  $x = -X_{\max}$ , until you reach the *matching radius*  $x_{\text{match}}$ . The exact value of this matching radius is not important, and our final solution should be independent of it. In Fig. 16.1, we show a sample solution with  $x_{\text{match}} = -a$ , that is, we match at the left edge of the potential well. In Fig. 16.2 we see some guesses that do not match.
3. Start on the very far *right*, that is, at  $x = +X_{\max} \simeq +\infty$ , with a wave function that satisfies the RH boundary condition:

$$\psi_R(x = +X_{\max}) = e^{-x(+X_{\max})} = e^{-X_{\max}}.$$

4. Use your favorite ODE solver to step  $\psi_R(x)$  in toward the origin (to the left) from  $x = +X_{\max}$ , until you reach the *matching radius*  $x_{\text{match}}$ . This means that we have stepped through the potential well (Fig. 16.1).
5. In order for probability and current to be continuous,  $\psi(x)$  and  $\psi'(x)$  must be continuous at  $x = x_{\text{match}}$ . Requiring the ratio  $\psi'(x)/\psi(x)$ , called the *logarithmic derivative*, to be continuous encapsulates both continuity conditions into a single condition, and is independent of  $\psi$ 's normalization.
6. Even though we do not know ahead of time which energies  $E$  are eigenvalues, we still need a value for the energy in order to use our ODE solver. Such being the case, we start off the solution with a guess for the energy. A good guess for the ground state energy would be a value somewhat up from the bottom of the well.
7. Because it is unlikely that any guess will be correct, the left- and right-wave functions will not quite match at  $x = x_{\text{match}}$  (Fig. 16.2). This is okay because we can use the amount of mismatch to improve the next



**Fig. 16.2** Left: A first guess at a wave function with an energy  $E$  that is 0.5% too low. We see that the left wave function does not vary rapidly enough to match the right one at  $x = 500$ . Right: A second guess at a wave function with an energy  $E$  that is 0.5% too high. We see that, now, the left wave function varies too rapidly.

guess. We measure how well the right and left wave functions match by calculating the difference

$$\Delta(E) = \left| \frac{\psi'_L(x)/\psi_L(x) - \psi'_R(x)/\psi_R(x)}{\psi'_L(x)/\psi_L(x) + \psi'_R(x)/\psi_R(x)} \right|_{x=x_{\text{match}}} \quad (16.8)$$

where the denominator is used to avoid overly large or small numbers. Then we try a different energy, note how much  $\Delta(E)$  has changed, and then make an intelligent guess at an improved energy. The search continues until the left- and right-wave logarithmic derivatives match within some level of tolerance.

**Listing 16.1:** QuantumEigen.java solves the 1D, time-independent Schrödinger equation for bound state energies using the rk4 algorithm.

```
// QuantumEigen.java: solves Schroed eq via rk4 + Bisection Algor

import java.io.*;

public class QuantumEigen {
    static double eps = 1E-6;                      // Class variables , precision
    static int n_steps = 501;                         // Number int steps

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {

        double E = -17., h = 0.04;      // Initial E in MeV, step size in fm
        double Emax, Emin, Diff;
        int count, count_max = 100;

        Emin = 1.1*E;      Emin = E/1.1;
```

```

    // Iteration loop
for ( count=0; count <= count_max; count++ ) {
    E = (Emax + Emin)/2.;                                     // Bisection
    Diff = diff(E, h);
    System.out.println("E = " + E + ", L-R Log deriv(E) = " +
        Diff);                                                 // Bisection algorithm
    if (diff(Emax, h)*Diff > 0) Emax = E;
    else Emin = E;
    if ( Math.abs(Diff) < eps ) break;
}
plot(E, h);
System.out.println("Final eigenvalue E = " + E);
System.out.println("iterations, max = " +count+", " + count_max);
System.out.println("WF in QuantumL/R.dat, V in QuantumV.dat");
}                                                               // End main

public static double diff(double E, double h)           // L-R log deriv
    throws IOException, FileNotFoundException {
    double left, right, x;
    int ix, nL, nR, i_match;
    double y[] = new double[2];
    i_match = n_steps/3;                                         // Matching radius
    nL = i_match + 1;
    y[0] = 1.E-15;                                              // Initial wf on left
    y[1] = y[0]*Math.sqrt(-E*0.4829);                         // Left wf
    for ( ix = 0; ix < nL + 1; ix++ ) {
        x = h * (ix -n_steps/2);
        rk4(x, y, h, 2, E);
    }
    left = y[1]/y[0];                                           // Log derivative
    y[0] = 1.E-15;                                              // - slope for even; reverse for odd
    y[1] = -y[0]*Math.sqrt(-E*0.4829);                        // Initialize R wf
    for ( ix = n_steps ; ix > nL + 1; ix-- ) {
        x = h * (ix + 1 -n_steps/2);
        rk4(x, y, -h, 2, E);
    }
    right = y[1]/y[0];                                         // Log derivative
    return( (left-right)/(left + right) );
}

// Repeat integrations for plot, can't integrate out decaying wf
public static void plot(double E, double h)
    throws IOException, FileNotFoundException {
    PrintWriter L =
        new PrintWriter(new FileOutputStream ("QuantumL.dat"), true);
    PrintWriter R =
        new PrintWriter(new FileOutputStream ("QuantumR.dat"), true);
    PrintWriter Vx =
        new PrintWriter(new FileOutputStream ("QuantumV.dat"), true);
    double left, right, normL, x = 0.;
    int ix, nL, nR, i_match;
    double y[] = new double[2];  double yL[][] = new double [2][505];
    int n_steps = 1501;          // Total no integration steps
    i_match = 500;              // Matching point
}

```

```

nL = i_match + 1;                                // Initial wf on the left
y[0] = 1.E-40;
y[1] = -Math.sqrt(-E*0.4829) *y[0];
for ( ix = 0; ix <= nL; ix++ ) {
    yL[0][ix] = y[0];
    yL[1][ix] = y[1];
    x = h * (ix -n_steps/2);
    rk4(x, y, h, 2, E);
}                                              // Integrate to the left
y[0] = -1.E-15;                                // - slope: even; reverse for odd
y[1] = -Math.sqrt(-E*0.4829)*y[0];
for ( ix = n_steps -1; ix >= nL + 1; ix-- ) { // Integrate in
    x = h * (ix + 1 -n_steps/2);
    R.println(x + " " + y[0] + " " + y[1]);      // File print
    Vx.println(x + " " + 1.E34*V(x));           // Scaled V
    rk4(x, y, -h, 2, E);
}
x = x - h;
R.println(x + " " + y[0] + " " + y[1]);          // File print
normL = y[0]/yL[0][nL];                          // Renormalize L wf & derivative
for ( ix = 0; ix <= nL; ix++ ) {
    x = h * (ix-n_steps/2 + 1);
    y[0] = yL[0][ix]*normL;
    y[1] = yL[1][ix]*normL;
    L.println(x + " " + y[0] + " " + y[1]);      // File print
    Vx.println(x + " " + 1.E34*V(x));           // Print V
}
return;
}

public static void f(double x, double y[], double F[], double E)
{ F[0] = y[1];      F[1] = -(0.4829)*(E-V(x))*y[0]; }

public static double V(double x)
{ if (Math.abs(x) < 10.) return ( -16.); else return (0.) ; }

// rk4 algorithm
public static void rk4(double t, double y[], double h,
                      int Neqs, double E)
{
    int i;
    double F[] = new double[Neqs];
    double ydumb[] = new double[Neqs];
    double k1[] = new double[Neqs]; double k2[] = new double[Neqs];
    double k3[] = new double[Neqs]; double k4[] = new double[Neqs];

    f(t, y, F,E);
    for (i=0; i<Neqs; i++)
    { k1[i] = h*F[i];
      ydumb[i] = y[i] + k1[i]/2; }

    f(t + h/2, ydumb, F,E);
    for (i=0; i<Neqs; i++)
    { k2[i] = h*F[i];
      ydumb[i] = y[i] + k2[i]/2; }

    f(t + h/2, ydumb, F,E);
    for (i=0; i<Neqs; i++)

```

```

{ k3[ i ]= h*F[ i ];
ydump[ i ] = y[ i ] + k3[ i ];}

f( t + h, ydump, F,E);
for ( i=0; i<Neqs; i++)
{ k4[ i ] = h*F[ i ];
y[ i ] = y[ i ] + (k1[ i ] + 2*(k2[ i ]+k3[ i ]) + k4[ i ]) / 6;
}
}

```

## 16.1.3

**Implementation: Eigenvalues via ODE Solver + Bisection**

1. Combine your bisection algorithm search program with your `rk4` ODE solver program. Start with a step size  $h = 0.04$ .
2. Write a subroutine which calculates the matching function  $\Delta(E)$  as a function of energy and matching radius. This subroutine will be called by the bisection algorithm program to search for the energy at which  $f(E, x = 2)$  vanishes.
3. As a first guess, take  $E \simeq 65$  MeV.
4. Search until  $\Delta(E)$  changes in the fourth decimal place, and modify the `f` function in `rk4` as appropriate for the Schrödinger equation (16.5). We do this in the code `QuantumEigen.java` shown in Listing 16.1.
5. Print out the value of the energy for each iteration. This will give you a feel as to how well the procedure converges, as well as a measure of the precision obtained. Try different values for the tolerance until you are confident that you are obtaining three good decimal places in the energy.
6. Build in a limit to the number of energy iterations you permit, and print out when the iteration scheme fails.
7. As we have done in Fig. 16.1, plot the wave function and potential on the same graph (you will have to scale the potential to get them both to fit).
8. Deduce, by counting the number of nodes in the wave function, whether the solution found is a ground state (no nodes) or an excited state (with nodes), and whether the solution is even or off (the ground state must be even).
9. Include in your version of Fig. 16.1, a horizontal line within the potential indicating the energy of the ground state relative to the potential's depth.

10. Increase the value of the initial energy guess and search for excited states. Make sure to examine the wave function for each state found to ensure that it is continuous, and to count the number of nodes. The number of nodes should increase as the levels get higher.
11. Add each new state found as another horizontal bar within the potential.
12. Verify that you have solved the **problem**, that is, that the spacing between levels is on the order of MeV for a nucleon bound in a several-fm well

#### 16.1.4

#### Explorations

1. Check to see how well your search procedure works by using arbitrary values for the starting energy. For example, because no bound-state energies can lie below the bottom of the well, try  $E = -V_0$  as well as some arbitrary fractions of  $V_0$ . In every case examine the resulting wave function and check that it is both symmetric and continuous.
2. Increase the depth of your potential progressively until you find more than one bound state. Look at the wave function in each case and correlate the number of nodes in the wave function and the position of the bound state in the well.
3. Explore how a bound-state energy changes as you change the depth  $V_0$  of the well. In particular, as you keep decreasing the depth, watch the eigenenergy move closer to  $E = 0$ , and see if you can find the potential depth at which the bound state has  $E \simeq 0$ .
4. For a fixed well depth  $V_0$ , explore how the energy of a bound state changes as the well radius  $a$  is varied.
5. Conduct some explorations in which you discover different values of  $(V_0, a)$  that give the same ground-state energies. The existence of several different combinations means that knowledge of a ground-state energy is not enough to determine a unique depth of the well.
6. Modify the procedures to solve for the eigenvalue and eigenfunction for odd wave functions.
7. Solve for the wave function of a linear potential:

$$V(x) = -V_0 \begin{cases} |x| & \text{for } |x| < a \\ 0 & \text{for } |x| > a \end{cases}$$

There is less potential here than for a square well, so you may expect lower binding energies and a less-confined wave function. (For this potential, there are no analytic results with which to compare.)

8. Compare the results obtained, and the time the computer took to get them, with the Numerov method and with `rk4`.

## 17

**Fourier Analysis of Linear and Nonlinear Signals**

*In this chapter we examine Fourier series and Fourier integrals (or transforms). This is the traditional tool for decomposing both periodic and nonperiodic motions, respectively, into an infinite number of harmonic functions. Because it represents a function as a series of sines and cosines of a time variable, a Fourier series always generates a periodic function. This clearly is desirable if the function we wish to approximate is periodic, in which case we construct the series to have the correct period. A nonperiodic function can frequently be well approximated by a Fourier series over some limited time, but the Fourier series will eventually show its periodicity for values outside this range. In this latter case of limited range, the Fourier integral is more appropriate.*

## 17.1

**Harmonics of Nonlinear Oscillations (Problem 1)**

Consider a particle oscillating in a nonharmonic potential. This could be the nonharmonic oscillator (15.4),

$$V(x) = \frac{1}{p}k|x|^p \quad (17.1)$$

for  $p \neq 2$ , the perturbed harmonic oscillator (15.2),

$$V(x) = \frac{1}{2}kx^2 \left(1 - \frac{2}{3}\alpha x\right) \quad (17.2)$$

or the realistic pendulum of Section 19.1. While free oscillations in these potentials are always periodic, they are not truly sinusoidal. Your **problem** is to take the solution of one of these nonlinear oscillators and relate it to the solution

$$x(t) = A_0 \sin(\omega t + \phi_0) \quad (17.3)$$

of the linear, harmonic oscillator. (In your future study of chaos, you will want to extend this analysis to the response of a damped, oscillating system driven by an external force.) If your oscillator is sufficiently nonlinear to behave like

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5

the sawtooth function (Fig. 17.1, left), then the Fourier spectrum you obtain should be similar to that shown on the right of Fig. 17.1.

In general, when we want to undertake such a spectral analysis, we want to analyze the steady-state behavior of a system. This means that the initial transient behavior has had a chance to die off. Just what is the initial transient is easy to identify for linear systems, but may be less so for nonlinear systems in which the “steady state” jumps among a number of configurations.

## 17.2

### Fourier Analysis (Math)

Nonlinear oscillations are interesting in part because they hardly ever are studied in traditional courses. This is true even though the linear term is just a first approximation to a naturally oscillating system. If the force on a particle is always toward its equilibrium position (a restoring force), then the resulting motion will be *periodic*, but not necessarily *harmonic*. A good example is the motion in a highly anharmonic well  $p \approx 10$ , which produces an  $x(t)$  looking like a series of pyramids; this is periodic but not harmonic.

On a computer, the distinction between a Fourier integral and a Fourier series is less clear because the integral is approximated as a finite series. We will illustrate both methods by analyzing anharmonic oscillations with the series and by analyzing the charge density of elementary systems with the integral.

In a sense, our approach is the inverse of the traditional one in which the *fundamental* oscillation is determined analytically, and the higher frequency *overtones* are determined in perturbation theory [31]. We start with the full (numerical) periodic solution and then decompose it into what may be called *harmonics*. When we speak of fundamentals, overtones, and harmonics, we speak of solutions to the *boundary-value problem*, for example, of waves on a plucked violin string. In this latter case, and when given the correct conditions (enough musical skill), it is possible to excite individual harmonics or sums of them within the series

$$y(t) = b_0 \sin \omega_0 t + b_1 \sin \frac{n\omega_0 t}{m} + \dots \quad (17.4)$$

The anharmonic oscillator vibrates with a single frequency (which may change with changing amplitude) but not a sinusoidal waveform. Expanding the anharmonic vibration as a Fourier series does not imply that the individual harmonics can be “played.”

You may recall from classical mechanics that the most general solution for some vibrating physical system can be expressed as the sum of the *normal modes* of that system. These expansions are possible because we have *linear*

operators and, subsequently, the *principle of superposition*: If  $x_1(t)$  and  $x_2(t)$  are solutions of some linear equation, then  $\alpha_1x_1(t) + \alpha_2x_2(t)$  is also a solution.

The principle of linear superposition does not hold when we solve nonlinear problems. Nevertheless, it is always possible to expand a *periodic* solution of a *nonlinear* problem in terms of trigonometric functions that have frequencies that are integer multiples of the true frequency of the nonlinear oscillator. This is a consequence of *Fourier's theorem* being applicable to any single-valued, periodic function with only a finite number of discontinuities. We assume we know the period  $T$ , that is, that

$$y(t+T) = y(t) \quad (17.5)$$

This tells us the “true” frequency  $\omega$ :<sup>1</sup>

$$\omega \equiv \omega_1 = \frac{2\pi}{T} \quad (17.6)$$

Any such periodic function can be expanded as a series of harmonic functions with frequencies that are multiples of the true frequency:

$$y(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos n\omega t + b_n \sin n\omega t) \quad (17.7)$$

This equation represents a signal as the simultaneous sum of pure tones of frequency  $n\omega$ . The coefficients  $a_n$  and  $b_n$  are a measure of the amount of  $\cos n\omega t$  and  $\sin n\omega t$  present in  $y(t)$ , specifically, the intensity or power at each frequency proportional to  $a_n^2 + b_n^2$ .

The Fourier series (17.7) is a “best fit” in the least-squares sense of Chap. 8 because it minimizes  $\sum_i [y(t_i) - y_i]^2$ . This means that the series converges to the average behavior of the function, but misses the function at discontinuities (at which points it converges to the mean) or at sharp corners (where it overshoots). A general function  $y(t)$  may contain an infinite number of Fourier components, although a good approximation is usually possible with a small number of harmonics.

The coefficients  $a_n$  and  $b_n$  are determined by the standard techniques for function expansion. To find them, you multiply both sides of (17.7) by  $\cos n\omega t$  or  $\sin n\omega t$ , integrate over one period, and project out a single  $a_n$  or  $b_n$ :

$$\begin{pmatrix} a_n \\ b_n \end{pmatrix} = \frac{2}{T} \int_0^T dt \begin{pmatrix} \cos n\omega t \\ \sin n\omega t \end{pmatrix} y(t) \quad \omega \stackrel{\text{def}}{=} \frac{2\pi}{T} \quad (17.8)$$

<sup>1</sup> We remind the reader that every periodic system by definition has a period  $T$  and consequently a “true” frequency  $\omega$ .

Nonetheless, this does not imply that the system behaves like  $\sin \omega t$ . Only harmonic oscillators do that.

As seen in the  $b_n$  coefficients (Fig. 17.1, right), these coefficients usually decrease in magnitude as the frequency increases, and can occur with negative sign, the negative sign indicating opposite phase.

Awareness of the *symmetry* of the function  $y(t)$  may eliminate the need to evaluate all the expansion coefficients. For example

- $a_0$  is twice the average value of  $y$ .

$$a_0 = 2 \langle y(t) \rangle \quad (17.9)$$

- For an *odd function*, that is, one for which  $y(-t) = -y(t)$ , all the coefficients  $a_n \equiv 0$  and only half the integration range is needed to determine  $b_n$ :

$$b_n = \frac{4}{T} \int_0^{T/2} dt y(t) \sin n\omega t \quad (17.10)$$

However, if there is no input signal for  $t < 0$ , we do not have a truly odd function, and so small values of  $a_n$  may occur.

- For an *even function*, that is, one for which  $y(-t) = y(t)$ , the coefficient  $b_n \equiv 0$  and only half the integration range is needed to determine  $a_n$ :

$$a_n = \frac{4}{T} \int_0^{T/2} dt y(t) \cos n\omega t \quad (17.11)$$

### 17.2.1

#### Example 1: Sawtooth Function

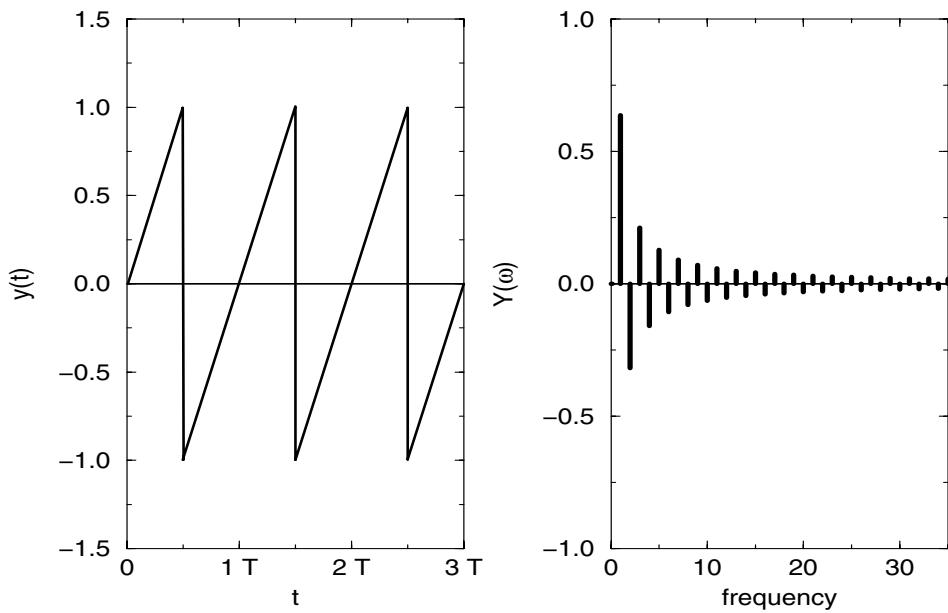
The sawtooth function (Fig. 17.1) is described mathematically as

$$y(t) = \begin{cases} \frac{t}{T/2} & \text{for } 0 \leq t \leq \frac{T}{2} \\ \frac{t-T}{T/2} & \text{for } \frac{T}{2} \leq t \leq T \end{cases} \quad (17.12)$$

It is clearly periodic, nonharmonic, and discontinuous. Yet it is also an odd function, and so can be represented more simply by shifting the period over to the left:

$$y(t) = \frac{t}{T/2} \quad -\frac{T}{2} \leq t \leq \frac{T}{2} \quad (17.13)$$

While it is relatively easy to reproduce the general shape of this function with only a few terms of the Fourier series, many components are needed to reproduce the discontinuities at the sharp corners. Because the function is odd, the



**Fig. 17.1** *Left:* A sawtooth function in time. *Right:* The Fourier spectrum of frequencies in natural units contained in this sawtooth function.

Fourier series is a sine series, and (17.8) determines the values

$$\begin{aligned} b_n &= \frac{2}{T} \int_{-T/2}^{+T/2} dt \sin n\omega t \frac{t}{(T/2)} = \frac{\omega}{\pi} \int_{-\pi/\omega}^{+\pi/\omega} dt \sin n\omega t \frac{\omega t}{\pi} \\ &= \frac{2}{n\pi} (-1)^{n+1} \end{aligned} \quad (17.14)$$

$$\Rightarrow y(t) = \frac{2}{\pi} [\sin \omega t - \frac{1}{2} \sin 2\omega t + \frac{1}{3} \sin 3\omega t - \dots] \quad (17.15)$$

Note that the terms in this series alternate in sign, which means that successive frequency components are out of phase with each other, yet that they all add together to give a straight line for  $-T/2 \leq t \leq T/2$ , with finite discontinuities as the interval repeats.

### 17.2.2

#### Example 2: Half-Wave Function

The half-wave function is

$$y(t) = \begin{cases} \sin \omega t & \text{for } 0 < t < T/2 \\ 0 & \text{for } T/2 < t < T \end{cases} \quad (17.16)$$

It is periodic, nonharmonic (the upper half of a sine wave), continuous, but with discontinuous derivatives. Because it lacks the sharp “corners” of the sawtooth, it is easier to reproduce with a finite Fourier series. Equations (17.8) determine

$$a_n = \begin{cases} \frac{-2}{\pi(n^2-1)}, & n \text{ even or } 0, \\ 0, & n \text{ odd,} \end{cases} \quad b_n = \begin{cases} \frac{1}{2}, & n = 1, \\ 0, & n \neq 1, \end{cases}$$

$$\Rightarrow y(t) = \frac{1}{2} \sin \omega t + \frac{1}{\pi} - \frac{2}{3\pi} \cos 2\omega t - \frac{2}{15\pi} \cos 4\omega t + \dots \quad (17.17)$$

### 17.3

#### Summation of Fourier Series(Exercise)

1. **Sawtooth function:** Sum the Fourier series for the *sawtooth function* up to order  $n = 2, 4, 10, 20$ , and plot the results over two periods.
  - (a) Check that in each case the series gives the mean value of the function *at* the points of discontinuity.
  - (b) Check that in each case the series *overshoots* by about 9% the value of the function on either side of the discontinuity (the *Gibbs phenomenon*).
2. **Half-wave function:** Sum the Fourier series for the *half-wave function* up to order  $n = 2, 4, 10$ , and plot the results over two periods. (The series converges quite well, doesn't it?)

### 17.4

#### Fourier Transforms (Theory)

Although a Fourier *series* is the right tool for approximating or analyzing periodic functions, the Fourier *transform* or *integral* is the right tool for nonperiodic functions. We transform the series formalism to the integral formalism by imagining a system described by a continuum of “fundamental” frequencies. We therefore deal with *wave packets* containing continuous rather than discrete frequencies.<sup>2</sup> While the difference between series and transform methods may appear clear mathematically, if we transform a function known only over a fi-

<sup>2</sup> We follow convention and consider time  $t$  as the function's variable and frequency  $\omega$  as the transform's variable. Nonetheless,

these can be reversed or other variables such as position  $x$  and wave vector  $k$  may also be used.

nite length of times, or approximate the Fourier integral as a finite sum, then the Fourier transform and the Fourier series become equivalent.

As an analogy to (17.7), we now imagine our function or signal  $y(t)$  expressed in terms of a continuous series of harmonics

$$y(t) = \int_{-\infty}^{+\infty} d\omega Y(\omega) \frac{e^{i\omega t}}{\sqrt{2\pi}} \quad (17.18)$$

where for compactness we use a complex exponential function.<sup>3</sup> Here the expansion amplitude  $Y(\omega)$  is analogous to  $(a_n, b_n)$  and is called the *Fourier transform* of  $y(t)$ .

A plot of the squared modulus  $|Y(\omega)|^2$  versus  $\omega$  is called the *power spectrum*. Actually, (17.18) is more properly called the *inverse transform* because it converts  $Y(\omega)$  to  $y(t)$ . The *Fourier transform* converts  $y(t)$  to  $Y(\omega)$ :

$$Y(\omega) = \int_{-\infty}^{+\infty} dt \frac{e^{-i\omega t}}{\sqrt{2\pi}} y(t) \quad (17.19)$$

You will note in (17.18) and (17.19) that only the relative sign in the exponential matters. In addition, you should note that we have chosen a symmetric  $1/\sqrt{2\pi}$  normalization factor, which is common in quantum mechanics [30] but differs from that used in engineering.

If  $y(t)$  represents the response of some system as a function of time,  $Y(\omega)$  is a *spectral function* that measures the amount of frequency  $\omega$  making up this response. While usually we think of measuring  $y(t)$  in the laboratory and numerically transforming it to obtain  $Y(\omega)$ , some experiments may well measure  $Y(\omega)$  directly [in which case a transform is needed to obtain  $y(t)$ ]. Clearly, the mathematics is symmetric even if the real world is not.

If the Fourier transform and its inverse are consistent with each other, we should be able to substitute (17.18) into (17.19) and obtain an identity:

$$\begin{aligned} Y(\omega) &= \int_{-\infty}^{+\infty} dt \frac{e^{-i\omega t}}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} d\omega' \frac{e^{i\omega' t}}{\sqrt{2\pi}} Y(\omega') \\ &= \int_{-\infty}^{+\infty} d\omega' \left\{ \int_{-\infty}^{+\infty} dt \frac{e^{i(\omega'-\omega)t}}{2\pi} \right\} Y(\omega') \end{aligned}$$

For this to be an identity the term in braces must be the *Dirac delta function*:

$$\int_{-\infty}^{+\infty} dt e^{i(\omega'-\omega)t} = 2\pi\delta(\omega' - \omega) \quad (17.20)$$

<sup>3</sup> Recall the principle of linear superposition and that  $\exp(i\omega t) = \cos \omega t + i \sin \omega t$ . This means that the real part of  $y$  gives the cosine series and the imaginary part the sine series.

While the delta function is one of the most common and useful functions in theoretical physics, it is not well behaved in a mathematical sense and is terribly misbehaved in a computational sense. While it is possible to create numerical approximations to  $\delta(\omega' - \omega)$ , they may well be borderline pathological. It is probably better for you to do the delta function part of an integration analytically and leave the nonsingular leftovers to the computer.

## 17.5

### Discrete Fourier Transform Algorithm (DFT)

If  $y(t)$  or  $Y(\omega)$  is known analytically, the integral (17.18) or (17.19) can be evaluated analytically or numerically using the integration techniques studied earlier (particularly Gaussian quadrature). Likewise, if a table of  $N$  values for  $y(t)$  is known, interpolations within the table can be used to evaluate the integral.

Here we will consider a technique for directly Fourier transforming functions that are known only for a finite number  $N$  of times  $t$  (for instance, as sampled in an experiment). This *discrete Fourier transform* (DFT) is an “approximate” procedure because the integrals are evaluated numerically.<sup>4</sup> By sampling a nonperiodic function at  $N$  times, we can determine  $N$  values of the Fourier transform of this function [ $N$  independent  $y(t)$  values can produce  $N$  independent  $Y(\omega)$  values]. We can then use those values of the transform to approximate the original function at any value of time. In this way the DFT can also be thought of as a technique for interpolating and extrapolating data.

Assume that the function  $y(t)$  we wish to transform is measured or sampled at a discrete number  $N + 1$  of times ( $N$  time intervals)

$$y_k \stackrel{\text{def}}{=} y(t_k) \quad k = 0, 1, 2, \dots, N \quad (17.21)$$

Assume that these times are evenly spaced with a time step  $h$ :

$$t_k = kh \quad h = \Delta t \quad (17.22)$$

In other words, we measure the signal  $y(t)$  once every  $h$  seconds during a total time interval of  $T$  and with a *sampling rate*  $s$ , with the two inversely related:

$$T \stackrel{\text{def}}{=} Nh \quad s = \frac{N}{T} = \frac{1}{h} \quad (17.23)$$

Regardless of the true periodicity of the function being sampled, when we choose a total time  $T$  over which to sample the function, the mathematics produces a  $y(t)$ , which is periodic with period  $T$ . To make this self-consistent and

<sup>4</sup> More discussion can be found in the book [32] devoted to just this topic.

ensure that there are only  $N$  independent function values used in the transform, we require the first and the last  $y$  values to be the same:

$$y(t+T) = y(t) \quad \Rightarrow \quad y_0 = y_N \quad (17.24)$$

If we are analyzing a truly periodic function, then the first  $N$  points should all be within one period to guarantee their independence. Unless we make further assumptions, these  $N$  independent input data  $y(t_k)$  can determine no more than  $N$  independent output Fourier components  $Y(\omega_k)$ .

The time interval  $T$  (which should be the period for periodic functions) is the largest time over which we consider variation of  $y(t)$ . Consequently, it determines the lowest frequency  $\omega_1 = 2\pi/T$  contained in our Fourier representation of  $y(t)$  (unless you want to be picky and argue that there may also be an  $\omega = 0$  or “DC” component).

While we will be able to compute  $N$  independent values  $Y(\omega_n)$  for  $n = 1, N$ , the values for the frequencies  $\omega_n$  are determined by the number of samples taken and the total sampling time  $T$ . To make the connection with Fourier series, we choose the frequencies as

$$\omega_n = n\omega_1 = n\frac{2\pi}{Nh} \quad n = 0, 1, \dots, N \quad (17.25)$$

Here the  $n = 0$  value corresponds to the zero frequency or DC component,  $\omega_0 = 0$ . We now see clearly that by limiting the time interval over which we sample the input function, we are making an approximation that limits the maximum frequency of the Fourier components we can compute.

Note that (17.25) indicates that the larger we make the time  $T = Nh$  over which we sample the function, the smaller will be the frequency steps. Accordingly, if you want a smooth frequency spectrum, you need small steps in frequency, and, such being the case, large  $T$ . While the best approach would be to measure the input signal for longer times, in practise a measured signal  $y(t)$  is often “padded” with zeros for  $t > T$  in order to produce a smoother spectrum. Effectively, this is building into the analysis the experimentalist’s belief that the signal does not repeat.

The discrete Fourier transform results from (1) evaluating the integral in (17.19), not from  $-\infty$  to  $+\infty$ , but rather from the times 0 to  $T$  over which the

signal is measured, and from (2) using the trapezoid rule for the integration<sup>5</sup>

$$Y(\omega_n) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} dt \frac{e^{-i\omega_n t}}{\sqrt{2\pi}} y(t) \simeq \int_0^T dt \frac{e^{-i\omega_n t}}{\sqrt{2\pi}} y(t) \quad (17.26)$$

$$\approx \sum_{k=1}^N h y(t_k) \frac{e^{-i\omega_n t_k}}{\sqrt{2\pi}} = h \sum_{k=1}^N y_k \frac{e^{-2\pi i k n / N}}{\sqrt{2\pi}} \quad (17.27)$$

To keep the final notation more symmetric, the step size  $h$  is factored from the transform  $Y$  and a discrete value  $Y_n$  is defined by

$$Y_n \stackrel{\text{def}}{=} \frac{1}{h} Y(\omega_n) = \sum_{k=1}^N y_k \frac{e^{-2\pi i k n / N}}{\sqrt{2\pi}} \quad (17.28)$$

With this same care in accounting, and  $d\omega \rightarrow 2\pi/Nh$ , we invert the  $Y_n$ 's:

$$y(t) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} d\omega \frac{e^{i\omega t}}{\sqrt{2\pi}} Y(\omega) \approx \sum_{n=1}^N \frac{2\pi}{Nh} \frac{e^{i\omega_n t}}{\sqrt{2\pi}} Y(\omega_n) \quad (17.29)$$

Once we know the  $N$  values of the transform, (17.29), which leaves off the DC component, is useful for evaluating  $y(t)$  for any value of the time  $t$ .

There is nothing illegal about evaluating the DFT expressions for  $Y_n$  and  $y_k$  for arbitrarily large values of  $n$  and  $k$ , yet there is also nothing to be gained. Because the trigonometric functions are periodic, we just get the old answer back:

$$y(t_{k+N}) = y(t_k) \quad Y(\omega_{n+N}) = Y(\omega_n) \quad (17.30)$$

Another way of stating this is to observe that none of the equations change if we replace  $\omega_n t$  by  $\omega_n t + 2\pi n$ . There are still just  $N$  independent output numbers for  $N$  independent inputs.

While periodicity is expected for Fourier *series*, it is somewhat surprising for Fourier *integrals*, which have been touted as the right tool for nonperiodic functions. The periodicity arises from approximating the integral as a sum over a finite number of periodic functions. Clearly, if we input values of the signal for longer lengths of time, then the inherent period becomes longer, and if the repeat period is very long, it may be of little consequence for times short compared to the period.

If  $y(t)$  is actually periodic with period  $Nh$ , then the integration formulas converge very rapidly and the DFT is an excellent way of obtaining Fourier

<sup>5</sup> The alert reader may be wondering what has happened to the  $h/2$  with which the trapezoid rule weights the initial and final

points. Actually, they are there, but because we have set  $y_0 \equiv y_N$ , two  $h/2$ 's have been added to produce one  $h$ .

series. If the input function is not periodic, then the DFT can be a bad approximation near the endpoints of the time interval (after which the function will repeat), or for the (low) frequencies corresponding to the entire function repeating.

The discrete Fourier transform and its inverse can be written in a concise and insightful way, and can be evaluated efficiently, by introducing a complex variable  $Z$  for the exponential, and then raising  $Z$  to various powers:

$$Y_n = \frac{1}{\sqrt{2\pi}} \sum_{k=1}^N Z^{nk} y_k \quad n = 0, 1, \dots, N \quad (17.31)$$

$$y_k = \frac{\sqrt{2\pi}}{N} \sum_{n=1}^N Z^{-nk} Y_n \quad Z = e^{-2\pi i/N} \quad (17.32)$$

where  $Z^{nk} \equiv [(Z)^n]^k$ . Now the computer needs to compute only powers of  $Z$ . We give our DFT code in Listing 17.1 (which uses approximately half of its length on plotting).

If your preference is to avoid complex numbers, we can rewrite (17.31) in terms of separate real and imaginary parts by applying Euler's theorem:

$$Z = e^{-i\theta} \quad \Rightarrow \quad Z^{\pm nk} = e^{\mp ink\theta} = \cos nk\theta \mp i \sin nk\theta \quad (17.33)$$

where  $\theta \stackrel{\text{def}}{=} 2\pi/N$ . In terms of the explicit real and imaginary parts:

$$\begin{aligned} Y_n &= \frac{1}{\sqrt{2\pi}} \sum_{k=1}^N [ (\cos(nk\theta) \operatorname{Re} y_k + \sin(nk\theta) \operatorname{Im} y_k \\ &\quad + i(\cos(nk\theta) \operatorname{Im} y_k - \sin(nk\theta) \operatorname{Re} y_k)], \end{aligned} \quad (17.34)$$

$$\begin{aligned} y_k &= \frac{\sqrt{2\pi}}{N} \sum_{n=1}^N [ (\cos(nk\theta) \operatorname{Re} Y_n - \sin(nk\theta) \operatorname{Im} Y_n \\ &\quad + i(\cos(nk\theta) \operatorname{Im} Y_n + \sin(nk\theta) \operatorname{Re} Y_n)]. \end{aligned} \quad (17.35)$$

Equation (17.34) is interesting in that it shows that a real function produces a real Fourier transform only if all the  $\sin nk\theta$  terms cancel out. This is to be expected if  $y(t)$  is an even function of  $t$  and if we perform an exact transform with integration from  $-\infty$  to  $+\infty$ . Yet because we have no signal for  $t < 0$ , and because we only approximate the transform integral, even real functions may end up with DFTs. However, the imaginary components should get smaller as we increase the number of integration (sampling) points.

**Listing 17.1:** DFT.java computes the discrete Fourier transform for the signal given in the method f(signal[]). You will have to add output and plotting to see the results. (The Instructor's version also does an inverse transform and plots the results with PtPlot.)

```
//DFT.java : Discrete Fourier Transform

import java.io.*;

public class DFT {
    static final int N = 1000, Np = N; // Global constants
    static double [] signal = new double[N + 1];
    static double twopi = 2.*Math.PI, sq2pi = 1./Math.sqrt(twopi);
    static double h = twopi/N;

    public static void main(String[] argv){

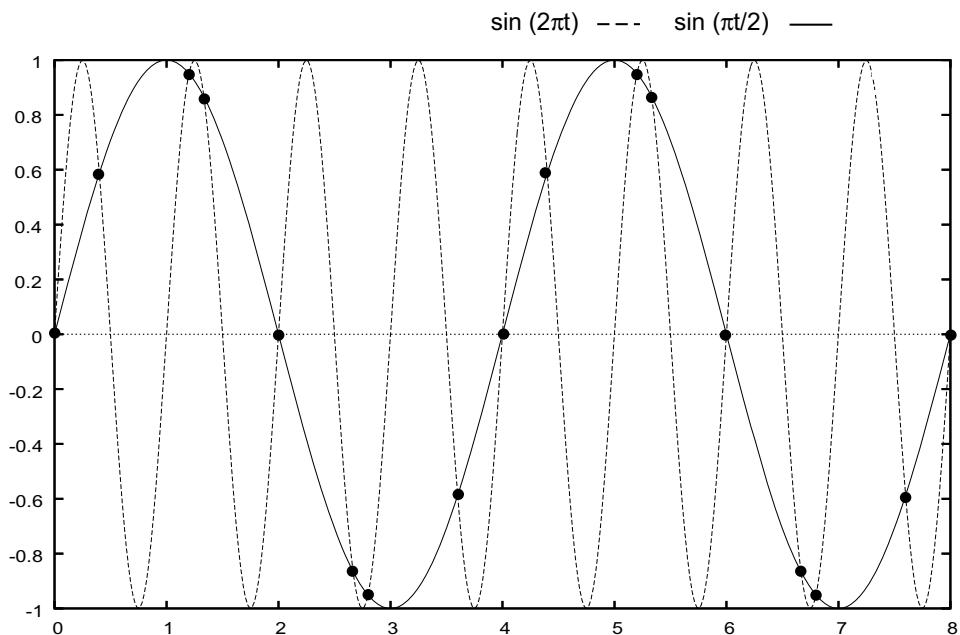
        double dftreal[] = new double[Np];
        double dftimag[] = new double[Np];

        f(signal);
        fourier(dftreal, dftimag);
    }

    public static void fourier(double dftreal[], double dftimag[]){
        double real, imag; // Calc & plot Y(w)
        int n, k;
        for ( n = 0; n < Np; n++ ) { // Loop on frequency
            real = imag = 0.; // Clear variables
            for ( k = 0; k < N; k++ ) { // Major loop
                real += signal[k] * Math.cos( twopi*k*n/N );
                imag += signal[k] * Math.sin( twopi*k*n/N );
            }
            dftreal[n] = real*sq2pi;
            dftimag[n] = -imag*sq2pi;
        }
    }

    public static void f(double [] signal) { // Signal function
        int i;
        double step = twopi/N, x = 0.;
        for ( i=0; i <= N; i++ ){
            signal[i] = 5. + 10*Math.sin(x+2.);
            x += step;
        }
    }
}
```

The actual computation time for a discrete Fourier transform can be reduced even further by use of the *fast Fourier transform (FFT)* algorithm. An examination of (17.31) shows that the DFT is evaluated as a matrix multiplication of a vector of length  $N$  of  $Z$  values times a vector of length  $N$  of  $y$  value. The time for this DFT scales like  $N^2$ . With the FFT algorithm, the time would scale like  $N \log_2 N$ . While this may not seem like much at first, for  $N = 10^{2-3}$ , the dif-



**Fig. 17.2** A plot of the functions  $\sin(\pi t/2)$  and  $\sin(2\pi t)$ . If the sampling rate is not high enough, these signals would appear indistinguishable. If both were present in a signal that was not sampled at a high enough rate, the deduced low-frequency component would be contaminated by the higher frequency component.

ference of a factor  $10^{3-5}$  is the difference between a minute and a week. This is the reason FFT is often used for the online analysis of data.

## 17.6 Aliasing and Antialiasing (Assessment)

A consequence of sampling a signal for only a finite number of times, as done with a DFT, is that this may lead to a poor deduction of the high-frequency components present in the signal. Clearly, obtaining good information about very high frequencies requires sampling the signal with very small time steps. While a poor deduction of the high-frequency components may be tolerable if all we care about are the low-frequency ones, those high-frequency components remain present in the signal and may contaminate the low-frequency components that we deduce. This effect is called *aliasing*, and is the cause of the Moire pattern distortion in digital images.

As an example, consider Fig. 17.2 showing the two functions  $\sin(\frac{\pi}{2}t)$  and  $\sin(2\pi t)$  for  $0 \leq t \leq 8$ , with their points of overlap emphasized. If we were unfortunate enough to sample a signal containing these functions at the times

$t = 0, 2, 4, 6, 8$ , then we would measure 0 and assume that there was no signal at all. However, if we were unfortunate enough to measure the signal at the filled dots in Fig. 17.2 where  $\sin(\frac{\pi}{2}t) = \sin(2\pi t)$ , specifically,  $t = 0, \frac{12}{10}, \frac{4}{3}, \dots$ , then our Fourier analysis would completely miss the high-frequency component. In DFT jargon, we would say that the high frequency has been *aliased* by the low-frequency component. In other cases, some high-frequency values may be included in our sampling of the signal, but our sampling rate may not be high enough to include enough of them to resolve the high-frequency component properly. In this case, some high-frequency signals get included spuriously as part of the low-frequency spectrum, and this leads to spurious low-frequency oscillations when the signal is synthesized from its Fourier components.

More precisely, aliasing occurs when a signal containing frequency  $f$  is sampled at a rate of  $s = N/T$  measurements per unit time, with  $s \leq f/2$ . In this case, the frequencies  $f$  and  $f - 2s$  yield the same DFT, and we would not be able to determine that there are two frequencies present. That being the case, to avoid aliasing we want no frequencies  $f > s/2$  to be present in our input signal. This is known as the *Nyquist criterion*.

In practice we could avoid the effects of aliasing by filtering out the high frequencies from our signal and then analyzing the remaining low-frequency part. (The low-frequency *sinc filter* is often used for this.) Even though this approach eliminates some information about the high frequencies, it avoids the contamination, and consequent distortion, of the low-frequency components. This often leads to an improved reproduction of the signal.

If accurate values for the high frequencies are required, then we need to increase the sampling rate  $s$  by increasing the number  $N$  of samples taken within our fixed sampling time  $T = Nh$ . By keeping the sampling time constant and increasing the number of samples taken, we make the time step  $h$  smaller, and this picks up the higher frequencies. By increasing the number  $N$  of frequencies that you compute, you move the higher frequency components you are interested in closer to the middle of the spectrum and thus away from the error-prone ends.

While we are talking about these kind of practical matters, it is worthwhile to point out again the effect of varying the total time  $T = Nh$  over which the signal is sampled, but not the sampling rate  $s = N/T = 1/h$ . Because the discrete frequencies of the DFT,

$$\omega_n = n\omega_1 = n\frac{2\pi}{T} \quad (17.36)$$

are measured in steps of  $\omega_1$ , if we increase the total time  $T$  over which the signal is sampled,  $\omega_1$  gets smaller, and the corresponding frequency spectrum looks smoother. However, to keep the time step the same, we would also need

to increase the number of samples,  $N$ . And as we said, this is often done, after the fact, by “padding” the end of the data set with zeros.

## 17.7

### DFT for Fourier Series (Algorithm)

For simplicity let us consider the Fourier cosine series:

$$y(t) = \sum_{n=0}^{\infty} a_n \cos(n\omega t) \quad a_k = \frac{2}{T} \int_0^T dt \cos(k\omega t) y(t) \quad (17.37)$$

Here  $T \stackrel{\text{def}}{=} 2\pi/\omega$  is the actual period of the system (not necessarily the period of the simple harmonic motion occurring for small amplitude). We assume that the function  $y(t)$  is sampled for a discrete set of times

$$y(t = t_k) \equiv y_k \quad k = 0, 1, \dots, N \quad (17.38)$$

Because we are analyzing a periodic function, we will retain the conventions used in the DFT and require the function to repeat itself with period  $T = Nh$ , that is, we assume that the amplitude is the same at the first and last points:

$$y_0 = y_N \quad (17.39)$$

This means that there are only  $N$  independent values of  $y$  being used as input. For these  $N$  independent  $y_k$  values, we can determine uniquely only  $N$  expansion coefficients  $a_k$ . If we use the trapezoid rule to approximate the integration in (17.37), we determine the  $N$  independent Fourier components as

$$a_n \simeq \frac{2h}{T} \sum_{k=1}^N \cos(n\omega t_k) y(t_k) = \frac{2}{N} \sum_{k=1}^N \cos\left(\frac{2\pi nk}{N}\right) y_k, \quad n = 0, \dots, N \quad (17.40)$$

Because for  $N$  independent  $y(t)$  values we can determine only  $N$  Fourier components, our Fourier series for the function  $y(t)$  must be in terms of only these components:

$$y(t) \simeq \sum_{n=0}^N a_n \cos(n\omega t) = \sum_{n=0}^N a_n \cos\left(\frac{2\pi nt}{Nh}\right) \quad (17.41)$$

In summary, we sample the function  $y(t)$  at  $N$  times,  $t_1, \dots, t_N$ . Because  $y(t)$  is periodic, if we sample within one period, we are ensured of independent input data. You see that all  $N$  values of  $y$  sampled contribute to each  $a_k$ . Consequently, if we increase  $N$  in order to determine more coefficients, we must recompute all the  $a_n$  values. In the model-independent approach

discussed in Section 17.10, the theory is reformulated so that additional samplings determine higher Fourier components without affecting lower ones.

## 17.8 Assessments

- **Simple Analytic Input**

The simple checks here are generally good to do before examining more complex problems. If your system has some Fourier analysis packages (such as the graphing package *Ace/gr*), you may want to compare your results with those from the packages. Once you understand how the packages work, it makes sense to use them.

1. Sample the even signal,

$$y(t) = 3 \cos(\omega t) + 2 \cos(3\omega t) + \cos(5\omega t)$$

Decompose this into its components and check that they are essentially real and in the ratio 3:2:1 (or 9:4:1 for the power spectrum).

2. Experiment on the effect of picking different values of the step size  $h$  and, independently, of extending the measurement period  $T = Nh$  to larger values.
3. Sample the odd signal,

$$y(t) = \sin(\omega t) + 2 \sin(3\omega t) + 3 \sin(5\omega t)$$

Decompose this into its components and check that they are imaginary and in the ratio 1:2:3 (or 1:4:9 if a power spectrum is plotted).

4. Sample the mixed-symmetry signal

$$y(t) = 5 \sin(\omega t) + 2 \cos(3\omega t) + \sin(5\omega t)$$

Decompose this into its components and see if there are three of them in the ratio 5:2:1 (or 25:4:1 if a power spectrum is plotted). Then check that your  $Y_n$  values can be resummed to reproduce this input.

5. In our discussion of aliasing, we examined Fig. 17.2, which shows the functions  $\sin(\frac{\pi}{2}x)$  and  $\sin(2\pi x)$ . Sample the function

$$y(t) = \sin(\frac{\pi}{2}x) + \sin(2\pi x)$$

with a sampling rate that leads to aliasing, as well as a higher sampling rate at which there is no aliasing. Compare the resulting DFTs in each case, and check if your simulations agree with the Nyquist criterion.

- **Highly Nonlinear Oscillator**

Recall the numerical solution for oscillations of the spring with power  $p = 11$ , (Eq. (17.1)). Decompose the solution into a Fourier series and determine the number of higher harmonics that contribute at least 10%; for example, determine the  $n$  for which  $|b_n/b_1| < 0.1$ . Specifically, check that summing your series reproduces your original solution. (Warning: The  $\omega$  you use in your series must correspond to the actual frequency of the system, not just that in the small oscillation limit.)

- **Nonlinearly Perturbed Oscillator**

Recall the harmonic oscillator with a nonlinear perturbation (15.2):

$$V(x) = \frac{1}{2}kx^2 \left(1 - \frac{2}{3}\alpha x\right) \quad F(x) = -kx(1 - \alpha x) \quad (17.42)$$

For very small amplitudes of oscillation ( $x \ll 1/\alpha$ ), the solution  $x(t)$  will essentially be only the first term of a Fourier series.

1. We want to say that “we have approximately a 10% nonlinearity.” Such being the case, fix your value of  $\alpha$  so that  $\alpha x_{\max} \simeq 10\%$ , where  $x_{\max}$  is the maximum amplitude of oscillation. For the rest of the problem, keep the value of  $\alpha$  fixed.
2. Decompose your numerical solution into a discrete Fourier spectrum.
3. Plot a graph of the percentage importance of the first *two*, non-DC Fourier components as a function of the initial displacement for  $0 < x_0 < 1/2\alpha$ . You should find that higher harmonics are more important as the amplitude increases. Because there may be both even and odd components present, there may be both real and imaginary parts to  $Y_n$ . Consequently, one way to answer this question is to calculate  $|Y_{n+1}|/|Y_n|$ , where  $|Y_n| = \sqrt{(\text{Re}Y_n)^2 + (\text{Im}Y_n)^2}$ . Alternatively, you can also look at successive terms in the power spectrum, although then the 10% effect in amplitude becomes a 1% effect in power.
4. As always, make sure to check by resuming the series for  $y(t)$  and seeing if the input is reproduced.

(Warning: The  $\omega$  you use in your series must correspond to the *true* frequency of the system, not just  $\omega$  in the small oscillation limit.)

## 17.9

### DFT of Nonperiodic Functions (Exploration)

Consider a simple model of a “localized” electron that moves through space and time. We assume that the electron is described by a wave packet  $\psi(x)$  that

is a function of the spatial coordinate  $x$ . A good model for an electron initially localized around  $x = 5$  is a Gaussian multiplying a plane wave:

$$\psi(x, t = 0) = \exp \left[ -\frac{1}{2} \left( \frac{x - 5.0}{\sigma_0} \right)^2 \right] e^{ik_0 x} \quad (17.43)$$

This wave packet is not an eigenstate of the momentum operator<sup>6</sup>  $p = id/dx$ , and in fact contains a spread of momenta. The **problem** is to evaluate the Fourier transform,

$$\psi(p) = \int_{-\infty}^{+\infty} dx \frac{e^{ipx}}{\sqrt{2\pi}} \psi(x) \quad (17.44)$$

as a way of determining the momenta spectrum in (17.43).

### 17.10

#### Model Independent Data Analysis (Exploration) ⊖

The scattering of electrons and x-rays from solids, atoms, molecules, and nuclei provides a means of determining the charge density  $\rho(r)$  of the target. The experiments actually measure the *form factor* or *structure function* of the target,<sup>7</sup> that is, the Fourier transform of the charge density:

$$F(q) = \int d^3r e^{i\mathbf{q} \cdot \mathbf{r}} \rho(r) = 4\pi \int_0^{\infty} r^2 dr \frac{\sin qr}{qr} \rho(r) \quad (17.45)$$

Here  $\mathbf{q}$  is the momentum transferred during scattering and the 1D integral obtains for spherically symmetric  $\rho(r)$ . The **problem** is to determine  $\rho(r)$  from experimental measurements of  $F(q)$ , that is, to *invert* the transform in (17.45) to obtain  $\rho$  as a function of  $r$ . The real problem is that a laboratory beam of particles has a finite momentum, which means that there is a finite limit to the largest  $q$  value  $q_{\max}$  at which  $F(q)$  can be measured, and that experiments must be finished in a finite period of time, which means that only a finite number of  $q$  values can be measured. The  $q < q_{\max}$  limitation leads to uncertainties in those parts of  $\rho(r)$  that oscillate with high frequencies. The discrete number of measurements means that there are uncertainties in lower frequency components as well.

<sup>6</sup> We use natural units in which  $\hbar = 1$ .

<sup>7</sup> While the form factor can be deduced directly from experiment only in first Born approximation, the method described here is more general.

The traditional solution to this problem has been to *assume* that some specific functional form for the density  $\rho(r)$  containing a number of adjustable parameters, and then find a best fit to the data by varying the parameters. A shortcoming with this approach is that the values deduced for the density depend somewhat on the functional form assumed for  $\rho(r)$ .

A more general approach is called *model-independent analysis*. In it,  $\rho(r)$  is expanded in a complete set of functions

$$\rho(r) = \sum_{n=1}^{\infty} \rho_n(r) \quad (17.46)$$

with, typically, one parameter in each  $\rho_n(r)$ . If the  $\rho_n$  values form a complete set and if the sum actually goes out to  $n = \infty$ , then this is an exact representation of any  $\rho(r)$ . In practice, the discrete and finite nature of the measurements limit the number of  $\rho_n$  values that can be determined to a maximum number  $N$ , and so we have an approximate representation:

$$\rho(r) \approx \sum_{n=1}^N \rho_n(r) \quad (17.47)$$

To make a clear separation of how the deduced  $\rho_n(r)$  values are affected by the measurements of  $F(q)$  for  $q > q_{\max}$ , we impose the constraint that the  $\rho_n$  values determined for measurement with  $q < q_{\max}$  be orthogonal to those determined with  $q > q_{\max}$ . While the equations will appear similar to those in DFT, in DFT all  $N$  values of  $y_k$  are used to determine each  $Y_n$ , while here we require each  $\rho_n$  to be determined by only *one* measured  $F(q)$ . It is then manifest that each new experimental measurement determines one additional expansion coefficient.

Now for a specific example. The density  $\rho(r)$  within a specific atomic nucleus is known to vanish rapidly beyond some radius  $r \simeq R$  and to be approximately constant for  $r \simeq 0$ . We therefore model  $\rho(r)$  as the sine series:

$$r\rho(r) = \begin{cases} \sum_{n=1}^N b_n \sin(q_n r) & \text{for } r \leq R \\ 0 & \text{for } r > R \end{cases} \quad q_n = \frac{n\pi}{R} \quad (17.48)$$

where we are a simple formula in place of measured momentum transfer. This makes it clear that each additional measurement is made at a higher  $q$  value. The coefficients  $b_n$  are determined from the inversion formula (17.8):

$$b_n = \frac{2}{R} \int_0^R \sin(q_n r) r\rho(r) dr = \frac{q_n F(q_n)}{2\pi R} \quad (17.49)$$

The corresponding expression for the form factor as determined by the measured values  $F(q_n)$  is

$$F(q) = \frac{2\pi R}{q} \sum_{n=1}^N b_n \left\{ \frac{\sin[(q - q_n)R]}{(q - q_n)R} - \frac{\sin[(q + q_n)R]}{(q + q_n)R} \right\} \quad (17.50)$$

These relations show exactly how measurements at larger and larger momentum transfers,  $q_n$  values, determine the higher and higher Fourier components  $b_n$  values, which, in turn, are related to higher and higher frequency ripples in the charge density. (This is sometimes stated somewhat loosely as “large  $q$  are needed to measure small  $r$ .”) Regardless of the model assumed for the density  $\rho$ , any experiment with a definite  $q_{\max}$  has a limit on the components of the density it can deduce. As higher and higher  $q$  measurements are made, more components of  $\rho$  are determined.

### 17.11 Assessment

1. As a simple exercise, verify the sensitive relation between each measurement at  $q_n$  and the Fourier component  $b_n$  by evaluating the term in braces in (17.50) for  $n = 10$  and plotting it as a function of  $qR$ . You should find a peaking that indicates the region of sensitivity, that is, the region to which a single experimental measurement is most sensitive.
2. Now try a computer experiment that simulates a model-independent data analysis. Assume at first that you are mother nature and so you know that the actual charge distribution  $\rho(r)$  and form factor for some nucleus are

$$\rho(r) = \rho(0) \left[ 1 + \alpha(r/a)^2 \right] e^{-(r/a)^2} \quad (17.51)$$

$$F(q) = \left[ 1 - \frac{\alpha(qa)^2}{2(2+3\alpha)} \right] e^{-(qa)^2/4} \quad (17.52)$$

Use this analytic expression for  $F(q)$  to determine the values of the experimental  $F(q_m)$  as needed. In a real-world situation you would measure data and then fit them to determine the Fourier coefficients  $b_m$ .

- (a) Consider the nucleus  $^{16}\text{O}$  which has  $\alpha = \frac{4}{3}$ . We will measure distances in fermis,  $\text{fm} = 10^{-13} \text{ cm}$  and momentum transfers  $q$  in inverse fermis. In these units, mother nature knows that  $a \simeq 1.66$ , and you as an outsider would have to try values of  $R \simeq 5$  and  $R \simeq 6$  as the radius beyond which the nuclear density vanishes.
- (b) Generate from (17.52) a table of  $F(q_n)$  values for both values of  $R$  and for  $q$  values starting at zero and increasing to the point where  $F(q) \approx 10^{-11}$ . Plot up both (they should fall on the same curve).

- (c) Use (17.48) to calculate and plot the contribution to  $\rho(r)$  coming from progressively larger and larger values of  $q_n$ .
- (d) For 10 terms, examine how the sum for  $\rho(r)$  differs for the two  $R$  values used. This is a good measure of the *model dependence* in this “model independent” analysis.
- (e) Again, for the sum of 10 terms, examine how the Fourier series for  $\rho(r)$  differs from the actual functional form (17.51).
- (f) Examine the series expansion for  $F(q)$  and  $\rho(r)$  and note any unphysical oscillations.
- (g) Do a computer experiment in which you assume that a different form for the large  $q$  behavior of  $F(q)$  [e.g.,  $1/q^4$ ,  $\exp(-bq)$ ], and then see how this affects the deduced  $\rho(r)$ . You should find that the small  $r$  ripples are most sensitive to the assumed large  $q$  behavior.

## 18

**Unusual Dynamics of Nonlinear Systems**

*Nonlinear dynamics is one of the glorious successes of computational science. It has been explored by mathematicians, scientists and engineers, and usually with computers as an essential tool. (Even theologists have found motivation from the mathematical fact that simple systems can have very complicated behaviors.) The computed solutions have led to the discovery of new phenomena such as solitons, chaos, and fractals, and even our brief treatment of the subject will permit you to uncover unusual properties on your own. In addition, because biological systems often have complex interactions, and may not be in thermodynamic equilibrium states, models of them are often nonlinear, with properties similar to other complex systems.*

*In this chapter, we develop the logistics map as a model for how bug populations achieve dynamic equilibrium. It is an example of a very simple, but nonlinear, equation producing surprising complex behavior. In Chap. 19 we explore chaos for two continuous systems, the driven realistic pendulum [33, 34] and coupled predator-prey populations.*

**Problem:** The population of insects and the patterns of weather do not appear to follow any simple laws.<sup>1</sup> At times they appear stable, at other times they vary periodically, and at other times they appear chaotic, only to settle down to something simple again. Your **problem** is to deduce if a simple and discrete law can produce such complicated behavior.

## 18.1

**The Logistic Map (Model)**

Imagine a bunch of insects reproducing, generation after generation. We start with  $N_0$  bugs, then in the next generation we have to live with  $N_1$  of them, and after  $i$  generations there are  $N_i$  bugs to bug us. We want to define a model for how  $N_n$  varies with the discrete generation number  $n$ . For guidance, we look to the radioactive decay simulation in Chap. 11 where the discrete decay law,  $\Delta N / \Delta t = -\lambda N$ , led to exponential-like decay. Likewise, if we reverse

<sup>1</sup> Excepting Oregon, where storms spend their weekends.

of sign of  $\lambda$  we should get exponential-like growth, which is a good place to start our modeling. We assume that the bug breeding rate is proportional to the number of bugs:

$$\frac{\Delta N_i}{\Delta t} = \lambda N_i \quad (18.1)$$

We improve the model by incorporating the fact that bugs do live on love alone, they must also eat. But bugs, being not farmers, must compete for the available food supply, and this might limit their number to a maximum  $N_*$  (called the *carrying capacity*). Consequently, we modify the exponential growth model (18.1) by introducing a growth rate  $\lambda'$  that decreases as the population  $N_i$  approaches  $N_*$ :

$$\lambda' = \lambda(N_* - N_i) \Rightarrow \frac{\Delta N_i}{\Delta t} = \lambda'(N_* - N_i)N_i \quad (18.2)$$

We expect that when  $N_i$  is small compared to  $N_*$ , the population will grow exponentially, but as  $N_i$  approaches  $N_*$ , the growth rate will decrease, eventually becoming negative if  $N_i$  exceeds  $N_*$ .

Equation (18.2) is one form of the *logistic map*. It is usually written as a relation between the number of bugs in future and present generations:

$$N_{i+1} = N_i + \lambda' \Delta t (N_* - N_i) N_i \quad (18.3)$$

$$= N_i (1 + \lambda' \Delta t N_*) \left[ 1 - \frac{\lambda' \Delta t}{1 + \lambda' \Delta t N_*} N_i \right] \quad (18.4)$$

The map looks simple when expressed in terms of natural variables:

$$x_{i+1} = \mu x_i (1 - x_i) \quad (18.5)$$

$$\mu \stackrel{\text{def}}{=} 1 + \lambda' \Delta t N_* \quad x_i \stackrel{\text{def}}{=} \frac{\lambda' \Delta t}{\mu} N_i \simeq \frac{N_i}{N_*} \quad (18.6)$$

where  $\mu$  is a dimensionless growth parameter and  $x_i$  is a dimensionless population variable. Observe that the *growth rate*  $\mu$  equals 1 when the breeding rate  $\lambda' = 0$ , and is otherwise expected to be larger than 1. If the number of bugs born per generation  $\lambda' \Delta t$  is large, then  $\mu \approx \lambda' \Delta t N_*$  and  $x_i \approx N_i/N_*$ . That is,  $x_i$  is essentially the fraction of the maximum population  $N_*$ . Consequently, we consider  $x$  values in the range  $0 \leq x_i \leq 1$ , where  $x = 0$  corresponds to no bugs, and  $x = 1$  to the maximum population. Note that there is clearly a linear and quadratic dependence of the RHS of (18.5) on  $x_i$ . In general, a map uses a function  $f(x)$  to map one number in a sequence to another,

$$x_{i+1} = f(x_i) \quad (18.7)$$

For the logistic map,  $f(x) = \mu x(1 - x)$ , with the quadratic dependence of  $f$  on  $x$  makes this a nonlinear map, while the dependence on only the one variable  $x_i$  makes it a *one-dimensional* map.

## 18.2

### Properties of Nonlinear Maps (Theory)

Rather than do some fancy mathematical analysis to determine properties of the logistic map [35], we prefer to have you study it directly on the computer by plotting  $x_i$  versus generation number  $i$ . Some typical behaviors are shown in Fig. 18.1. In A we see equilibration into a single population; in B we see oscillation between two population levels; in C we see oscillating among four levels, and in D we see a chaotic system. The initial population  $x_0$  is known as the *seed*, and as long as it is not equal to zero, its exact value generally has little effect on the population dynamics (similar to what we found when generating pseudo-random numbers). In contrast, the dynamics are unusually sensitive to the value of the growth parameter  $\mu$ . For those values of  $\mu$  at which the dynamics are complex, there may be extreme sensitivity to the initial condition,  $x_0$ , as well as on the exact value of  $\mu$ .

#### 18.2.1

##### Fixed Points

An important property of the map (18.5) is the possibility of the sequence  $x_i$  reaching a *fixed point* at which  $x_i$  remains or fluctuates about. We denote such fixed points as  $x_*$ . At a *one-cycle* fixed point, there is no change in the population from generation  $i$  to generation  $i + 1$ , and so it must satisfy

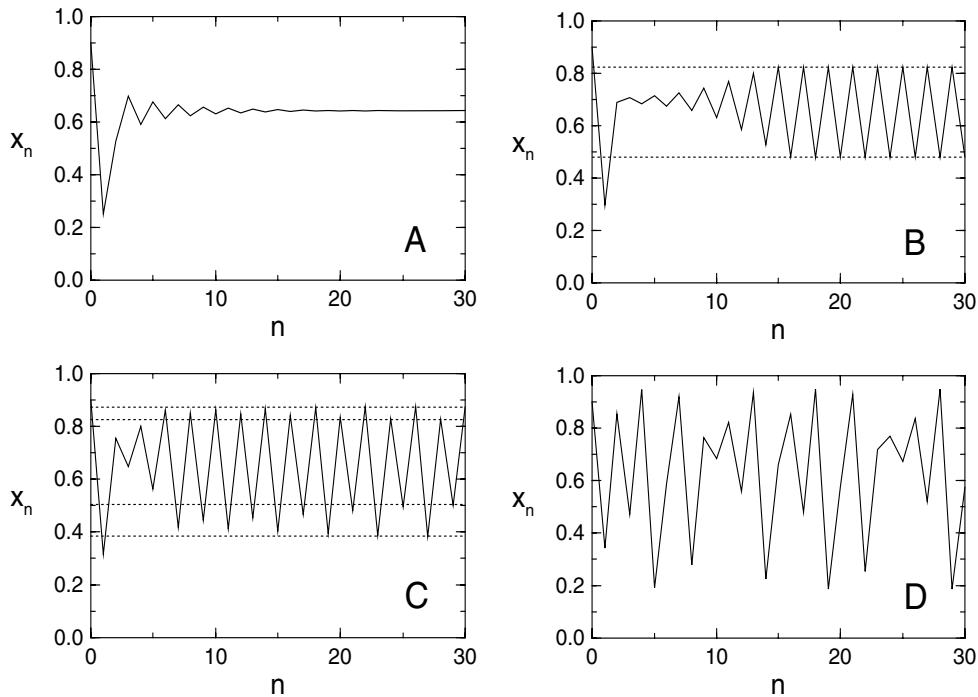
$$x_{i+1} = x_i = x_* \quad (18.8)$$

Using the logistic map (18.5) to relate  $x_{i+1}$  to  $x_i$ , yields the algebraic equation

$$\mu x_*(1 - x_*) = x_* \quad \Rightarrow \quad x_* = 0 \text{ or } x_* = \frac{\mu - 1}{\mu} \quad (18.9)$$

The nonzero fixed point  $x_* = (\mu - 1)/\mu$  corresponds to a stable population with a balance between birth and death that is reached regardless of the initial population (Fig. 18.1A). In contrast, the  $x_* = 0$  point is unstable and the population remains static only as long as no bugs exist; if even a few bugs are introduced, exponential growth occurs.

Further analysis based on the rate of change of  $x$  tells us that the stability of a population is determined by the magnitude of the derivative of the mapping



**Fig. 18.1** The insect population  $x_n$  versus generation number  $n$  for various growth rates. (A)  $\mu = 2.8$ , a period-one cycle. If the fixed point is  $x_n = 0$ , the system becomes extinct. (B)  $\mu = 3.3$ , a period-two cycle. (C)  $\mu = 3.5$ , a period-four cycle. (D)  $\mu = 3.8$ , a chaotic regime. (If  $\mu < 1$ , the population goes extinct.)

function  $f(x_i)$  at the fixed point [35]:

$$\left| \frac{df}{dx} \right|_{x_*} < 1 \text{ (stable)} \quad (18.10)$$

For the one-cycle of the logistic map (18.5), we have

$$\left. \frac{df}{dx} \right|_{x_*} = \mu - 2\mu x_* = \begin{cases} \mu & \text{stable at } x_* = 0 \text{ if } \mu < 1 \\ 2 - \mu & \text{stable at } x_* = \frac{\mu-1}{\mu} \text{ if } \mu < 3 \end{cases} \quad (18.11)$$

### 18.2.2

#### Period Doubling, Attractors

Equation (18.11) tells us that while the equation for fixed points (18.9) may be satisfied for all values of  $\mu$ , the populations will not be stable if  $\mu > 3$ . For  $\mu > 3$ , the system's long-term population *bifurcates* into two populations (a *two-cycle*), an effect known as *period doubling* (Fig. 18.1B). Because the system now acts as if it were attracted to two populations, these populations are

called *attractors* or *cycle points*. We can easily predict the  $x$  values for these two-cycle attractors by requiring that generation  $i + 2$  have the same population as generation  $i$ :

$$x_i = x_{i+2} = \mu x_{i+1}(1 - x_{i+1}) \quad (18.12)$$

$$\Rightarrow x_* = \frac{1 + \mu \pm \sqrt{\mu^2 - 2\mu - 3}}{2\mu} \quad (18.13)$$

We see that as long as  $\mu > 3$ , the square root produces a real number and thus physical solutions exist (complex or negative  $x_*$  values are unphysical).

We leave it for your computer explorations to discover how the system continues to double periods as for  $\mu$  continues to increase. In all cases the pattern is the same: one of the populations bifurcates into two.

### 18.3

#### Explicit Mapping Implementation

Program up the logistic map to produce a sequence of population values  $x_i$  as a function of the generation number  $i$ . These are called *map orbits*. The assessment consists of confirmation of Feigenbaum's observations [36] of the different behavior patterns shown in Fig. 18.1. These occur for growth parameter  $\mu = (0.4, 2.4, 3.2, 3.6, 3.8304)$  and seed population  $x_0 = 0.75$ . Identify the following on your graphs of  $x_i$  versus  $i$ :

1. **Transients:** Irregular behaviors before reaching a steady state. These transients differ for different seeds.
2. **Asymptotes:** In some cases the steady state is reached after only 20 generations, while for larger  $\mu$  values, hundreds of generations may be needed. These steady-state populations are independent of the seed.
3. **Extinction:** If the growth rate is too low,  $\mu \leq 1$ , the population dies off.
4. **Stable states:** The stable, single-population states attained for  $\mu < 3$  should agree with the prediction (18.9).
5. **Multiple cycles:** Examine the map orbits ( $x_i$  versus  $i$ ) for a growth parameter  $\mu$  increasing continuously through 3. Observe how the system continues to double periods as  $\mu$  increases. To illustrate, in Fig. 18.1C with  $\mu = 3.5$ , we notice a steady state in which the population alternates among four attractors (a *four-cycle*).
6. **Intermittency:** Observe simulations for  $3.8264 < \mu < 3.8304$ . Here the system should appear stable for a finite number of generations, then jump all around, only to become stable again.

7. **Chaos:** We define *chaos* as the deterministic behavior of a system displaying no discernible regularity. This may seem contradictory; if a system is deterministic, it must have step-to-step correlations (which, when added up, means long-range correlations); but if it is chaotic, the complexity of the behavior may hide the simplicity within. In an operational sense, a chaotic system is one with an extremely high sensitivity to parameters or initial conditions. This sensitivity to even minuscule changes is so high that it is impossible to predict the long-range behavior unless the parameters are known to infinite precision (a physical impossibility).

The system's behavior in the chaotic region is critically dependent on the exact value of  $\mu$  and  $x_0$ . Systems may start out with nearly identical values for  $\mu$  and  $x_0$ , but end up quite different. In some cases the complicated behaviors of nonlinear systems will be *chaotic*, but unless you have a bug in your program, they will not be random.<sup>2</sup>

- (a) Compare the long-term behaviors of starting with the two essentially identical seeds,  $x_0 = 0.75$ , and  $x'_0 = 0.75(1 + \epsilon)$ , where  $\epsilon \approx 2 \times 10^{-14}$ .
- (b) Repeat the simulation with  $x_0 = 0.75$ , and two essentially identical survival parameters,  $\mu = 4.0$ , and  $\mu' = 4.0(1 - \epsilon)$ . Both simulations should start off the same, but eventually diverge.

#### 18.4

##### Bifurcation Diagram (Assessment)

Computing and watching the population change with generation number gives a good idea of the basic dynamics, at least until they get too complicated to discern patterns. In particular, as the number of bifurcations keeps increasing and the system becomes chaotic, it is hard for our minds to see a simple underlying structure within the complicated behavior. One way to visualize what is going on is to concentrate on the attractors, that is, those populations that appear to attract the solutions and to which the solutions continuously return. A plot of these attractors (long-term iterates) of the logistic map as a function of the growth parameter  $\mu$  is an elegant way to summarize the results of extensive computer simulations.

A *bifurcation diagram* for the logistics map is given in Fig. 18.2, while one for a different map is given in Fig. 18.3. For each value of  $\mu$ , hundreds of iterations are made to make sure that all transients essentially die out, and

<sup>2</sup> You may recall from Chap. 11 that a random sequence of events does not even have step-by-step correlations.

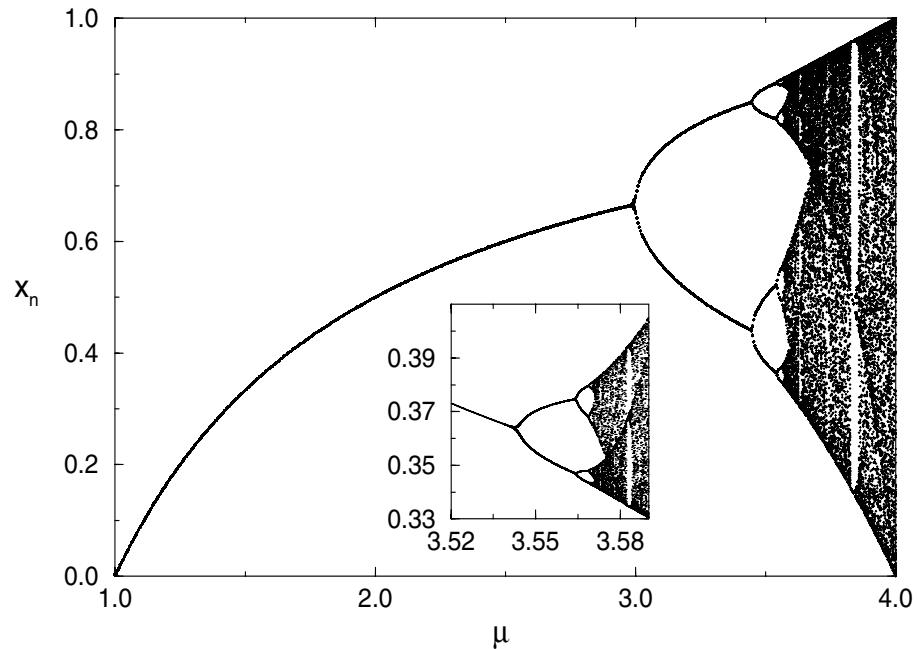
then the values  $(\mu, x_*)$  are written to a file for hundreds of iterations after that. If the system falls into an  $n$  cycle for this  $\mu$  value, then there should predominantly be  $n$  different values written to the file. Next, the value of the initial populations  $x_0$  is changed slightly, and the entire procedure repeated to ensure that no fixed points are missed. When done, your program will have stepped through all values of growth parameter  $\mu$ , and for each value of  $\mu$  will have stepped through all values of initial population  $x_0$ .

#### 18.4.1

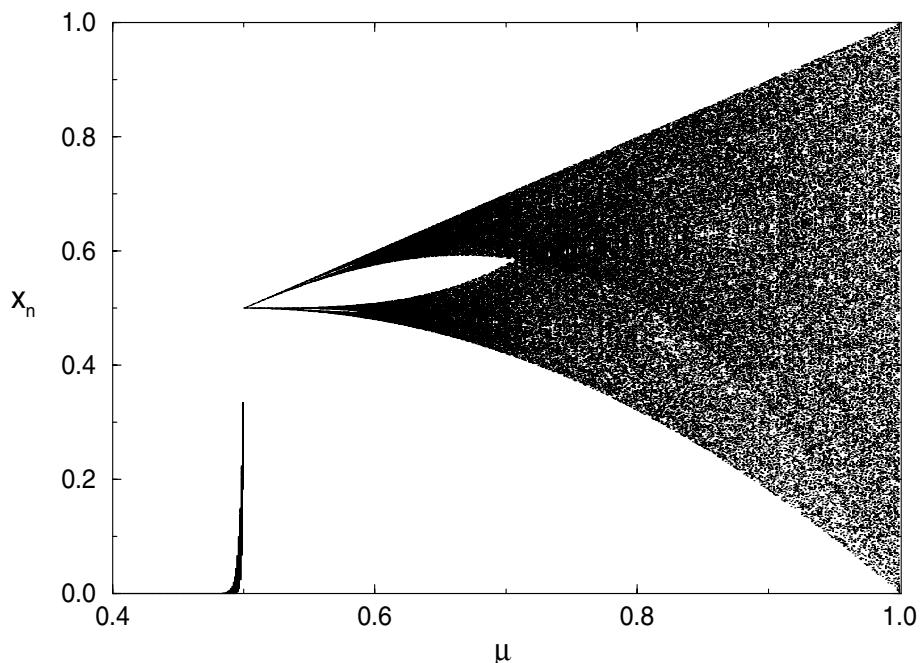
##### Bifurcation Diagram Implementation

The last part of this problem is to reproduce Fig. 18.2 at various levels of detail.<sup>3</sup> While the best way to make a visualization of this sort would be with visualization software that permits you to vary the intensity of each individual point on the screen, we simply plot individual points and have the density in each region determined by the number of points plotted there. When thinking about plotting many individual points to draw a figure, it is important to keep in mind that your screen resolution is  $\sim 100$  dots per inch and your laser printer resolution may be 300 dots per inch. This means that if you plotted

<sup>3</sup> You can listen to a sonification of this diagram on the CD.



**Fig. 18.2** The bifurcation plot, attractor populations versus growth rate, for the logistic map.



**Fig. 18.3** A bifurcation plot for the tent map,  $x_{i+1} = \mu(1 - 2|x_i - 1/2|)$ .

a point at each pixel, you would be plotting  $\sim 3000 \times 3000 \simeq 10$  million elements. *Beware*, this can require some time and may choke a printer. In any case, printing at finer resolution is a waste of time.

#### 18.4.2

##### Visualization Algorithm: Binning

1. Break up the range  $1 \leq \mu \leq 4$  into 1,000 steps and loop through them. These are the “bins” into which we will place the  $x_*$  values.
2. In order not to miss any structures in your bifurcation diagram, loop through a range of initial  $x_0$  values as well.
3. Wait at least 200 generations for the transients to die out, and then print out the next several hundred values of  $(\mu, x_*)$  to a file.
4. Print out your  $x_*$  values to no more than three or four decimal places. You will not be able to resolve more places than this on your plot, and this restriction will keep your output files smaller by permitting you to remove duplicates. It is hard to control the number of decimal places in output with Java’s standard print commands (although `printf` does permit control). A simple approach is to multiply the  $x_i$  values by 1000

and then throw away the part to the right of the decimal point. Because  $0 \leq x_n \leq 1$ , this means that  $0 \leq 100 * x_n \leq 1000$ , and you can throw away the decimal part by casting the resulting numbers as integers:

```
Ix[i] = (int)(1000*x[i]) // Convert to 0< ints < 1000
```

You may then divide by 1000 if you want floating point numbers.

5. You also need to remove duplicate values of  $(x, \mu)$  from your file (they just take up space and plot on top of each other). You can do that in Unix/Linus with the *sort -u* command.
  6. Plot up your file of  $x_*$  versus  $\mu$ . Use small symbols for the points and do not connect them.
  7. Enlarge sections of your plot and notice that a similar bifurcation diagram tends to be contained within each magnified portion (this is called *self-similarity*).
  8. Look over the series of bifurcations occurring at
- $$\mu_k \simeq 3, 3.449, 3.544, 3.5644, 3.5688, 3.569692, 3.56989, \dots \quad (18.14)$$
- The end of this series is a region of chaotic behavior.
9. Inspect the way this and other sequences begin and then end in chaos. The changes sometimes occur quickly and so you may have to make plots over a very small range of  $\mu$  values to see the structures.
  10. Close examination of Fig. 18.2 shows regions where, with a slight increase in  $\mu$ , a very large number of populations suddenly change to very few populations. Whereas these may appear to be artifacts of the video display, this is a real effect and these regions are called *windows*. Check that at  $\mu = 3.828427$ , chaos turns into a three-cycle population.

## 18.5

### Random Numbers via Logistic Map (Exploration)

There are claims that the logistic map in the chaotic region ( $\mu \geq 4$ )

$$x_{i+1} \simeq 4x_i(1 - x_i) \quad (18.15)$$

can be used to generate random numbers [37]. Although successive  $x_i$ 's are correlated, if the population for every  $\sim 6$ th generation is examined, the correlations die out and effectively random numbers result. To make the sequence more uniform, a trigonometric transformation is used:

$$y_i = \frac{1}{\pi} \cos^{-1}(1 - 2x_i) \quad (18.16)$$

Use the random-number tests discussed in Chap. 11 to test this claim.

### 18.6 Feigenbaum Constants (Exploration)

Feigenbaum discovered that the sequence of  $\mu_k$  values (18.14) at which bifurcations occur follow a regular pattern [36]. Specifically, it converges geometrically when expressed in terms of the distance between bifurcations  $\delta$ :

$$\mu_k \rightarrow \mu_\infty - \frac{c}{\delta^k} \quad \delta = \lim_{k \rightarrow \infty} \frac{\mu_k - \mu_{k-1}}{\mu_{k+1} - \mu_k} \quad (18.17)$$

Use your sequence of  $\mu_k$  values to determine the constants in (18.17) and compare to those found by Feigenbaum:

$$\mu_\infty \simeq 3.56995 \quad c \simeq 2.637 \quad \delta \simeq 4.6692 \quad (18.18)$$

Amazingly, the value of the *Feigenbaum constant*  $\delta$  is universal for all second-order maps.

### 18.7 Other Maps (Exploration)

Bifurcations and chaos are characteristic properties of nonlinear systems. Yet systems can be nonlinear in a number of ways. Table 18.1 lists four maps that generate  $x_i$  sequences containing bifurcations. The tent map (Fig. 18.3) derives its nonlinear dependence from the absolute value operator, while the logistics map is a subclass of the ecology map. Explore the properties of these other maps, and note the similarities and differences.

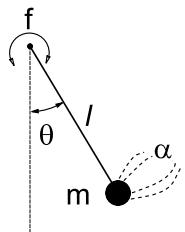
**Tab. 18.1** Several nonlinear maps to explore.

Name	$f(x)$	Name	$f(x)$
Logistic	$\mu x(1-x)$	Tent	$\mu(1-2 x-1/2 )$
Ecology	$xe^{\mu(1-x)}$	Quartic	$\mu[1-(2x-1)^4]$

## 19

## Differential Chaos in Phase Space

In Chap. 18 on bugs, we discovered that a simple nonlinear difference equation yields solutions that may be simple, complicated, or chaotic. In this chapter, we continue our study of nonlinear behavior, this time for two systems described by differential equations, the driven realistic pendulum, and coupled predator-prey populations. Because chaotic behavior may resemble noise, it is important to be confident that the unusual behaviors arise from physics and not numerics. Before we explore the solutions, we provide some theoretical background in the use of phase-space plots for revealing the beauty and simplicity underlying complicated behaviors. Our emphasis there is on using phase space as an example of the usefulness of an abstract space displaying the simplicity that often underlies complexity. Our study is based on the description in [35], on the analytic discussion of the parametric oscillator [31], and on a similar study of the vibrating pivot pendulum [17].



**Fig. 19.1** A pendulum of length  $l$  driven through air by an external, sinusoidal torque. The strength of the torque is given by  $f$  and that of air resistance by  $\alpha$ .

## 19.1

## Problem: A Pendulum Becomes Chaotic (Differential Chaos)

Your **problem** is to describe the motion of this pendulum, first with the driving torque turned off, but the initial velocity is large enough to send the pendulum over the top, and then when the driving torque is turned on. In Fig. 19.1, we see a pendulum of length  $l$ , driven by an external, sinusoidal torque  $f$  through air with a coefficient of drag  $\alpha$ . Because there is no restriction that the angular displacement  $\theta$  be small, we call this a *realistic* pendulum.

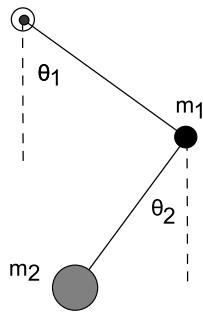


Fig. 19.2 A double pendulum.

**Alternative Problem:** For those of you who have already studied the realistic pendulum, study instead the double pendulum without any small-angle approximation (Fig. 19.2 and animation `DoublePend.mpg` on the CD). A double pendulum has a second pendulum connected to the first, and because each pendulum acts as a driving force to the other, we need not include an external driving torque to produce a chaotic system (there are enough degrees of freedom without it).

## 19.2

### Equation of Chaotic Pendulum

The theory of the *chaotic pendulum* is just that of a pendulum with friction and a driving torque (Fig. 19.1), but with no small-angle approximation. Newton's laws of rotational motion tell us that the sum of the gravitational torque  $-mgl \sin \theta$ , the frictional torque  $-\beta\dot{\theta}$ , and the external torque  $\tau_0 \cos \omega t$  equals the moment of inertia of the pendulum times its angular acceleration:

$$I \frac{d^2\theta}{dt^2} = -mgl \sin \theta - \beta \frac{d\theta}{dt} + \tau_0 \cos \omega t \quad (19.1)$$

$$\Rightarrow \frac{d^2\theta}{dt^2} = -\omega_0^2 \sin \theta - \alpha \frac{d\theta}{dt} + f \cos \omega t \quad (19.2)$$

where in (19.2) we obtain the traditional form with  $\omega_0 = mgl/I$ ,  $\alpha = \beta/I$ , and  $f = \tau_0/I$  [35]. Equation (19.2) is a second-order, time-dependent, nonlinear differential equation. The nonlinearity arises from the  $\sin \theta$ , as opposed to the  $\theta$ , dependence of the gravitational torque. The parameter  $\omega_0$  is the natural frequency of the system arising from the restoring torque,  $\alpha$  is a measure of the strength of friction, and  $f$  is a measure of the strength of the driving torque. In the standard ODE form,  $dy/dt = y$  (Chap. 15), we have two simultaneous first-order equations:

$$\frac{dy^{(0)}}{dt} = y^{(1)} \quad \frac{dy^{(1)}}{dt} = -\omega_0^2 \sin y^{(0)} - \alpha y^{(1)} + f \cos \omega t \quad (19.3)$$

where  $y^{(0)} = \theta(t)$  and  $y^{(1)} = d\theta(t)/dt$

### 19.2.1

#### Oscillations of a Free Pendulum

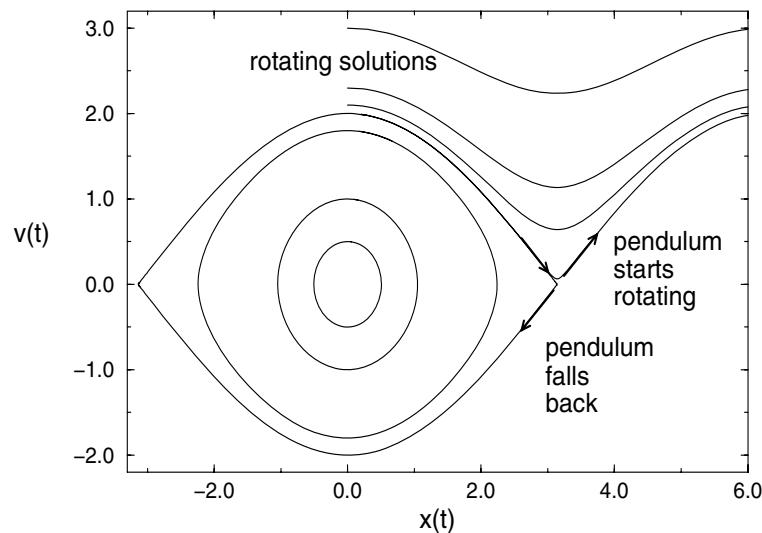
If we ignore friction and external torques, (19.2) takes the simple form

$$\frac{d^2\theta}{dt^2} = -\omega_0^2 \sin \theta \quad (19.4)$$

If displacements are small, we can approximate  $\sin \theta$  by  $\theta$  and obtain the linear equation of simple harmonic motion with frequency  $\omega_0$ :

$$\frac{d^2\theta}{dt^2} \simeq -\omega_0^2 \theta \quad \Rightarrow \quad \theta(t) = \theta_0 \sin(\omega_0 t + \phi) \quad (19.5)$$

In Chap. 15, we have studied how nonlinearities produce anharmonic oscillations, and, indeed, (19.4) is another good candidate for such studies. As before, we expect solutions of (19.4) for the free realistic pendulum to be periodic, but with a frequency  $\omega$  that equals  $\omega_0 = 2\pi/T_0$ , only for small oscillations. Furthermore, because the restoring torque  $mgl \sin \theta \simeq mgl(\theta - \theta^3/3)$ , is less than the  $mgl\theta$  assumed in a harmonic oscillator, realistic pendulum swing slower (longer periods) as their angular displacements are made larger.



**Fig. 19.3** Phase-space trajectories for a plane pendulum including “over the top” or rotating solutions. (Although not shown completely, the trajectories are symmetric with respect to vertical and horizontal reflections through the origin.)

## 19.2.2

**Pendulum's "Solution" as Elliptic Integrals**

The analytic solution to the realistic, free pendulum is a text book problem [23, 26,31]; except it is hardly a solution, and it is hardly analytic. The "solution" is based on energy being a constant (integral) of the motion. For simplicity, we start the pendulum off at rest from its maximum displacement  $\theta_m$ . Because the initial energy is all potential, we know that the total energy of the system is its initial potential energy (Fig. 19.1),

$$E = PE(0) = mgl - mgl \cos \theta_m = 2mgl \sin^2(\theta_m/2) \quad (19.6)$$

Yet since  $E = KE + PE$  is a constant, we can write for any value of  $\theta$

$$\begin{aligned} 2mgl \sin^2 \frac{\theta_m}{2} &= \frac{1}{2}I(d\theta/dt)^2 + 2mgl \sin^2 \frac{\theta}{2} \\ \Rightarrow \frac{d\theta}{dt} &= 2\omega_0 \left[ \sin^2 \frac{\theta_m}{2} - \sin^2 \frac{\theta}{2} \right]^{1/2} \Rightarrow \frac{dt}{d\theta} = \frac{T_0/\pi}{\left[ \sin^2 \frac{\theta_m}{2} - \sin^2 \frac{\theta}{2} \right]^{1/2}} \\ \Rightarrow \frac{T}{4} &= \frac{T_0}{4\pi} \int_0^{\theta_m} \frac{d\theta}{\left[ \sin^2 \frac{\theta_m}{2} - \sin^2 \frac{\theta}{2} \right]^{1/2}} = \frac{T_0}{4\pi \sin \theta_m} F\left(\frac{\theta_m}{2}, \frac{\theta}{2}\right) \end{aligned} \quad (19.7)$$

$$\Rightarrow T \simeq T_0 \left[ 1 + \frac{1}{4} \sin^2 \frac{\theta_m}{2} + \frac{9}{64} \sin^4 \frac{\theta_m}{2} + \dots \right] \quad (19.8)$$

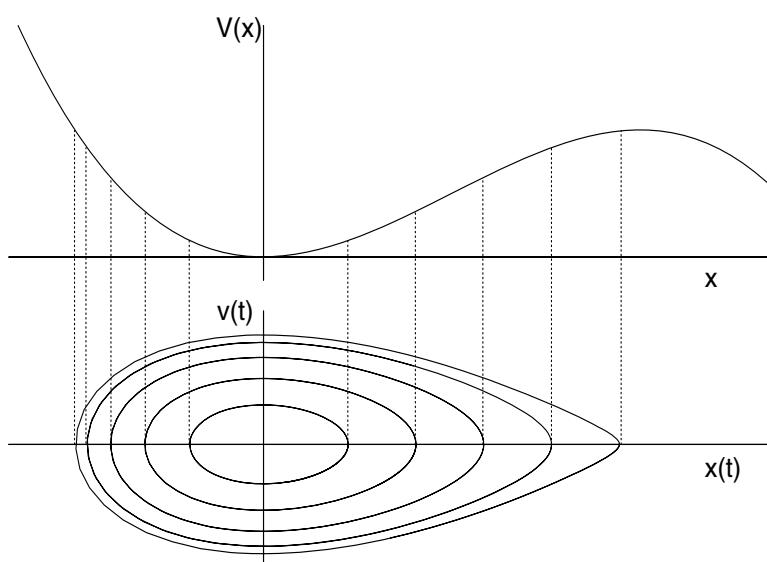
where we have assumed that it takes  $T/4$  for the pendulum to travel from  $\theta = 0$  to  $\theta_m$ . The integral in (19.7) is an *elliptic integral of the first kind*. If you think of an elliptic integral as a generalization of a trigonometric function, then this is a closed-form solution; otherwise it's an integral needing computation. The series expansion (19.8) is obtained by expanding the denominator and integrating term by term. It tells us, for example, that an amplitude of  $80^\circ$  leads to a 10% slowdown of the pendulum. In contrast, we will determine the period empirically by solving for  $\theta(t)$  and counting times between one  $\theta = 0$  to the next.

## 19.2.3

**Implementation and Test: Free Pendulum**

As a preliminary to the solution of the full equation (19.2), modify your `rk4` program to solve (19.4) for the free oscillations of a realistic pendulum.

1. Start your pendulum off at  $\theta = 0$  with  $\dot{\theta}(0) \neq 0$ . Gradually increase  $\dot{\theta}(0)$  to increase the importance of nonlinearities.
2. Test your program for the linear case ( $\sin \theta \rightarrow \theta$ ), verify that



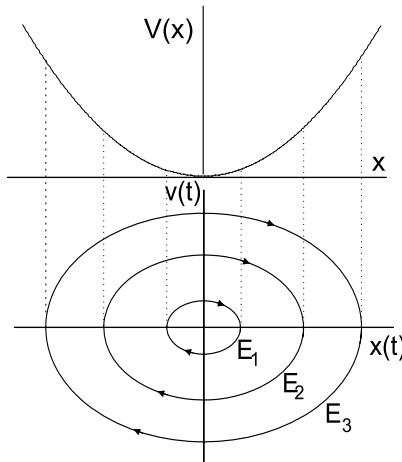
**Fig. 19.4** A potential-energy plot for a nonharmonic oscillator (top) and a phase-space plot (lower) for the same nonharmonic oscillator. Note that the ellipse-like figures are neither ellipses nor symmetric with respect to the  $v$  axis. The different orbits correspond to different energies, as indicated by the limits within the potentials.

- (a) your solution is harmonic with frequency  $\omega_0 = 2\pi/T_0$ , and
  - (b) the frequency of oscillation is independent of the amplitude.
3. Devise an algorithm to determine the period  $T$  of the oscillation by counting the time it takes for four successive passes of the amplitude through  $\theta = 0$ . (You need *four* passes because a general oscillation may not be symmetric about the origin.) Test your algorithm for simple harmonic motion where you know  $T_0$ .
  4. For the realistic pendulum, observe the change in period as a function of increasing initial energy or displacement. Plot your observations along with (19.8).
  5. Verify that as the initial KE approaches  $2mgl$ , the motion remains oscillatory, but not harmonic.
  6. At  $E = 2mgl$  (the *separatrix*), the motion changes from oscillatory to rotational (“over the top” or “running”). See how close you can get to the separatrix and try to verify that at precisely this energy it takes an infinite time for a single oscillation.

## 19.3

## Visualization: Phase-Space Orbits

The conventional solution to an equation of motion is the position  $x(t)$  and the velocity  $v(t)$  as functions of time. Often, behaviors that appear complicated as functions of time, appear simpler when viewed in an abstract space called *phase space*, where the ordinate is the velocity  $v(t)$  and the abscissa is the position  $x(t)$  (Figs. 19.5 and 19.4). As we see from the figures, when viewed in phase space, solutions of the equations of motion form geometric objects that are easy to recognize.



**Fig. 19.5** A phase-space plot of velocity versus position for a harmonic oscillator. Because the ellipses close, the system must be periodic. The different orbits correspond to different energies, as indicated by the limits within the potentials.

The position and velocity of a free harmonic oscillator are given by the trigonometric functions:

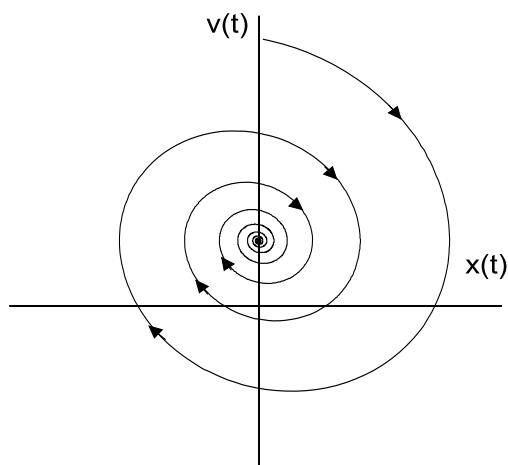
$$x(t) = A \sin(\omega t) \quad v(t) = \frac{dx}{dt} = \omega A \cos(\omega t) \quad (19.9)$$

When substituted into the total energy, we obtain two important results:

$$E = KE + PE = (\frac{1}{2}m) v^2 + (\frac{1}{2}\omega^2 m^2) x^2 \quad (19.10)$$

$$= \frac{\omega^2 m^2 A^2}{2m} \cos^2(\omega t) + \frac{1}{2}\omega^2 m^2 A^2 \sin^2(\omega t) = \frac{1}{2}m\omega^2 A^2 \quad (19.11)$$

The first equation, being that of an ellipse, proves that the harmonic oscillator follows closed elliptical orbits in phase space, with the size of the ellipse increasing with the system's energy. The second equation proves that the total energy is a constant of the motion. Different initial conditions having the

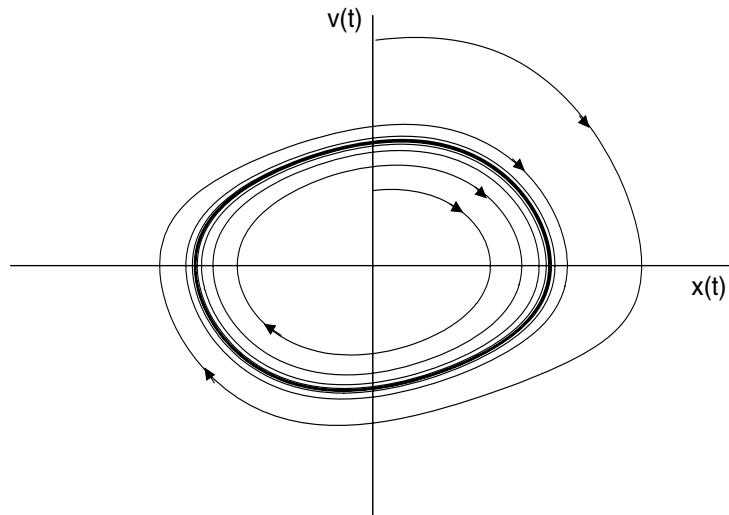


**Fig. 19.6** Phase-space trajectories for a particle in a repulsive potential. Notice the absence of trajectories in the regions forbidden by energy conservation.

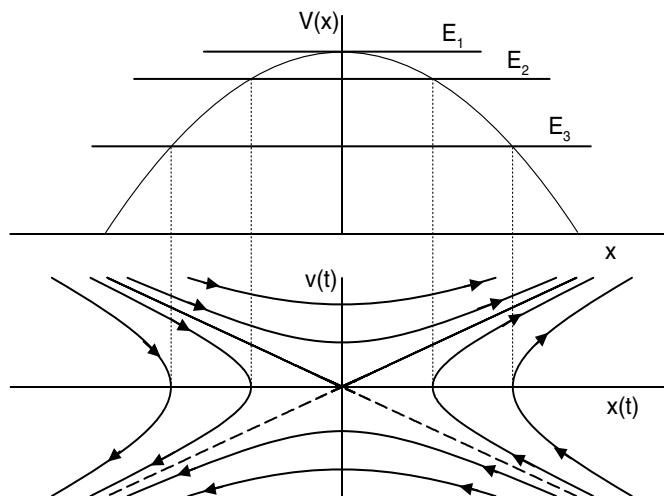
same energy start at different places on the same ellipse, but transverse the same orbits.

In Figs. 19.3–19.8, we show some typical phase-space structures. *Study these figures and their captions, and note the following:*

- For anharmonic oscillations, the orbits will still be ellipse like, but with angular corners that become more distinct with increasing nonlinearity.



**Fig. 19.7** A phase-space orbit for an oscillator with friction. The system eventually comes to rest at the origin.



**Fig. 19.8** Two phase-space trajectories corresponding to solutions of the van der Pol equation. One trajectory approaches the limit cycle (the dark curve) from the inside, while the other approaches it from the outside.

- Closed trajectories describe periodic oscillations (the same  $(x, v)$  occur again and again), with clockwise motion.
- Open orbits correspond to nonperiodic or “running” motion (a pendulum rotating like a propeller).
- Regions of space where the potential is repulsive, lead to open trajectories in phase space (Fig. 19.8).
- As seen in Fig. 19.3, the separatrix corresponds to the trajectory in phase space that separates open and closed orbits. Motion on the separatrix is indeterminant as the pendulum may balance at the maxima of  $V(\theta)$ .
- Friction may cause the energy to decrease with time and the phase-space orbit to spiral into a *fixed point* (Fig. 19.6).
- For certain parameters, a closed *limit cycle* (Fig. 19.8) occurs in which the energy pumped in by the external torque exactly balances that lost by friction.
- Because solutions for different initial conditions are unique, different orbits do not cross. Nonetheless, open orbits do come together at the points of unstable equilibrium (*hyperbolic points* in Figs. 19.8 and 19.3), where an indeterminacy exists.

## 19.3.1

**Chaos in Phase Space**

It is easy to solve the nonlinear ODE (19.3) on the computer. It is not so easy, however, to understand the solutions because they are so rich (*alias* for complex and highly sensitive to initial conditions). The solutions are easier to understand in phase space, and particularly if you learn to recognize some characteristic structures there. Actually, there are a number of “tools” that can be used to decide if a system is chaotic, in contrast to just complex. Phase-space structures is one of them, and determination the Lyupanov coefficient [3], is another. What is important is being able to deduce the simplicity that lies within the complicated behavior.

What is surprising is that even though the ellipse-like figures we have been discussing were deduced for free systems with no friction and no driving torque, similar structures continue to exist for driven systems with friction. The trajectories may not remain on a single structure for all times, but they are *attracted* to them. In contrast to periodic motion, which corresponds to closed figures in phase space, random motion appears as a diffuse cloud filling an entire energetically accessible region. Complex, or chaotic motion falls someplace in between (Fig. 19.9). If viewed for long times and many initial conditions, chaotic flows through phase space, while resembling the familiar geometric figures, may contain dark or diffuse *bands* in places rather than single lines. The continuity of trajectories within bands means that there is a continuum of solutions possible, and that the system flows continuously among the different trajectories forming the band. The transitions among solutions is what causes the coordinate space solutions to appear chaotic, and is what makes them hypersensitive to initial conditions (the slightest change in which causes the system to flow to nearby trajectories).

So even though the motion may be chaotic, the definite shapes of the phase-space structures means that there is a well-defined and simple underlying order within chaos. Pick out the following phase-space structures in your simulations:

**Limit cycles.** When the chaotic pendulum is driven by a not-too-large driving torque, it is possible to pick the magnitude for this torque such that after the initial transients die off, the average energy put into the system during one period exactly balances the average energy dissipated by friction during that period (Fig. 19.8):

$$\langle f \cos \omega t \rangle = \left\langle \alpha \frac{d\theta}{dt} \right\rangle = \left\langle \alpha \frac{d\theta}{dt}(0) \cos \omega t \right\rangle \quad \Rightarrow \quad f = \alpha \frac{d\theta}{dt}(0) \quad (19.12)$$

This leads to *limit cycles*, which appear as closed ellipse-like figures, even in the presence of friction and driving torque. Yet the solution may be unstable and make sporadic jumps between limit cycles.

**Predictable attractors.** Well-defined, fairly simple periodic behaviors that are not particularly sensitive to initial conditions. These are orbits, such as fixed points and limit cycles, into which the system settles. If your location in phase space is near a predictable attractor, ensuing times will bring you to it.

**Strange attractors:** Well-defined, yet complicated, semiperiodic behaviors that appear to be uncorrelated to the motion at an earlier time. They are distinguished from predictable attractors by being fractal (Chap. 20), chaotic, and highly sensitive to initial conditions [22]. Even after millions of oscillations, the motion remains *attracted* to them.

**Chaotic paths:** Regions of phase space that appear as filled-in bands rather than lines. Continuity within the bands implies complicated behaviors, yet still with simple underlying structure.

**Mode locking:** When the magnitude  $f$  of the driving torque is larger than that for a limit cycle, (19.12), the driving torque overpowers the natural oscillations, and the steady-state motion is at the frequency of the driver. This is *mode locking*. While mode locking can occur for linear or nonlinear systems, for nonlinear systems the driving torque may lock onto the system by exciting its overtones, leading to a rational relation between driving frequency and natural frequency:

$$\frac{\omega}{\omega_0} = \frac{n}{m} \quad n, m = \text{integers} \quad (19.13)$$

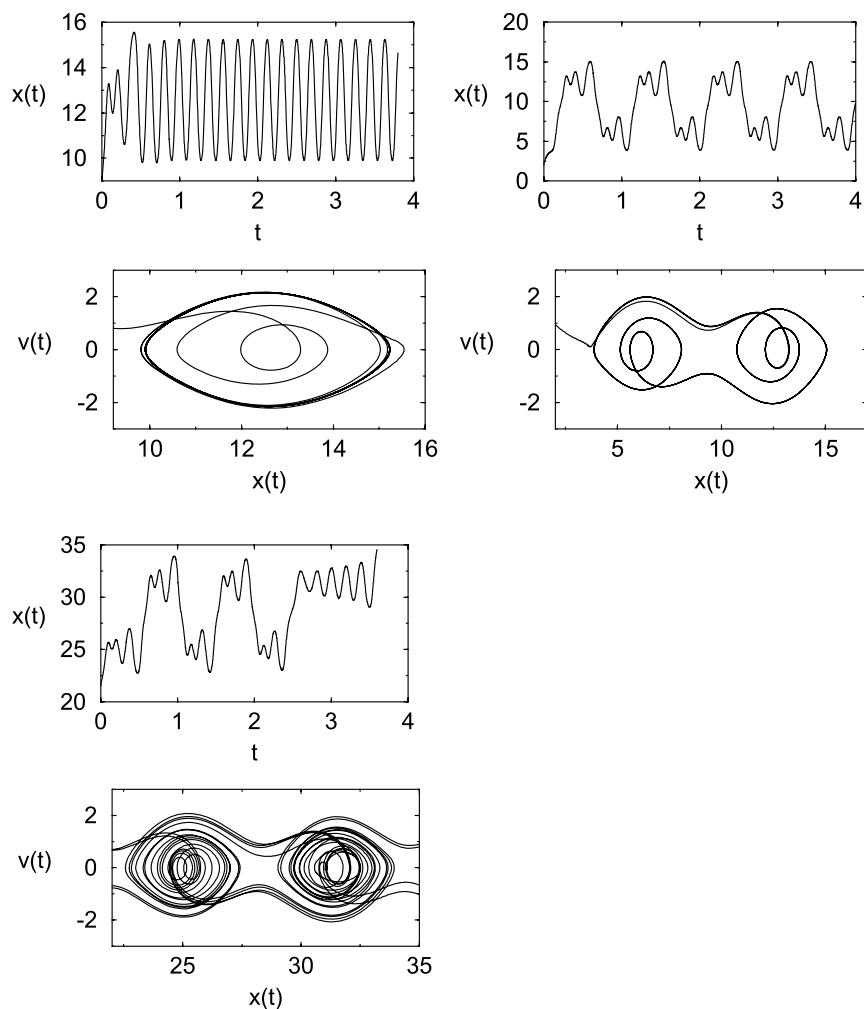
**Butterfly effects:** One of the classic quips about the hypersensitivity of chaotic systems to initial conditions is that the weather pattern in North America is hard to predict well because it is sensitive to the flapping of butterfly wings in South America. Although this appears to be counter intuitive because we know that systems with essentially identical initial conditions should behave the same, eventually the systems diverge.

### 19.3.2

#### Assessment in Phase Space

The challenge with simulations of the chaotic pendulum (19.3) is that the 4D parameter space  $(\omega_0, \alpha, f, \omega)$  is immense. For normal behavior, sweeping through  $\omega$  should show resonances and beating; sweeping through  $\alpha$  should show underdamping, critical damping, and overdamping; sweeping through  $f$  should show mode locking (for the right values of  $\omega$ ). All these behaviors can be found in the solution of your differential equation, yet because they get mixed together, your solution may exhibit quite complex behavior.

*In this assessment you should try to reproduce the behaviors shown in the phase-space diagrams of Fig. 19.9. Beware, because the system is chaotic, your results may be very sensitive to the exact values of the initial conditions, or to the precision of your integration routine. We suggest that you experiment; start with the parameter*



**Fig. 19.9** Position and phase-space plots for the chaotic pendulum with  $\omega_0 = 1$ ,  $\alpha = 0.2$ ,  $f = 0.52$ , and  $\omega = 0.666$ . The angular position is  $x(t)$  and the angular velocity is  $v(t)$ . The chaotic regions are the dark bands in the bottom figure.

values we used to produce our plots, and then observe the effects of making very small changes in parameter until you obtain different modes of behavior. Absence of agreement with our exact values does not imply that anything is “wrong.”

1. Take your solution to the realistic pendulum ( $\sin \theta$ ), and include friction, making  $\alpha$  an input parameter. Run for a variety of initial conditions, including over-the-top ones; you should see spirals like those in Figs. 19.7 and 19.8.

2. Next, verify that with no friction, but with a very small driving torque, you obtain a perturbed ellipse.
3. Set driving torque's frequency to be close to the natural frequency  $\omega_0$  of the pendulum, and produce beats. Note that you may need to adjust the magnitude and phase of the driving torque to avoid an "impedance mismatch" between the pendulum and driver.
4. Finally, include friction and a variable-frequency driving torque. Scan  $\omega$  to produce a nonlinear resonance (which looks like beating).
5. **Explore chaos:** Start off with the initial conditions we used in Fig. 19.9,

$$(x_0, v_0) = (-0.0885, 0.8) \quad (-0.0883, 0.8) \quad (-0.0888, 0.8)$$

To save time and storage, you may want to use a larger time step for plotting than that used to solve the differential equations.

6. Indicate which parts of the plots correspond to transients.
7. Ensure that you have found
  - (a) a period-three limit cycle where the pendulum jumps between three major orbits in phase space;
  - (b) a running solution where the pendulum keeps going over the top;
  - (c) chaotic motion in which some paths in phase space appear as bands.
8. Look for the "butterfly effect." Start two pendulums off with identical positions, but velocities that differ by one part in a thousand. Notice how the initial motion is essentially identical, but how at some later time the motions diverge.

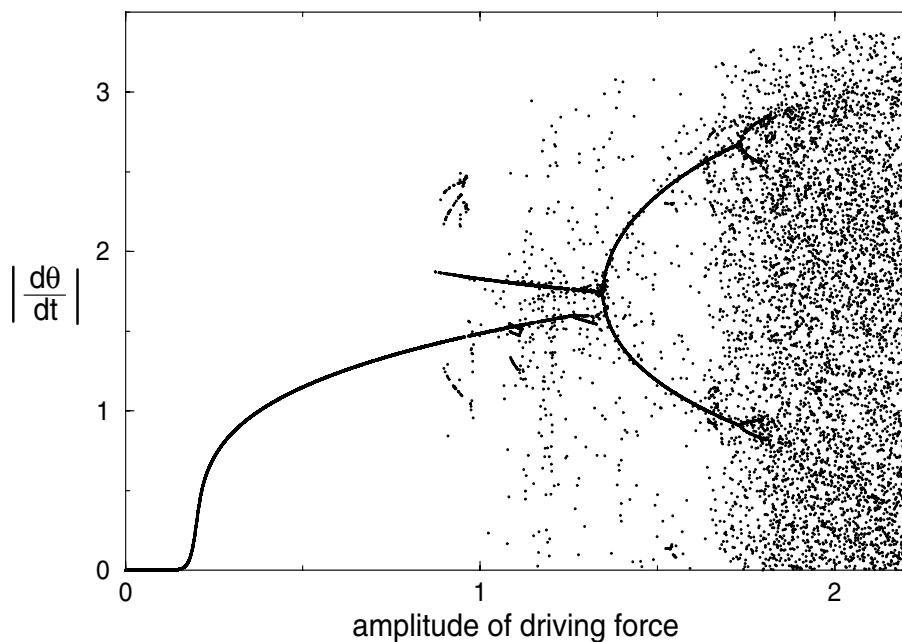
#### 19.4

##### Assessment: Fourier Analysis of Chaos

We have seen that a realistic pendulum feels a restoring torque,  $\tau_g \propto \sin \theta \simeq \theta - \theta^3/3! + \theta^5/5! + \dots$ , that contains nonlinear terms which lead to nonharmonic behavior. In addition, when a realistic pendulum is driven by an external sinusoidal torque, the pendulum may mode lock with the driver and so oscillate at a frequency that is rationally related to the driver's. Consequently, the behavior of the realistic pendulum is expected to be a combination of various periodic behaviors, with discrete jumps between different modes.

In this assessment you should determine the Fourier components present in the pendulum's complicated and chaotic behaviors. You should show that a

three-cycle structure, for example, contains three major Fourier components, while a five-cycle has five. You should also notice that when the pendulum goes over the top, its spectrum contains a steady-state (“dc”) component.



**Fig. 19.10** Bifurcation diagram for the damped pendulum with a vibrating pivot. The ordinate is  $|d\theta/dt|$ , the absolute value of the instantaneous angular velocity at the beginning of the period of the driver, and the abscissa is the magnitude of the driving force  $d$ . Note that the heavy line results from overlapping of points, not from connecting the points.

1. Dust off your program which analyzes a  $y(t)$  into Fourier components. Alternatively, you may use a Fourier analysis tool contained in your graphics program or system library (e.g., Grace and OpenDX).
2. Apply your analyzer to the solution of the chaotic pendulum for the cases where there are one-, three-, and five-cycle structures in phase space. Deduce the major frequencies contained in these structures.
3. Try to deduce a relation between the Fourier components, the natural frequency  $\omega_0$ , and the driving frequency  $\omega$ .
4. A classic signal of chaos is a broadband, although not necessarily flat, Fourier spectrum. Examine your system for parameters that give chaotic behavior and verify this statement.

### 19.5

#### Exploration: Bifurcations in Chaotic Pendulum

We have already stated that a chaotic system contains a finite number of dominant frequencies that occur sequentially, in contrast to linear systems where they occur simultaneously. Consequently, if we sample the instantaneous angular velocity  $\dot{\theta} = d\theta/dt$  of a chaotic system at various instances of time, we should get different values each time, but with the major Fourier components occurring more often than others.<sup>1</sup> These are the frequencies to which the system is *attracted*. Yet if we change some parameter of the system, then we expect the dominant components to change, with some new ones entering, and some old ones departing. That being the case, if we make a scatter plot of the frequencies sampled for all times at one particular value of the driving force, and then change the magnitude of the driving force slightly and sample frequencies again, the resulting plot should show distinctive patterns of frequencies. That a bifurcation diagram is obtained by making such a plot, and that it is similar to the bifurcation diagram for bug populations studied in Unit I, is one of the mysteries of life.

In the scatter plot in Fig. 19.10, we sampled  $\dot{\theta}$  for the motion of a chaotic pendulum with a vibrating pivot point (in contrast to our usual vibrating external torque). This pendulum is similar to our chaotic one (19.2), but with the driving force depending on  $\sin \theta$ :

$$\frac{d^2\theta}{dt^2} = -\alpha \frac{d\theta}{dt} - (\omega_0^2 + f \cos \omega t) \sin \theta \quad (19.14)$$

Essentially, the acceleration of the pivot is equivalent to a sinusoidal variation of  $g$  or  $\omega_0^2$ . Analytic [31, Sections 25–30] and numeric [17, 38] studies of this system exist.

We leave it to you to produce a bifurcation diagram from your chaotic pendulum with the same technique. We obtained the bifurcation diagram (Fig. 19.10) by following these steps (a modification of those in [38]):

1. Use the initial conditions:  $\theta(0) = 1$  and  $\dot{\theta}(0) = 1$ .
2. Set  $\alpha = 0.1$ ,  $\omega_0 = 1$ ,  $\omega = 2$ , and let  $f$  vary through the range in Fig. 19.10.
3. For each value of  $f$ , wait 150 periods of the driver before sampling to permit transients to die off. Sample the instantaneous angular velocity  $\dot{\theta}$  for 150 times whenever the driving force passes through  $\theta = 0$ .
4. Plot the 150 values of  $|\dot{\theta}|$  versus  $f$ .
5. Repeat the procedure for each new value of  $f$ .

<sup>1</sup> We refer to this angular velocity as  $\dot{\theta}$  since we have already used  $\omega$  for the frequency of the driver and  $\omega_0$  for the natural frequency.

## 19.6

### Exploration: Another Type of Phase-Space Plot

Imagine that you have measured the displacement of some system as a function of time. Your measurements appear to indicate characteristic nonlinear behaviors, and you would like to check this by making a phase-space plot, but without going to the trouble of measuring the conjugate momenta to plot versus displacement. Amazingly enough, one may also plot  $x(t + \tau)$  versus  $x(t)$  as a function of time to obtain a phase-space plot [39]. Here  $\tau$  is a *lag time* that should be chosen as some fraction of a characteristic time for the system under study. While this may not seem like a valid way to make a phase-space plot, recall the forward-difference approximation for the derivative,

$$v(t) = \frac{dx(t)}{dt} \simeq \frac{x(t + \tau) - x(t)}{\tau} \quad (19.15)$$

We see that plotting  $x(t + \tau)$  vs.  $x(t)$  is equivalent to plotting  $v(t)$  vs.  $x(t)$ .

**Exercise:** Create a phase-space plot from the output of your chaotic pendulum by plotting  $\theta(t + \tau)$  versus  $\theta(t)$  for a large range of  $t$  values. Explore how the graphs change for different values of the lag time  $\tau$ . Compare your results to the conventional phase-space plots you obtained previously.

## 19.7

### Further Explorations

1. The nonlinear behavior in once-common objects such as vacuum tubes and metronomes are described by the **van der Pool equation**:

$$\frac{d^2x}{dt^2} + \mu(x^2 - x_0^2)\frac{dx}{dt} + \omega_0^2x = 0 \quad (19.16)$$

The behavior predicted for these systems is *self-limiting* because the equation contains a limit cycle that is also a predictable attractor. You can think of (19.16) as describing an oscillator with  $x$ -dependent damping (the  $\mu$  term). If  $x > x_0$ , friction slows the system down; if  $x < x_0$ , friction speeds the system up. A resulting phase-space orbit is similar to that shown in Fig. 19.8. The heavy curve is the *limit cycle*. Orbits internal to the limit cycle spiral out until they reach the limit cycle; orbit external to it spiral in.

2. **Duffing oscillator:** Another damped and driven nonlinear oscillator is

$$\frac{d^2\theta}{dt^2} - \frac{1}{2}\theta(1 - \theta^2) = -\alpha\frac{d\theta}{dt} + f \cos \omega t \quad (19.17)$$

While similar to the chaotic pendulum, it is easier to find multiple attractors with this oscillator [40].

3. **Lorenz attractor:** In 1962 Lorenz [24] was looking for a simple model for weather prediction, and simplified the heat-transport equations to

$$\frac{dx}{dt} = 10(y - x) \quad \frac{dy}{dt} = -xz + 28x - y \quad \frac{dz}{dt} = xy - \frac{8}{3}z \quad (19.18)$$

The solution of these simple nonlinear equations gave the complicated behavior that has led to the modern interest in chaos (after considerable doubt regarding the reliability of the numerical solutions).

4. **A 3D computer fly:** Plot, in 3D space, the equations

$$x = \sin ay - z \cos bx \quad y = z \sin cx - \cos dy \quad z = e \sin x \quad (19.19)$$

Here the parameter  $e$  controls the degree of apparent randomness.

5. **Hénon-Heiles potential:** The potential and Hamiltonian

$$V(x, y) = \frac{1}{2}x^2 + \frac{1}{2}y^2 + x^2y - \frac{1}{3}y^3 \quad H = \frac{1}{2}p_x^2 + \frac{1}{2}p_y^2 + V(x, y) \quad (19.20)$$

are used to describe three astronomical objects interacting. They bind the objects near the origin, but release them if they move far out. The equations of motion for this problem follow from the Hamiltonian equations:

$$\frac{dp_x}{dt} = -x - 2xy \quad \frac{dp_y}{dt} = -y - x^2 + y^2 \quad \frac{dx}{dt} = p_x \quad \frac{dy}{dt} = p_y$$

- (a) Numerically solve for the position  $[x(t), y(t)]$  for a particle in the Hénon-Heiles potential.
- (b) Plot  $[x(t), y(t)]$  for a number of initial conditions. Check that the initial condition  $E < 1/6$  leads to a bounded orbit.
- (c) Produce a Poincaré section in the  $(y, p_y)$  plane by plotting  $(y, p_y)$  each time an orbit passes through  $x = 0$ .

## 20 Fractals

*It is common in nature to notice objects, called fractals, that do not have well-defined geometric shapes, but nevertheless appear regular and pleasing to the eye. These objects have dimensions that are fractions, and occur in plants, sea shells, polymers, thin films, colloids, and aerosols. We will not study the scientific theories that lead to fractal geometry, but rather will look at how some simple models and rules produce fractals. To the extent that these models generate structures that look like those in nature, it is reasonable to assume that the natural processes must be following similar rules arising from the basic physics or biology that creates the objects. As a case in point, if we look at the bifurcation plot of the logistics map, Fig. 18.2, we see a self-similarity of structure that is characteristic of fractals; in this case we know the structure arises from the equation  $x_{n+1} = \mu x_n(1 - x_n)$  used for the map. Detailed applications of fractals can be found in many literature sources [34, 41–43].*

### 20.1 Fractional Dimension (Math)

Benoit Mandelbrot, who first studied the fractional-dimension figures with the supercomputers at IBM Research, gave them the name *fractal* [44]. Some geometric objects, such as the Koch curves, are exact fractals with the same dimension for all their parts. Other objects, such as bifurcation curves, are statistical fractals in which elements of randomness occur and the dimension can be defined only locally or on the average.

Consider an abstract “object” such as the density of charge within an atom. There are an infinite number of ways to measure the “size” of this object, for example; each moment of the distribution provides a measure of the size, and there are an infinite number of moments. Likewise, when we deal with complicated objects that have fractional dimensions, there are different definitions of dimension, and each may give a somewhat different dimension. In addition, the fractal dimension is often defined by using a measuring box whose size approaches zero. In realistic applications there may be numerical or conceptual difficulties in approaching such a limit, and for this reason a precise value of the fractional dimension may not be possible.

Our first definition of fractional dimension  $d_f$  (or *Hausdorff-Besicovitch dimension*) is based on our knowledge that a line has dimension 1; a triangle, dimension 2, and a cube, dimension 3. It seems perfectly reasonable to ask if there is some mathematical formula, which agrees with our experience for regular objects, yet can also be used for determining fractional dimensions.

For simplicity, let us consider objects that have the same length  $L$  on each side, as do equilateral triangles and squares, and which have uniform density. We postulate that the dimension of an object is determined by the dependence of its total mass upon its length:

$$M(L) \propto L^{d_f} \quad (20.1)$$

where the power  $d_f$  is the *fractal dimension*. As you may verify, this rule works with the 1D, 2D, and 3D regular figures of our experience, so it is a reasonable hypothesis. When we apply (20.1) to fractal objects, we end up with fractional values for  $d_f$ . Actually, we will find it easier to determine the fractal dimension not from an object's mass, which is *extensive* (depends on size), but rather from its density, which is *intensive*. The density is defined as mass/length for a linear object, as mass/area for a planar object, and mass/volume for a solid object. That being the case, for a planar object we hypothesize that

$$\rho = \frac{M(L)}{\text{Area}} \propto \frac{L^{d_f}}{L^2} \propto L^{d_f - 2} \quad (20.2)$$

## 20.2

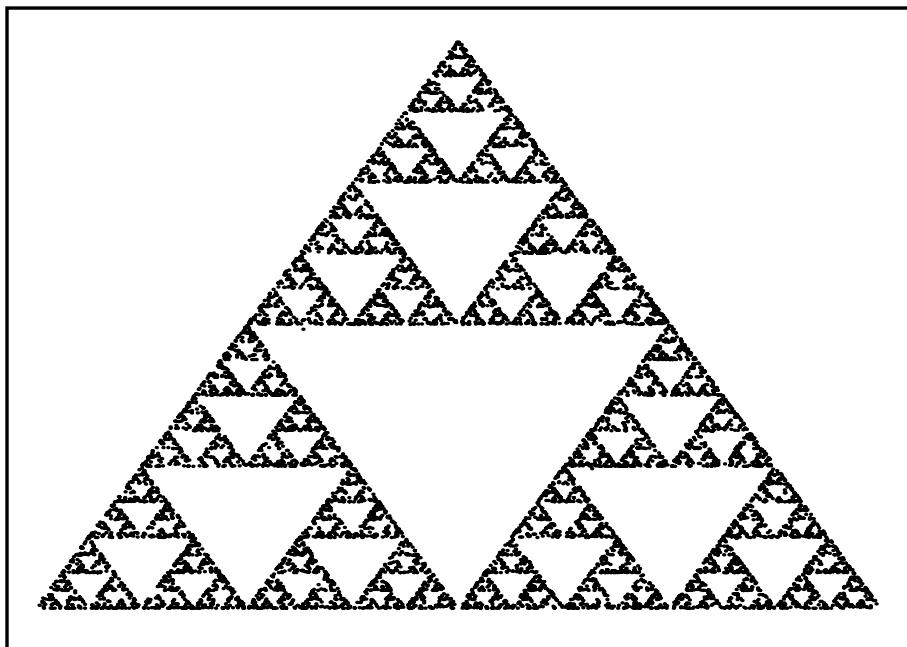
### The Sierpiński Gasket (Problem 1)

To generate our first fractal (Fig. 20.1), we play a game in which we pick points and place dots on them. Here are the rules (which you should try out now):

1. Draw an equilateral triangle with vertices and coordinates:

$$\text{vertex 1 : } (a_1, b_1) \quad \text{vertex 2 : } (a_2, b_2) \quad \text{vertex 3 : } (a_3, b_3)$$

2. Place a dot at an arbitrary point  $P = (x_0, y_0)$  within this triangle.
3. Find the next point by selecting randomly the integer 1, 2, or 3:
  - (a) If 1, place a dot halfway between  $P$  and vertex 1.
  - (b) If 2, place a dot halfway between  $P$  and vertex 2.
  - (c) If 3, place a dot halfway between  $P$  and vertex 3.
4. Repeat the process, using the last dot as the new  $P$ .



**Fig. 20.1** A Sierpiński gasket containing 15,000 points constructed as a statistical fractal. Each filled part of this figure is self-similar.

Mathematically, the coordinates of successive points are given by the formula

$$(x_{k+1}, y_{k+1}) = \frac{(x_k, y_k) + (a_n, b_n)}{2} \quad n = \text{integer } (1 + 3r_i) \quad (20.3)$$

where  $r_i$  is a random number between 0 and 1, and where the *Integer* function outputs the closest integer smaller than, or equal to, the argument. After 15,000 points, you should obtain a collection of dots like Fig. 20.1.

### 20.2.1

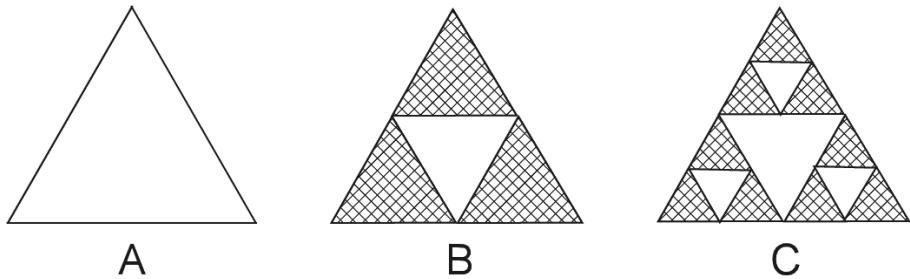
#### Sierpinsky Implementation

Write a program to produce a Sierpiński gasket. Determine empirically the fractal dimension of your figure. Assume that each dot has mass 1 and that  $\rho = CL^\alpha$ . (You can have the computer do the counting by defining an array *box* of all 0 values and then change a 0 to a 1 when a dot is placed there.)

### 20.2.2

#### Assessing Fractal Dimension

The topology of Fig. 20.1 was first analyzed by the Polish mathematician Sierpiński. Observe that there is the same structure in a small region as there is in



**Fig. 20.2** A Sierpinsky gasket constructed by successively connecting the midpoints of the sides of each equilateral triangle. A, B, and C show the first three steps in the process.

the entire figure. In other words, if the figure had infinite resolution, any part of the figure could be scaled up in size and will be similar to the whole. This property is called *self-similarity*.

We construct a regular form of the Sierpinsky gasket by removing an inverted equilateral triangle from the center of all filled equilateral triangles to create the next figure (Fig. 20.2). We then repeat the process *ad infinitum*, scaling up the triangles so each one has side  $r = 1$  after each step. To see what is unusual about this type of object, we look at how its density (mass/area) changes with size, and then apply (20.2) to determine its fractal dimension. Assume that each triangle has mass  $m$ , and assign unit density to the single triangle:

$$\rho(L = r) \propto \frac{M}{r^2} = \frac{m}{r^2} \stackrel{\text{def}}{=} \rho_0 \quad (\text{Fig. 20.2A}).$$

Next, for the equilateral triangle with side  $L = 2$ , the density

$$\rho(L = 2r) \propto \frac{(M = 3m)}{(2r)^2} = 34mr^2 = \frac{3}{4}\rho_0 \quad (\text{Fig. 20.2B}).$$

We see that the extra white space in Fig. 20.2B leads to a density that is  $\frac{3}{4}$  that of the previous stage. For the structure in Fig. 20.2C, we obtain

$$\rho(L = 4r) \propto \frac{(M = 9m)}{(4r)^2} = (34)^2 \frac{m}{r^2} = \left(\frac{3}{4}\right)^2 \rho_0 \quad (\text{Fig. 20.2C}).$$

We see that as we continue the construction process, the density of each new structure is  $\frac{3}{4}$  that of the previous one. This is unusual. Yet in (20.2) we have derived that

$$\rho \propto CL^{d_f - 2} \quad (20.4)$$

Equation (20.2) implies that a plot of the logarithm of the density  $\rho$  versus the logarithm of the length  $L$  for successive structures, yields a straight line of

slope:

$$d_f - 2 = \frac{\Delta \log \rho}{\Delta \log L} \quad (20.5)$$

As applied to our problem:

$$d_f = 2 + \frac{\Delta \log \rho(L)}{\Delta \log L} = 2 + \frac{\log 1 - \log(3/4)}{\log 1 - \log 2} \simeq 1.58496 \quad (20.6)$$

As is evident in Fig. 20.2, as the gasket gets larger and larger (and consequently more massive), it contains more and more open space. So even though its mass approaches infinity as  $L \rightarrow \infty$ , its density approaches zero! And since a two-dimensional figure like a solid triangle has a constant density as its length increases, a 2D figure would have a slope equal to 0. Since the Sierpiński gasket has a slope  $d_f - 2 \simeq -0.41504$ , it fills space to a lesser extent than a 2D object, but more than does a 1D object; it is a fractal.

## 20.3

### **Beautiful Plants (Problem 2)**

It seems paradoxical that natural processes subject to chance can produce objects of high regularity and symmetry. For example, it is hard to believe that something as beautiful and symmetric as a fern (Fig. 20.3) has random elements in it. Nonetheless, there is a clue here in that much of the fern's beauty arises from the similarity of each part to the whole (self-similarity), with different ferns similar, but not identical, to each other. These are characteristics of fractals. Your **problem** is to discover if a simple algorithm including some randomness can draw regular ferns. If the algorithm produces objects that resemble ferns, then presumably you have uncovered mathematics similar to that responsible for the shape of ferns.

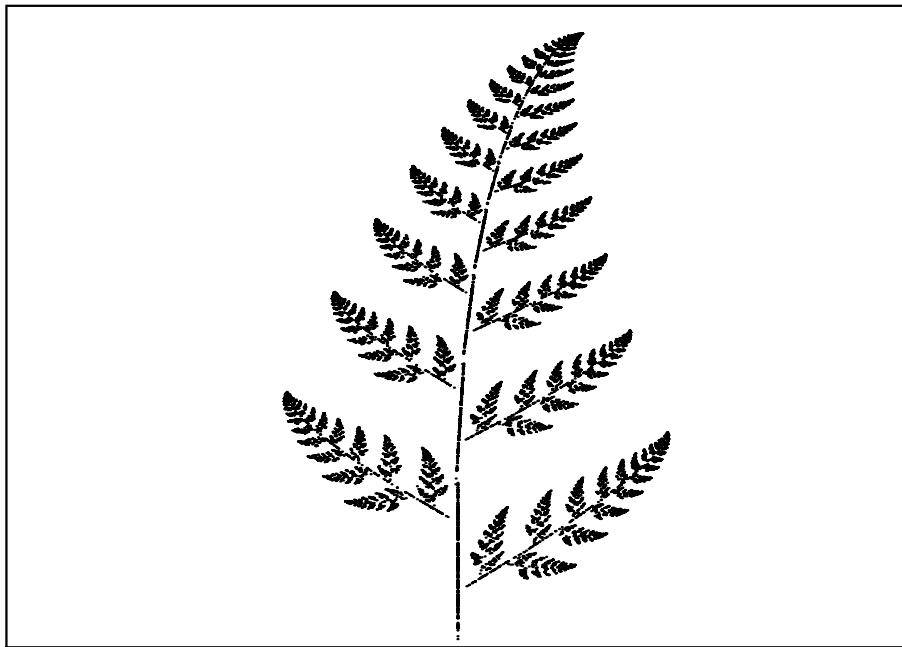
#### 20.3.1

##### **Self-Affine Connection (Theory)**

In (20.3), which defines mathematically how a Sierpiński gasket is constructed, a *scaling factor* of  $\frac{1}{2}$  is part of the relation of one point to the next. A more general transformation of a point  $P = (x, y)$  into another point  $P' = (x', y')$  via *scaling* is

$$(x', y') = s(x, y) = (sx, sy) \quad (\text{scaling}) \quad (20.7)$$

If the scale factor  $s > 0$ , an amplification occurs, whereas if  $s < 0$ , a reduction occurs. In our definition (20.3) of the Sierpiński gasket, we also added in a



**Fig. 20.3** A fern after 30,000 iterations of the algorithm (20.10). If you enlarge this, you will see that each frond has similar structure.

constant  $a_n$ . This is a *translation operation*, which has the general form

$$(x', y') = (x, y) + (a_x, a_y) \quad (\text{translation}) \quad (20.8)$$

Another operation, not used in the Sierpiński gasket, is a *rotation* by angle  $\theta$ :

$$x' = x \cos \theta - y \sin \theta \quad y' = x \sin \theta + y \cos \theta \quad (\text{rotation}) \quad (20.9)$$

The entire set of transformations, scalings, rotations, and translations, define an *affine transformation* (“affine” denotes a close relation between successive points). The transformation is still considered affine even if it is a more general linear transformation with the coefficients not all related to one  $\theta$  (in that case, we can have contractions and reflections). What is important is that the object created with these rules turn out to be self-similar; each step leads to new parts of the object that bear the same relation to the ancestor parts as did the ancestors to theirs. This is what makes the object look similar at all scales.

### 20.3.2

#### Barnsley's Fern Implementation (fern.c)

We obtain a Barnsley's Fern by extending the dots game to one in which new points are selected using an affine connection with some elements of chance

mixed in

$$(x, y)_{n+1} = \begin{cases} (0.5, 0.27y_n) & \text{with 2\% probability} \\ (-0.139x_n + 0.263y_n + 0.57 \\ 0.246x_n + 0.224y_n - 0.036) & \text{with 15\% probability} \\ (0.17x_n - 0.215y_n + 0.408 \\ 0.222x_n + 0.176y_n + 0.0893) & \text{with 13\% probability} \\ (0.781x_n + 0.034y_n + 0.1075 \\ -0.032x_n + 0.739y_n + 0.27) & \text{with 70\% probability} \end{cases} \quad (20.10)$$

To select a transformation with probability  $\mathcal{P}$ , we select a uniform random number  $r$  in the interval  $[0, 1]$  and perform the transformation if  $r$  is in a range proportional to  $\mathcal{P}$ :

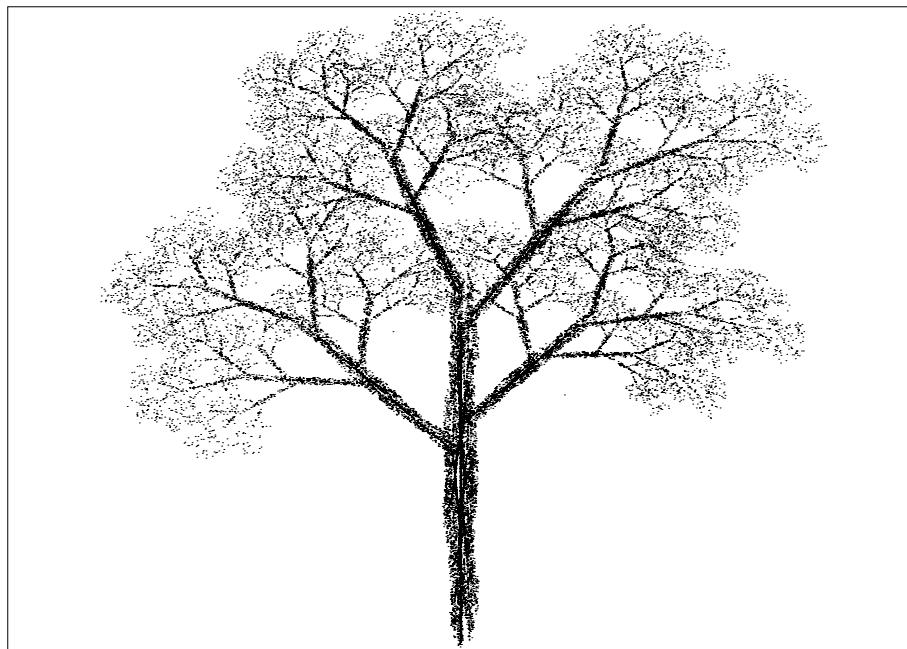
$$\mathcal{P} = \begin{cases} 2\% & r < 0.02 \\ 15\% & 0.02 \leq r \leq 0.17 \\ 13\% & 0.17 < r \leq 0.3 \\ 70\% & 0.3 < r < 1 \end{cases} \quad (20.11)$$

The rules (20.10) and (20.11) can be combined into one:

$$(x, y)_{n+1} = \begin{cases} (0.5, 0.27y_n) & r < 0.02 \\ (-0.139x_n + 0.263y_n + 0.57 \\ 0.246x_n + 0.224y_n - 0.036) & 0.02 \leq r \leq 0.17 \\ (0.17x_n - 0.215y_n + 0.408 \\ 0.222x_n + 0.176y_n + 0.0893) & 0.17 < r \leq 0.3 \\ (0.781x_n + 0.034y_n + 0.1075 \\ -0.032x_n + 0.739y_n + 0.27) & 0.3 < r < 1 \end{cases} \quad (20.12)$$

Although (20.10) makes the basic idea clearer, (20.12) is easier to program.

The starting point in Barnsley's fern (Fig. 20.3) is  $(x_1, y_1) = (0.5, 0.0)$ , and the points are generated by repeated iterations. An important property of this fern is that it is not completely self-similar, as you can see by noting how different are the stems and the fronds. Nevertheless, the stem can be viewed as a compressed copy of a frond, and the fractal obtained with (20.10) is still *self-affine*, yet with a dimension that varies in different parts of the figure.



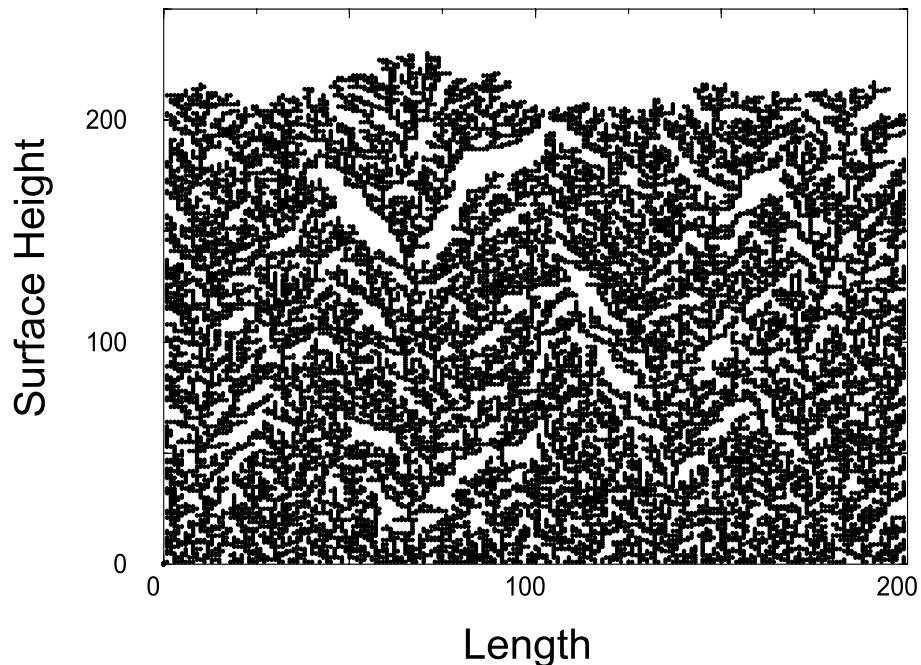
**Fig. 20.4** A fractal tree created with the simple algorithm (20.13).

### 20.3.3

#### **Self-Affinity in Trees Implementation (tree.c)**

Now that you know how to grow ferns, look around and notice the regularity in trees (such as Fig. 20.4). Can it be that this also arises from a self-affine structure? Write a program, similar to the one for the fern, starting at  $(x_1, y_1) = (0.5, 0.0)$  and iterating the following self-affine transformation:

$$(x_{n+1}, y_{n+1}) = \begin{cases} (0.05x_n, 0.6y_n) & 10\% \text{ probability} \\ (0.05x_n, -0.5y_n + 1.0) & 10\% \text{ probability} \\ (0.46x_n - 0.15y_n, 0.39x_n + 0.38y_n + 0.6) & 20\% \text{ probability} \\ (0.47x_n - 0.15y_n, 0.17x_n + 0.42y_n + 1.1) & 20\% \text{ probability} \\ (0.43x_n + 0.28y_n, -0.25x_n + 0.45y_n + 1.0) & 20\% \text{ probability} \\ (0.42x_n + 0.26y_n, -0.35x_n + 0.31y_n + 0.7) & 20\% \text{ probability} \end{cases} \quad (20.13)$$



**Fig. 20.5** A simulation of the ballistic deposition of 20,000 particles on a substrate of length 200. The vertical height increases with the length of deposition time so that the top is the final surface.

## 20.4

### Ballistic Deposition (Problem 3)

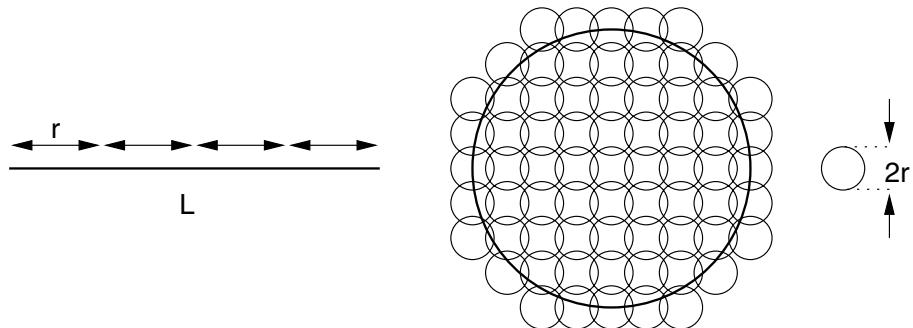
There are a number of physical and manufacturing processes in which particles are deposited on a surface and form a film. Because the particles are evaporated from a hot filament, there is randomness in the emission process, yet the produced films turn out to have well-defined, regular structures. Again we suspect fractals. Your **problem** is to develop a model that simulates this growth process, and compare your produced structures to those observed.

#### 20.4.1

##### Random Deposition Algorithm (film.c)

The idea of simulating random depositions began with [45], who used tables of random numbers to simulate the sedimentation of moist spheres in hydrocarbons. We shall examine a method of simulation [46] which results in the deposition shown in Fig. 20.5.

Consider particles falling onto and sticking to a horizontal line of length  $L$  composed of 200 deposition sites. All particles start from the same height, but to simulate their different velocities, we assume they start at random distances



**Fig. 20.6** Examples of the use of box counting to determine fractal dimension. On the left the perimeter is being covered, while on the right the entire figure is being covered.

from the left side of the line. The simulation consists of generating uniform random sites between 0 and  $L$ , and having the particle stick to the site on which it lands. Because a realistic situation may have columns of aggregates of different heights, the particle may be stopped before it makes it to the line, or may bounce around until it falls into a hole. We therefore assume that if the column height at which the particle lands is greater than that of both its neighbors, it will add to that height. If the particle lands in a hole, or if there is an adjacent hole, it will fill up the hole. We speed up the simulation by setting the height of the hole equal to the maximum of its neighbors:

1. Choose a random site  $r$ .
2. Let the array  $h_r$  be the height of the column at site  $r$ .
3. Make the decision:

$$h_r = \begin{cases} h_r + 1 & \text{if } h_r \geq h_{r-1} \text{ and } h_r > h_{r+1} \\ \max[h_{r-1}, h_{r+1}] & \text{if } h_r < h_{r-1} \text{ and } h_r < h_{r+1} \end{cases} \quad (20.14)$$

The results of this type of simulation show several empty regions scattered throughout the line (Fig. 20.5). This is an indication of the statistical nature of the process while the film is growing. Simulations by Fereydon reproduced the experimental observation that the average height increases linearly with time, and produced fractal surfaces. (You will be asked to determine the fractal dimension of a similar surface as an exercise.)

## 20.5

### Length of British Coastline (Problem 4)

In 1967 Mandelbrot [47] asked a classic question “How long is the Coast of Britain?” If Britain had the shape of Colorado or Wyoming, both of which have straight line boundaries, its perimeter would be a curve of dimension one with a finite length. However, coast lines are geographic not geometric curves, with each portion of the coast often statistically self-similar to the entire coast, yet at a reduced scale. In these latter cases, the perimeter is a fractal, and the length is either infinite or meaningless. Mandelbrot deduced the dimension of the west coast of Great Britain to be  $d_f = 1.25$ . In your **problem** we ask you to determine the dimension of the perimeter one of our fractal simulations.

#### 20.5.1

##### Coastline as Fractal (Model)

The length of the coastline of an island is the perimeter of that island. While the concept of perimeter is clear for regular geometric figures, some thought is required to give the concept meaning for an object that may be infinitely self-similar. Let us assume that a map maker has a ruler of length  $r$ . If he walks along the coastline and counts the number of times  $N$  that he must place the ruler down in order to *cover* the coastline, he will obtain a value for the length  $L$  of the coast as  $Nr$ . Imagine now that the map maker keeps repeating his walk with smaller and smaller rulers. If the coast were a geometric figure, or a *rectifiable curve*, at some point the length  $L$  would become essentially independent of  $r$  and it would approach a constant. Nonetheless, as discovered empirically by Richardson [48] for natural coast lines, such as that of South Africa and Britain, the perimeter appears to be the function of  $r$

$$L(r) = Mr^{1-d_f} \quad (20.15)$$

where  $M$  and  $d_f$  are empirical constants. For a geometric figure, or for Colorado,  $d_f = 1$ , and the length approaches a constant as  $r \rightarrow 0$ . Yet for a fractal with  $d_f > 1$ , the perimeter  $L \rightarrow \infty$  as  $r \rightarrow 0$ . This means that as a consequence of self-similarity fractals may be of finite size, but have infinite perimeter. However, at some point there may be no more details to discern as  $r \rightarrow 0$  (say at the quantum), and so the limit may not be meaningful.

Our sample simulation is `Fractals/Film.java` on the CD (`FilmDim.java` on the Instructor's CD). It contains the essential loop:

```
int spot = random.nextInt(200);
if (spot == 0) {
    if (coast[spot] < coast[spot+1]) coast[spot] = coast[spot+1];
    else coast[spot]++;
}
else if (spot == coast.length - 1) {
    if (coast[spot] < coast[spot-1]) coast[spot] = coast[spot-1];
    else coast[spot]++;
}
else if (coast[spot]<coast[spot-1] && coast[spot]<coast[spot+1] ) {
    if (coast[spot-1] > coast[spot+1] ) coast[spot] = coast[spot-1];
    else coast[spot] = coast[spot+1];
}
else coast[spot]++;
}
```

The results of this type of simulation show several empty regions scattered throughout the line (Fig. 20.5). This is an indication of the statistical nature of the process while the film is growing. Simulations by Fereydoon reproduced the experimental observation that the average height increases linearly with time, and produced fractal surfaces. (You will be asked to determine the fractal dimension of a similar surface as an exercise.)

### 20.5.2

#### Box Counting Algorithm

Consider a line of length  $L$  broken up into segments of length  $r$  (Fig. 20.6 left). The number of segments or “boxes” needed to cover the line is related to the size  $r$  of the box by

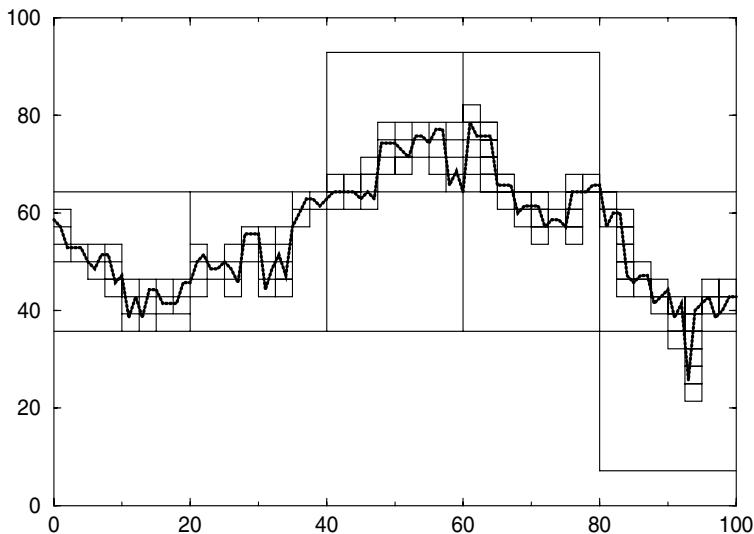
$$N(r) = \frac{L}{r} \propto \frac{1}{r} \quad (20.16)$$

A possible definition of fractional dimension is the power of  $r$  in this expression as  $r \rightarrow 0$ . In our example, it tells us that the line has dimension  $d_f = 1$ . If we now ask how many little circles of radius  $r$  it takes to *cover* or fill a circle of area  $A$  (Fig. 20.6, right), we would find

$$N(r) = \lim_{r \rightarrow 0} \frac{A}{\pi r^2} \quad \Rightarrow \quad d_f = 2 \quad (20.17)$$

as expected. Likewise, counting the number of little spheres or cubes that can be packed within a large sphere tells us that a sphere has dimension  $d_f = 3$ . In general, if it takes  $N$  “little” spheres or cubes of side  $r \rightarrow 0$  to cover some object, then the fractal dimension  $d_f$  can be deduced as

$$N(r) \propto \left(\frac{1}{r}\right)^{d_f} \propto s^{d_f} \quad (\text{as } r \rightarrow 0) \quad d_f = \lim_{r \rightarrow 0} \frac{\log N(r)}{\log(1/r)} \quad (20.18)$$



**Fig. 20.7** Example of the use of box counting to determine the fractal dimension of the perimeter along a surface. Observe how the “coastline” is being covered by boxes of two different sizes (scales).

Here  $s \propto 1/r$  is called the *scale* in geography, so  $r \rightarrow 0$  corresponds to infinite scale. To illustrate, you may be familiar with the low scale on a map being 10,000 m to the cm, while the high scale is 100 m to the cm. If we want the map to show small details (sizes), we want a map of higher scale.

We will use box counting to determine the dimension of a perimeter, not of an entire figure. Once we have a value for  $d_f$  we can determine a value for the length of the perimeter via (20.15).

### 20.5.3

#### Coastline Implementation

Rather than ruin our eyes with a geographic map, we use a mathematical one. Specifically, with a little imagination you will see that the top portion of the graph in Fig. 20.5 looks like a natural coastline. Determine  $d_f$  by covering this figure, or one you have generated, with a semitransparent piece of graph paper.<sup>1</sup>, and counting the number of boxes containing any part of the coastline (Fig. 20.7)

1. Print out your coastline graph with the same physical scale for the vertical and horizontal axes.

<sup>1</sup> Yes, we are suggesting a painfully analog technique based on the theory that trauma leaves a lasting impression. If you prefer, you can store your output as a matrix of 1 and 0 values and let the computer do the counting, but this will take more of your time!

2. The vertical height in our printout was 17 cm. This sets the scale of the graph as 1:17, or  $s = 17$ .
3. The largest boxes on our graph paper were  $1 \text{ cm} \times 1 \text{ cm}$ . We found that the coastline passed through  $N = 24$  of these large boxes (i.e., 24 large boxes covered the coastline at  $s = 17$ ).
4. The next smaller boxes on our graph paper were  $0.5 \text{ cm} \times 0.5 \text{ cm}$ . We found that 51 smaller boxes covered the coastline at a scale of  $s = 34$ .
5. The smallest boxes on our graph paper were  $1 \text{ mm} \times 1 \text{ mm}$ . We found that 406 smallest boxes covered the coastline at a scale of  $s = 170$ .
6. Equation (20.18) tells us that as the box sizes get progressively smaller, we have

$$\log N \simeq \log A + d_f \log s$$

$$\Rightarrow d_f \simeq \frac{\Delta \log N}{\Delta \log s} = \frac{\log N_2 - \log N_1}{\log s_2 - \log s_1} = \frac{\log(N_2/N_1)}{\log(s_2/s_1)}$$

Clearly, only the relative scales matter because the proportionality constants cancel out in the ratio. A plot of  $\log N$  versus  $\log s$ , should yield a straight line (the third point verifies that it is a line). In our example we found a slope  $d_f = 1.23$ .

As given by (20.15), the perimeter of the coastline

$$L \propto s^{1.23-1} = s^{0.23} \quad (20.19)$$

If we keep making the boxes smaller and smaller so that we are looking at the coastline at higher and higher scale, *and* if the coastline is a fractal with self-similarity at all levels, then the scale  $s$  keeps getting larger and larger with no limits (or at least until we get down to some quantum limits). This means

$$L \propto \lim_{s \rightarrow \infty} s^{0.23} = \infty \quad (20.20)$$

We conclude that, in spite of being only a small island, to a mathematician the coastline of Britain is, indeed, infinite.

## 20.6

### Problem 5: Correlated Growth, Forests, and Films

It is an empirical fact that in nature there is an increased likelihood for a plant to grow if there is another one nearby (Fig. 20.8). This *correlation* is also valid



**Fig. 20.8** A scene as might be seen in the undergrowth of a forest or in a correlated ballistic deposition.

for our “growing” of surface films, as in the previous algorithm. Your **problem** is to include correlations in the surface simulation.

### 20.6.1

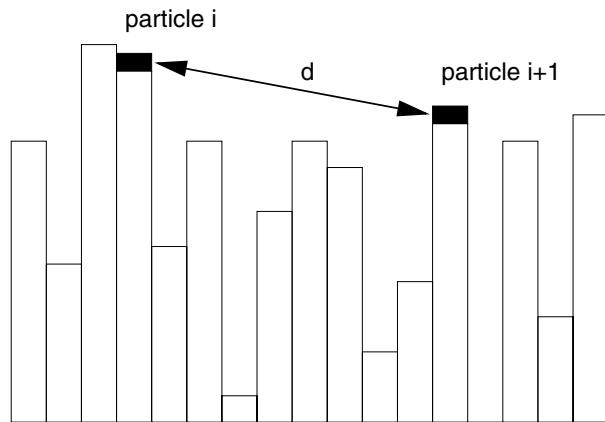
#### Correlated Ballistic Deposition Algorithm (column.c)

A variation of the ballistic deposition algorithm, known as *correlated ballistic deposition*, simulates mineral deposition onto substrates in which dendrites form [49]. We extend the previous algorithm to include the likelihood that a freshly deposited particle will attract another particle. We assume that the probability of sticking  $\mathcal{P}$  depends on the distance  $d$  that the added particle is from the last one (Fig. 20.9):

$$\mathcal{P} = c d^\eta \quad (20.21)$$

Here  $\eta$  is a parameter and  $c$  is a constant that sets the probability scale.<sup>2</sup> For our implementation we choose  $\eta = -2$ , which means that there is an inverse square attraction between the particles (less probable as they get farther apart).

<sup>2</sup> The absolute probability, of course, must be less than one, but it is nice to choose  $c$  so that the relative probabilities produce a graph with easily seen variations.



**Fig. 20.9** The probability of particle  $i + 1$  sticking in some column depends on the distance  $d$  from the previously deposited particle  $i$ .

As in our study of uncorrelated deposition, a uniform random number in the interval  $[0, L]$  determines the column in which the particle is deposited. We use the same rules about the heights as before, but now a second random number is used in conjunction with (20.21) to decide if the sticks. For instance, if the computed probability is 0.6 and if  $r < 0.6$ , the particle is accepted (sticks); if  $r > 0.6$ , the particle is rejected.

#### 20.6.2

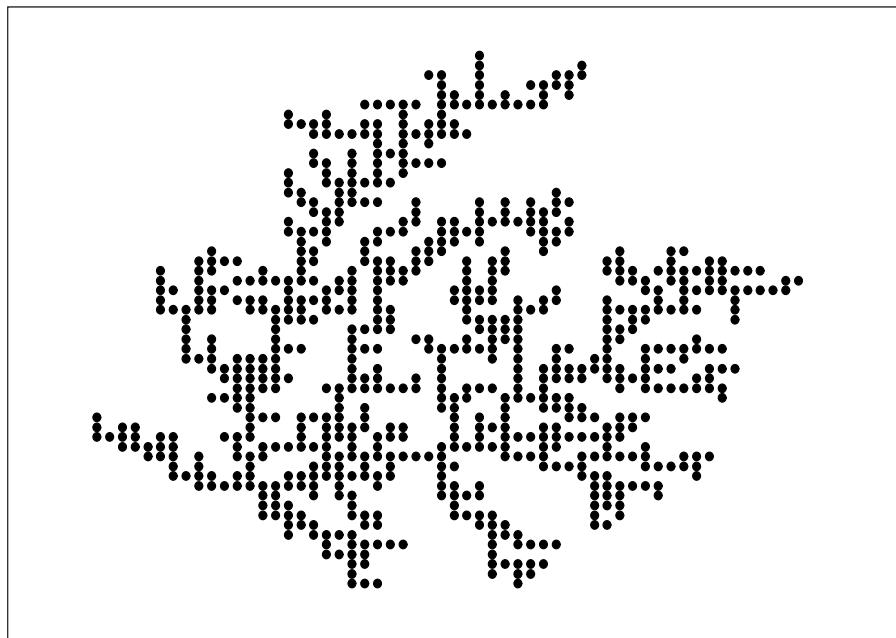
##### Globular Cluster (Problem)

Consider a bunch of grapes on an overhead vine. Your **problem** is to determine how its tantalizing shape arises. As a hint, you are told that these shapes, as well as others such as dendrites, colloids, and thin-film structure, appear to arise from an aggregation process that is limited by diffusion.

#### 20.6.3

##### Diffusion-Limited Aggregation Algorithm (dla.c)

A model of diffusion-limited aggregation (DLA) has successfully explained the relation between a cluster's perimeter and mass [50]. We start with a 2D lattice containing a seed particle in the middle. We draw a circle around the particle and place another particle on the circumference of the circle at some random angle. We then release the second particle and have it execute a random walk, much like the one we studied in Chap. 11 but restricted to vertical or horizontal jumps between lattice sites. This is a type of *Brownian motion* that simulates the diffusion process. To make the model more realistic, we let the length of each step vary according to a random Gaussian distribution. If at



**Fig. 20.10** *Left:* A globular cluster of particles of the type that might occur in a colloid. There is a seed at the center, and random walkers are started at random points around a circle, with not all reaching the center.

some point during its random walk, the particle finds another particle within one lattice spacing, they stick together and the walk terminates. If the particle passes outside the circle from which it was released, it is lost forever. The process is repeated as often as desired and results in clusters (Fig. 20.10).

1. Write a subroutine that generates random numbers with a Gaussian distribution.<sup>3</sup>
2. Define a 2D lattice of points represented by the array `grid[400][400]` with all elements initially zero.
3. Place the seed at the center of the lattice, that is, set `grid[199][199]=1`.
4. Imagine a circle of radius 180 lattice constants centered at `grid[199][199]`. This is the circle from which we release particles.
5. Determine the angular position of the new particle on the circle's circumference by generating a uniform random angle between 0 and  $2\pi$ .
6. Compute the  $x$  and  $y$  positions of the new particle on the circle.

<sup>3</sup> We indicated how to do this in Section 11.7.3.

7. Determine whether the particle moves horizontally or vertically by generating a uniform random number  $0 < r_{xy} < 1$  and applying the rule

$$\text{if } r_{xy} \begin{cases} < 0.5 & \text{motion is vertical} \\ > 0.5 & \text{motion is horizontal} \end{cases} \quad (20.22)$$

8. Generate a Gaussian-weighted random number in the interval  $[-\infty, \infty]$ . This is the size of the step, with the sign indicating direction.
9. We now know the total distance and direction in which the particle will move. It “jumps” one lattice spacing at a time until this total distance is covered.
10. Before a jump, check whether a nearest-neighbor site is occupied:
- If occupied, stay at present position and the walk is over.
  - If unoccupied, the particle jumps one lattice spacing.
  - Continue the checking and jumping until the total distance is covered, until the particle sticks, or until it leaves the circle.
11. Once one random walk is over, another particle can be released and the process repeated. This is how the cluster grows.

Because many particles get “lost,” you may need to generate hundreds of thousands of particles to form a cluster of several hundred particles.

#### 20.6.4

##### **Fractal Analysis of DLA Graph (Assessment)**

A cluster generated with the DLA technique is shown in Fig. 20.10. We wish to analyze it to see if the structure is a fractal, and, if so, to determine its dimension. The analysis is a variation of the one used to determine the length of the coastline of Britain.

1. Draw a square of length  $L$ , small relative to the size of the cluster, around the seed particle. (“Small” might be seven lattice spacings to a side.)
2. Count the number of particles within the square.
3. Compute the density  $\rho$  by dividing the number of particles by the number of sites available in the box (49 in our example).
4. Repeat the procedure using larger and larger squares.
5. Stop when the cluster is covered.

6. The (box-counting) fractal dimension  $d_f$  is estimated from a log–log plot of the density  $\rho$  versus  $L$ . If the cluster is a fractal, then (20.2) tells us that  $\rho \propto L^{d_f - 2}$ , and the graph should be a straight line of slope  $d_f - 2$ .

The graph we generated had a slope of  $-0.36$ , which corresponds to a fractal dimension of  $1.66$ . Because random numbers are involved, the graph you generate will be different, but the fractal dimension should be similar. (Actually, the structure is multifractal, and so the dimension varies with position.)

## 20.7

### Problem 7: Fractal Structures in Bifurcation Graph

Recall the project on the logistics map where we plotted the values of the stable population numbers versus growth parameter  $\mu$ . Take one of the bifurcation graphs you produced and determine the fractal dimension of different parts of the graph by using the same technique that was applied to the coastline of Britain.

**21****Parallel Computing**

*In this chapter, we examine parallel computers and one method used to program them. This subject continues to change rapidly, and, accordingly, the meaning of various terms, as well as the way to program them also changes. Our view is influenced by the overriding aim of this book, namely, how to use the computer to assist with science. By the same token, we view much of the present day developments in parallel computers as being of limited usefulness because it is so hard to implement general applications on these machines. In contrast, the computer manufacturers view their creations as incredibly powerful machines, as indeed they are for some problems.*

*This chapter contains a high-level survey of parallel computing, while Chap. 22 contains a detailed tutorial on the use of the Message Passing Interface (MPI) package. An earlier package, Parallel Virtual Machine (PVM), is still in use, but not as popular as MPI. If you have not already done so, we recommend that you read Chap. 13 before you proceed with this chapter. Our survey is brief and given from a practitioner's point of view. The text [51] surveys parallel computing and MPI from a computer science point of view, and provides some balance to this chapter. References on MPI include Web resources [52–54] and the texts [51, 55, 56]. References on parallel computing include [51, 57–60].*

**Problem:** Start with the program you wrote to generate the bifurcation plot for bug dynamics in Chap. 19 and modify it so that different ranges for the growth parameter  $\mu$  are computed simultaneously on multiple CPUs. Although this small problem is not worth investing your time to obtain a shorter turnaround time, it is worth investing your time to gain some experience in parallel computing. In general, parallel computing holds the promise of permitting you to get faster results, to solve bigger problems, to run simulations at finer resolutions, or to model physical phenomena more realistically; but it takes some work to accomplish this.

## 21.1

### Parallel Semantics (Theory)

We have seen in Chap. 13 that many of the tasks undertaken by a high-performance computer are run in parallel by making use of internal structures such as pipelined and segmented CPUs, hierarchical memory, and separate I/O processors. While these tasks are run “in parallel,” the modern use of *parallel computing* or *parallelism* denotes applying multiple processors to a single problem [51]. It is a computing environment in which some numbers of CPUs are running asynchronously and communicating with each other in order to exchange intermediate results and coordinate their activities.

For instance, consider matrix multiplication in terms of its elements

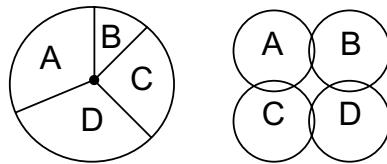
$$[B] = [A][B] \quad \Rightarrow \quad B_{i,j} = \sum_{k=1}^N A_{i,k} B_{k,j} \quad (21.1)$$

Because the computation of a  $B_{i,j}$  for particular values of  $i$  and  $j$  is independent of the computation for all other values, each  $B_{i,j}$  could be computed in parallel, or each row or column of  $[B]$  can be computed in parallel. However, because the  $B_{k,j}$  on the RHS of (21.1) must be the “old” values that existed before the matrix multiplication, some communication among the parallel processors is required to ensure that they do not store the “new” values of  $B_{k,j}$  before all of the multiplications are complete. This  $[B] = [A][B]$  multiplication is an example of *data dependency*, in which the data elements used in the computation depend on the order in which they are used. In contrast, the matrix multiplication  $[C] = [A][B]$  would be a *data parallel* operation, in which the data can be used in any order. So already we see the importance of communication, synchronization, and understanding of the mathematics behind an algorithm for parallel computation.

The processors in a parallel computer are placed at the *nodes* of a communication network. Each node may contain one, or a small number of CPUs, and the communication network may be internal to, or external to the computer. One way of categorizing parallel computers is by the approach they employ to handle instructions and data. From this viewpoint there are three types of machines.

- **Single instruction, single data (SISD):** These are the classic (VonNeumann) serial computers executing a single instruction on a single data stream before the next instruction and next data stream are encountered.
- **Single instruction, multiple data (SIMD):** Here instructions are processed from a single stream, but the instructions act concurrently on multiple data elements. Generally the nodes are simple and relatively slow, but are large in number.

- **Multiple instructions, multiple data (MIMD):** In this category each processor runs independently of the others with independent instructions and data. These are the type of machines that employ *message passing* packages, such as MPI, to communicate among processors. They may be a collection of workstations linked via a network, or more integrated machines with thousands of processors on internal boards, such as the Blue Gene computer. These computers, which do not have a shared memory space, are also called *multicomputers*. Although these types of computers are some of the most difficult to program, their low cost and effectiveness for certain classes of problems have led to their being the dominant type of parallel computer at present.



**Fig. 21.1** *Left:* Multitasking of four programs in memory at one time. On a SISD computer the programs are executed in “round-robin” order. *Right:* Four programs in the four separate memories of a MIMD computer.

The running of independent programs on a parallel computer is similar to the multitasking feature used by Unix and PCs. In multitasking, symbolized (Fig. 21.1, left), several independent programs reside in the computer’s memory simultaneously and share the processing time in a round-robin or priority order. On a SISD computer, only one program is running at a single time, but if other programs are in memory, then it does not take long to switch to them. In multiprocessing, these jobs may all be running at the same time, either in different parts of memory or on the memory of different computers (Fig. 21.1, right). Clearly, multiprocessing gets complicated if separate processors are operating on different parts of the *same* program, because then synchronization and load balance (keeping all processors equally busy) are concerns.

### 21.1.1 Granularity

In addition to instruction and data streams, another way to categorize parallel computation is by *granularity*. A *grain* is defined as a measure of the computational work to be done, more specifically, the ratio of computation work to communication work.

- **Coarse-grain parallel:** Separate programs running on separate computer systems with the systems coupled via a conventional communication network. To illustrate, six Linux PCs sharing the same files across a network but with a different central memory system for each PC. Each computer could

be operating on a different and independent part of one problem at the same time.

- **Medium-grain parallel:** Several processors executing (possibly different) programs simultaneously, while accessing a common memory. The processors are usually placed on a common *bus* (communication channel) and communicate with each other through the memory system. Medium-grain programs have different, independent, *parallel subroutines* running on different processors. Because the compilers are seldom smart enough to figure out which parts of the program to run where, the user must include the multitasking routines into the program.<sup>1</sup>
- **Fine-grain parallel:** As the granularity decreases and the number of nodes increases, there is an increased requirement for fast communication among the nodes. For this reason fine-grain systems tend to be custom-designed machines. The communication may be via a central bus or via shared memory for a small number of nodes, or through some form of high-speed network for massively parallel machines. In this latter case, the compiler divides the work among the processing nodes. For example, different `for` loops of a program may be run on different nodes.

## 21.2

### Distributed Memory Programming

An approach to concurrent processing that, because it is built from commodity PCs, has gained dominance acceptance for coarse- and medium-grain systems is *distributed memory*. In it, each processor has its own memory and the processors exchange data among themselves through a high-speed switch and network. The data exchanged or *passed* among processors have encoded *To* and *From* addresses and are called *messages*. The *clusters* of PCs or workstations that constitute a *Beowulf*<sup>2</sup> are examples of distributed memory computers. The unifying characteristic of a cluster is the integration of highly replicated compute and communication components into a single system, with each node still able to operate independently. For a Beowulf cluster, the components are commodity ones designed for a general market, as is the communication

<sup>1</sup> Some experts define our medium as coarse, yet this fine point changes with time.

<sup>2</sup> Presumably there is an analogy between the heroic exploits of the son of Ecgtheow and the nephew of Hygelac in the 1000 C.E. poem *Beowulf*, and the adventures of us common folk assembling parallel comput-

ers from common elements that surpassed the performance of major corporations and their proprietary, multimillion dollar supercomputers.

network and its high-speed switch (special interconnects are used by major manufacturers such as SGI and Cray, but they do not come cheaply). Note, a group of computers connected by a network may also be called a “cluster,” but unless they are designed for parallel processing, with the same type of processor used repeatedly, and with only a limited number of processors (the *front end*) onto which users may log in, they would not usually be called a Beowulf.

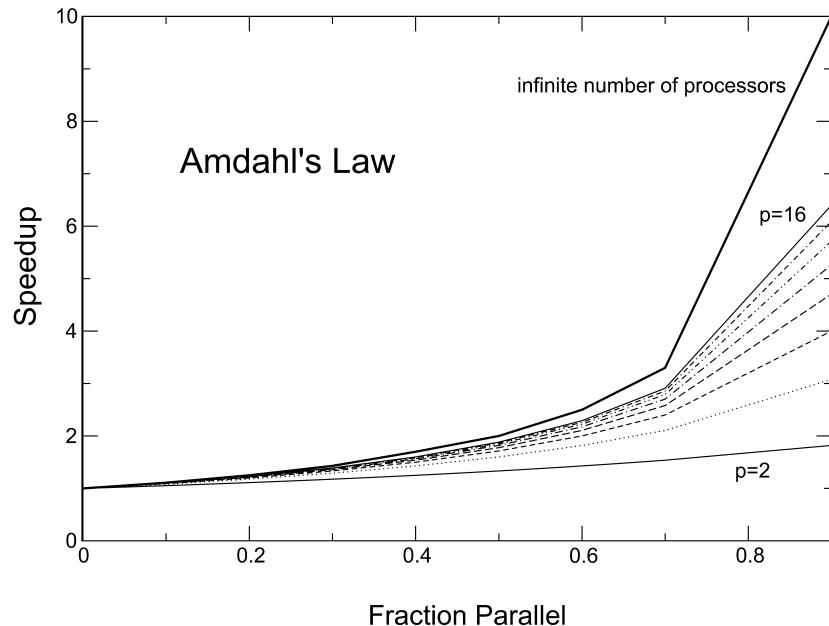
The literature contains frequent arguments concerning the differences between clusters, commodity clusters, Beowulfs, constellations, massively parallel systems, and so forth [59, 61]. Even though we recognize that there are major differences between the clusters on the *Top500* list of computers, and the ones which a university researcher may set up in his or her lab, we will not distinguish these fine points in the introductory materials we present here.

For a messages-passing program to be successful, the data must be divided among nodes so that, at least for a while, each node has all the data it needs to run an independent subtask. When a program begins execution, data are sent to all nodes. When all nodes have completed their subtasks, they exchange data again in order for each node to have a complete new set of data to perform the next subtask. This repeated cycle of data exchange followed by processing continues until the full task is completed. Message-passing MIMD programs are also *single-program multiple-data* programs, which means that the programmer writes a single program that is executed on all of the nodes. Often a separate host program, which starts the programs on the nodes, reads the input files and organizes the output.

### 21.3 Parallel Performance

Imagine a cafeteria line in which all the servers appear to be working hard and fast, yet the ketchup dispenser has a seed partially blocking its output, and so everyone in line must wait for the ketchup lovers up front to ruin their food before moving on. This is an example of the slowest step in a complex process determining the overall rate. An analogous situation holds for parallel processing, where the “seed” may be the issuing and communicating of instructions. Because the computation cannot advance until all instructions have been received, this one step may slow down or stop the entire process.

As we will soon demonstrate that the speedup of a program will not be significant unless you can get ~90% of it to run in parallel. An infinite number of processors running a half serial program can, at best, attain a speedup of 2. This means that you need to have a computationally intense problem to make parallelization worthwhile, and that much of the speedup will probably be obtained with only a small number of processors. This is one of the



**Fig. 21.2** The theoretical speedup of a program as a function of the fraction of the program that potentially may be run in parallel. The different curves correspond to different numbers of processors.

reasons why some proponents of parallel computers with thousands of processors suggest that you not apply the new machines to the old problems, but rather look for new problems which are both big enough and well suited for massively parallel processing to make the effort worthwhile.

The equation describing the effect on speedup of the balance between serial and parallel parts of a program is known as Amdahl's law [51, 62]. Let

$$p = \# \text{ of CPUs} \quad T_1 = 1\text{-CPU time} \quad T_p = p\text{-CPU time} \quad (21.2)$$

The maximum speedup  $S_p$  attainable with parallel processing is thus

$$S_p^{\max} = \frac{T_1}{T_p} \rightarrow p \quad (21.3)$$

This limit is never met for a number of reasons: some of the program is serial, data and memory conflicts occur, communication and synchronization of the processors takes time, and it is rare to attain perfect load balance among all the processors. For the moment we ignore these complications and concentrate on how the *serial* part of the code affects the speedup. Let  $f$  to be the fraction of the program that potentially may run on multiple processors,

$$f = p \frac{T_p}{T_1} \quad (\text{fraction parallel}) \quad (21.4)$$

The fraction  $1 - f$  of the code that cannot be run in parallel must be run via serial processing, and thus takes time

$$T_s = (1 - f)T_1 \quad (\text{serial time}) \quad (21.5)$$

The time  $T_p$  spent on the  $p$  parallel processors is related to  $T_s$  by

$$T_p = f \frac{T_1}{p} \quad (21.6)$$

That being the case, the speedup  $S_p$  as a function of  $f$  and the number of processors is

$$S_p = \frac{T_1}{T_s + T_p} = \frac{1}{1 - f + f/p} \quad (\text{Amdahl's law}) \quad (21.7)$$

Some theoretical speedups are shown in Fig. 21.2 for different numbers  $p$  of processors. Clearly the speedup will not be significant enough to be worth the trouble unless most of the code is run in parallel (this is where we got the 90% figure from). Even an infinite number of processors cannot increase the speed of running the serial parts of the code, and so it runs at one processor speed. In practice, this means many problems are limited to a small number of processors, and that often for realistic applications, only 10–20% of the computer’s peak performance may be obtained.

### 21.3.1

#### Communication Overhead

As discouraging as Amdahl’s law may seem, it actually *overestimates* speedup because it ignores the *overhead* of parallel computation. Here we look at communication overhead. Assume a completely parallel code so that its speedup is

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{T_1/p} = p \quad (21.8)$$

The denominator assumes that it takes no time for the processors to communicate. However, it takes a finite time, called *latency*, to get data out of memory and into the cache or onto the communication network. When we add in this latency, as well as other times that make up the *communication time*  $T_c$ , the speedup decreases to

$$S_p \approx \frac{T_1}{T_1/p + T_c} < p \quad (\text{with communication time}) \quad (21.9)$$

For the speedup to be unaffected by communication time, we need to have

$$\frac{T_1}{p} \gg T_c \quad \Rightarrow \quad p \ll \frac{T_1}{T_c} \quad (21.10)$$

This means that as you keep increasing the number of processors  $p$ , at some point the time spent on computation  $T_1/p$  must equal the time  $T_c$  needed for communication, and adding more processors leads to greater execution time as the processors wait around more to communicate. This is another limit, then, on the maximum number of processors that may be used on any one problem, as well as on the effectiveness of increasing processor speed without a commensurate increase in communication speed.

The continual and dramatic increases in CPU speed, along with the widespread adoption of computer clusters, is leading to a changing view as to how to judge the speed of an algorithm. Specifically, CPUs are already so fast that the rate-determining step in a process is the slowest step, and that is often memory access or communication between processors. Such being the case, while the number of computational steps is still important for determining an algorithm's speed, the number and amount of memory access and interprocessor communication must also be mixed into the formula. This is currently an active area of research in algorithm development.

**22****Parallel Computing with MPI**

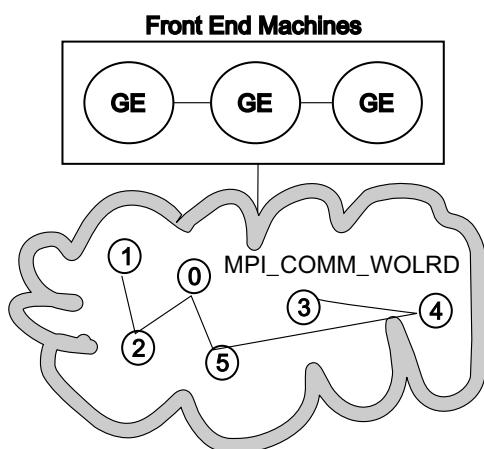
In this chapter, we present a tutorial on the use of MPI on a small Beowulf cluster composed of Unix or Linux computers. This follows our philosophy of “learning while doing.” Our presentation is meant to help a user from the ground up, something that might not be needed if you were working at a central computing center with a reasonable level of support. Although your **problem** is still to take the program you wrote to generate the bifurcation plot for bug populations and run different ranges of  $\mu$  values simultaneously on several CPUs, in a more immediate sense, your task is to get the experience of running MPI, to understand some of the MPI commands within the programs, and then to run a timing experiment. In Section 22.7, at the end of the chapter we give a listing and a brief description of the MPI commands and data types. General information about MPI is at [52], detailed information about the syntax of MPI commands is at [53], while other useful materials are at [54]. The standard reference on the C language is [63], although we prefer [64].<sup>1</sup> MPI is very much the standard software protocol for parallel computing, and is at a higher level than its predecessor PVM.

Nevertheless, we have found that the software for parallel computing is not as advanced as the hardware. There are just too many details for a computational scientist to worry about, and much of the coding is at a much lower, or more elementary level than that with which we usually deal. This we view as a major step backward from the high-level programming that was possible on the shared memory supercomputers of the past. This, apparently, is the price to be paid for the use of inexpensive clusters. Indeed, John Dongarra, a leading HPC scientist has called the state of parallel software a crisis, with programming still stuck in the 1960s.

While in the past we have run Java programs with a version of MPI, the different communication protocols used by MPI and Java have led to poor performance, or additional complications needed to improve performance [65].

<sup>1</sup> These materials were developed with the help of Kristopher Wieland, Kevin Kyle, Dona Hertel, and Phil Carter. Some of the other materials derive from class notes

from the Ohio Super Computer Center, which were supplied to us, and written in part, by Steve Gordon.



**Fig. 22.1** A schematic view of a cluster (cloud) connected to front end machines (box).

In addition, you usually would not bother parallelizing a program unless if it requires very large amounts of computing time or memory, and those types of programs are usually written in Fortran or C (both for historical reasons and because Java is slower). So it makes sense for us to use Fortran or C for our MPI examples. We will use C because it is similar to Java.

## 22.1

### Running on a Beowulf

A Beowulf cluster is a collection of independent computers each with its own memory and programs that are connected to each other by a fast communication network over which the messages are sent. MPI stands for *Message Passing Interface*. It is a library of commands that make communication between programs running on different computers possible. The messages sent are usually data contained in arrays. Because different processors cannot directly access the memory on some other computer, when a variable is changed on one computer, it does *not* get changed in the other copies of the program running on other processors. This is an example of where MPI comes into play.

In Fig. 22.1 we show a typical, but not universal, configuration for a Beowulf cluster. Almost all clusters have the common feature of using MPI for communication among computers and Unix/Linux for the operating system. The cluster in Fig. 22.1 is shown within a cloud. The cloud symbolizes the grouping and connection of what are still independent computers communicating via MPI (the lines). The `MPI_COMM_WORLD` within the cloud is an MPI

data type containing all the processors that are allowed to communicate with each other (in this case six).

The box in Fig. 22.1 represents the *front end* or *submit hosts*. These are the computers from which users submit their jobs to the Beowulf and later work with the output from the Beowulf. We have placed the front-end computers outside of the Beowulf cloud, although it could be within. This type of configuration frees the Beowulf cluster from administrative chores so that they may concentrate on number crunching, and is useful when there are multiple users on the Beowulf. However, some installations have the front-end machines as part of the Beowulf, which lets them be used for number crunching as well. Other installations, particularly those with only a single user, may have no front end and permit the users to submit their jobs on any computer.

Finally, note that we have placed the letters *GE* within the front-end machines. This represents a configuration in which these computers are also running some type of a *grid engine* or queuing system that oversees the running of jobs submitted to MPI by a number of users. For instance, if we have a cluster of 20 computers and user A requests 10 machines and user B requests 8 machines, then the grid engine would permit both users to run simultaneously and assign their jobs to different computers. However, if user A had requested 16 machines, then the grid engine would make one of the users wait while the other user gets their work done.

Some setup is required before you can run MPI on several computers at once. If someone has already done this for you, then you may skip the rest of this section and move on Section 22.2.2. Our instructions have been run on a cluster of Sun computers running Solaris Unix. You will have to change the computer names and such for your purposes, but the steps should remain the same.

- First you need to have an active account on each of the computers in the Beowulf cluster. Even if you usually run on a set of networked computers, it is likely that the Beowulf is set up to be separate.
- Open a shell on one of the Beowulf machines designated for users to sign on to (a *front end* machine). You probably do not have to be sitting at the front end to sign on, but instead can use *ssh* or *telnet*. Make the directory *mpi* in your home directory:

> <b>cd ~</b>	Change to home directory
> <b>mkdir output</b>	Screen output gets stored here first
> <b>mkdir output/error</b>	Place to store error messages
> <b>mkdir mpi</b>	A place to store your mpi stuff

- You need to have your Beowulf account configured so that Unix can find the MPI commands that you issue from the command line or from your programs. When you log onto the computer, the operating system reads a config-

uration file `.cshrc` file residing in your home directory. It contains the places where the operating system looks for commands. (We are assuming here that you are using either the `csh` or `tcsh`, if not, then modify your `.login`, which should work regardless of the shell.) When a file begins with a “dot,” it is usually hidden from view when you list files, but it can be seen with the command `ls -la`. The list of places where Unix looks for commands is an *environmental variable* called your `PATH`, and it needs to include the current version of the `mpich-n.m/bin` directory where the scripts for MPI reside, e.g.,

```
/usr/local/cluster/mpich-1.2.6/bin
```

This should be in your `PATH`

Here the `1.2.6` may need to be replaced by the current version of MPI that is installed on your computer, as well as possibly the directory name `cluster`.

- Because your `.cshrc` file controls your environment, having an error in this file can lead to a nonfunctional computer for you. And since the format is rather detailed and unforgiving, it is easy to make mistakes. So before you mess with your existing `.cshrc` file, make a backup copy of it:

```
> cp .cshrc .cshrc_bk
```

You can use this backup file as reference, or copy it back to `.cshrc` if things get to be too much of a mess. If you have really messed things up, then your system administrator may have to copy the file back for you.

- Edit your `.cshrc` file so that it contains a line in which `setenv PATH` includes `/usr/local/cluster/mpich-1.2.6/bin`. If you do not have a `.cshrc` file, just create one. Find a line containing `setenv PATH`, and add this in after one of the colons, making sure to separate the path names with colons. As an example, the `.cshrc` file for user `rubin` is

```
# @(#)cshrc 1.11 89/11/29 SMI umask 022
setenv PATH /usr/local/bin:/opt/SUNWspro/bin:/opt/SUNWrtvc/bin:
:/opt/SUNWste/bin:/usr/bin/X11:/usr/openwin/bin:/usr/dt/bin:/usr/ucb/:
/usr/ccs/bin:/usr/bin:/bin:/usr/sbin:/sbin:
/usr/local/cluster/mpich-1.2.6/bin: setenv PAGER less setenv
CLASSPATH /home/rubin:/home/rubin/dev/java/chapmanjava/classes/:
/home/rubin/dev/565/javacode/:
/home/rubin/dev/565/currproj:/home/rubin:/home/rubin/mpiJava:
/usr/local/mpiJava/lib/classes:
set prompt="%~::%m> "
```

- If you are editing your `.login` file, enter as the last line in the file:

```
set path = $path /usr/local/cluster/mpich-1.2.6/bin
```

- Because dotfiles are read by the system when you first log on, you will have to log off and back on for your changes to take effect. (You can use the `source` command to avoid logging off and on.) Once you have logged back on, check the values of your `PATH` environmental variable:

> **echo \$PATH** From Unix shell, tells you what Unix thinks

- Let us now take a look at what has been done to the computers to have them run as a Beowulf cluster. On Unix systems the “slash” directory / is the root or top directory. Change directory to /

> **cd /** Change to root directory

You should see files there, such as the kernel and the devices, that are part of the operating system. You may not be permitted to examine these files, which is a good thing since modifying them could cause real problems (it is the sort of thing that hackers and system administrators do).

- MPI is a *local* addition to the operating system. On our system the MPI and SGE commands and documentation [66] are in the /usr/local/cluster directory. Here the first / indicates the root directory and `usr` is the directory name under root. Change directory to /usr/local/cluster, or wherever MPI is kept on your system, and notice the directories `scripts` and `mpich-1.2.6` (or maybe just a symbolic link `mpich`). Feel free to explore these directories. The directory `scripts` contains various scripts designed to make running your MPI programs easier. (Scripts are small programs containing system commands that get executed in order when the file is run.)

- In the `mpich-1.2.6` directory, you will notice that there are examples in C, C++, and Fortran. Feel free to copy these to your home directory:

> **cp -r examples /home/userid/mpi**

where `userid` is your personal computer name. We encourage you to try out examples, although some may need modification to work on your local system.

- Further documentation can be found in

<code>/usr/local/cluster/mpich-1.2.6/doc/mpichman-chp4.pdf</code>	MPI documentation
<code>/usr/local/cluster/sge/doc/SGE53AdminUserDoc.pdf</code>	SGE documentation
<code>/usr/local/cluster/sge/doc/SGE53Ref.pdf</code>	SGE reference
<code>man qstat</code>	Manual page on <code>qstat</code>

- Copy the script `run_mpi.sh` from the `Codes/MPI/codes` on the CD to your personal `mpi` directory. This script contains the commands needed to run a program on a cluster.
- Copy the file `/usr/local/cluster/mpich/share/machines.solaris` to your home directory and examine it. (The `solaris` extender is there because we are using the *Solaris* version of the Unix operating system on our Beowulf, you may need to change this for your local system.) This file contains a list of all of the computers that are on the Beowulf cluster and available to MPI for use (though there is no guarantee that all machines are operative):

```
# Change this file to contain the machines that you want to use # to
run MPI jobs on. Format: 1 host per line, either hostname # or
hostname:n, where n is the number of processors. # hostname should
be the same as output from "hostname" command paul rose tomek manuel
```

### 22.1.1

#### An Alternative: BCCD = Your Cluster on a CD

One of the difficulties in learning how to parallel compute is the need for a parallel computer. Even though there may be many computers around that you may be able to use, knitting them all together into a parallel machine takes time and effort. However, if your interest is in learning about and experiencing distributed parallel computing, and not in setting up one of the fastest research machines in the world, then there is an easy way. It is called *bootable cluster CD* (BCCD) and is file on a CD. When you start your computer with the CD in place, you are given the option of having the computer ignore your regular operating system and instead run (boot) from the CD into a preconfigured distributed computing environment. The new system does not change your system, but rather is a nondestructive overlay on top of existing hardware that runs a full-fledged parallel computing environment on just about any workstation-class system, including Macs. You boot up every machine you wish to have on your cluster this way, and if needed, set up a DNS (Domain Name System) and DHCP (Dynamic Host Configuration Protocol) servers, which are also included. Did we mention that the system is free? [67]

### 22.2

#### Running MPI

If you are the only one working on a Beowulf cluster, then it may make sense to submit your jobs directly to MPI. However, if there is the possibility that a number of people may be using the cluster, or that you may be submitting a number of jobs to the cluster, then it is a good idea to use some kind of a queue management system to look after your jobs. This can avoid the inefficiency of having different jobs compete with each other for time and memory, or having the entire cluster “hang” because a job requested a processor that is not available. We use the *Sun Grid Engine* (SGE) [66]. This queue management system works on non-Sun machines and is free. When your program is submitted to a cluster via a management system, the system will install a copy of the same program on each computer assigned to run the program.

There are a number of scripts that interpret the MPI commands you give within your programs (the commands are not part of the standard Fortran or

C languages), and then call the standard compilers. These scripts are called *wrappers* because they surround the standard compilers as a way of extending them to include MPI commands

<b>mpicc</b>	C compiler	<b>mpicxx</b>	C++ compiler
<b>mpif77</b>	Fortran 77 compiler	<b>mpif90</b>	Fortran 90 compiler

Typically you compile your programs on the front end of the Beowulf, or the master machines, but not on the execution nodes. You use these commands just like you would the regular compiler commands, only now you may include MPI commands in your source program:

> <b>mpicc -o name name.c</b>	Compile name.c with MPI wrapper script
> <b>mpif77 -o name name.f</b>	Compile name.f with MPI wrapper script

### 22.2.1

#### MPI under a Queuing System

**Tab. 22.1** Some common SGE commands.

Command	Action
<b>qsub myscript</b>	Submit batch script or job <b>myscript</b>
<b>qhost</b>	Show job/host status
<b>qalter &lt;job_id&gt;</b>	Change parameters for job in queue
<b>qdel job_id</b>	Remove job_id
<b>qstat</b>	Display status of batch jobs
<b>qstat -f</b>	Full Listing for qstat
<b>qstat -u &lt;username&gt;</b>	User only for qstat
<b>qmon</b>	X Window Frontend (integrated functionality)
cpug, bradley, emma (local names)	Front end computers

Table 22.1 lists a number of commands that you may find useful if you are using the Sun Grid Engine (SGE). Other queuing systems should have similar commands. Once your program is compiled successfully, it can be executed on the cluster using these commands. The usual method of entering the program name or `a.out` at the prompt only runs it on the local computer; we want it to run on a number of machines. In order to run on the cluster, the program needs to be submitted to the queue management system. Listing 22.1 uses the `run_mpi.sh` script and the `qsub` command to submit jobs to in *batch* mode:

> <b>qsub runMPI.sh name</b>	Submit name to run on cluster
------------------------------	-------------------------------

This command returns a job ID number, which you should record to keep track of your program. *Note*, in order for this script to work, both `runMPI.sh` and `name` must be in the current directory. If you need to pass parameters to your program, place the program name and parameters in quotes:

> <b>qsub run_mpi.sh "name -r 10"</b>	Parameters and program name in quotes
---------------------------------------	---------------------------------------

**Listing 22.1:** The script `runMPI.sh` used to run an MPI program.

```

#
#----- SHORT COMMENT -----
# Template script for MPI jobs to run on mphase Grid Engine cluster.
# Modify it for your case and submit to CODINE with
# command "qsub mpi_run.sh".

# You may want to modify the parameters for
# "-N" (job queue name), "-pe" (queue type, numb of requested CPUs),
# "myjob" (your compiled executable).

# Can compile code, for example myjob.c (*.f), with GNU mpicc or
# mpif77 compilers as follows:
# "mpicc -o myjob myjob.c" or "mpif77 -o myjob myjob.f"

# You can monitor your jobs with command
# "qstat -u your_username" or "qstat -f" to see all queues.
# To remove your job, run "qdel job_id"
# To kill running job, use "qdel -f job_id"

# -----Attention: #$ is a special CODINE symbol, not a comment -----
#
#   The name, which will identify your job in the queue system
#$ -N MPI_job
#
#   Queue request, mpich. You can specify number of requested CPUs,
#   for example, from 2 to 3
#$ -pe class_mpi 4-6
#
# -----
#$ -cwd
#$ -o $HOME/output/$JOB_NAME-$JOB_ID
#$ -e $HOME/output/error/$JOB_NAME-$JOB_ID.error
#$ -v MPICH_HOME=/usr/local/cluster/mpich-1.2.6
# -----



echo "Got $NSLOTS slots."


# Don't modify the line below if you don't know what it is
$MPIR_HOME/bin/mpirun -np $NSLOTS $1

```

### 22.2.1.1 Status of Submitted Programs

After your program is successfully submitted, SGE places it into a queue where it waits for the requested number of processors to become available. It then executes your program on the cluster, and *directs the output to a file* in the `output` subdirectory within your home directory. The program itself uses the MPI and C/Fortran commands. In order to check the status of your submitted program, use `qstat` along with your Job ID number:

> `qstat 1263`

Tell me the status of JOB 1263

```
job-ID prior name user state submit/start at queue master ja-task-ID
1263 0 Test_MPI_J dhertel qw 07/20/2005 12:13:51
```

The is a typical output that you may see right after submitting your job. The `qw` in the `state` column indicates that the program is in the `queue` and `waiting` to be executed.

<pre>&gt; qstat 1263</pre>	Same as above, but at later time
<pre>job-ID prior name user state submit/start at queue master ja-task-ID 1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen11.q MASTER 1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen11.q SLAVE 1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen12.q SLAVE 1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen3.q SLAVE 1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen5.q SLAVE 1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen8.q SLAVE</pre>	

Here the program has been assigned a set of nodes (`eigenN` is the name of the computers), with the last column indicating whether that node is a master, host, slave, or guest (to be discussed further in Section 22.2.3). At this point the `state` column will have either a `t` indicating `transfer`, or an `r` indicating `running`.

The **output** from your run is sent to the file `Test_MPI.<jobID>.out` in the `output` subdirectory within your home directory. Error messages are sent to a corresponding file in the `error` subdirectory. Of course you can still output to a file in the current working directory, as well as **input** from a file.

## 22.2.2

### Your First MPI Program: MPIhello.c

Listing 22.2 gives the simple MPI program `hello2.c`. It has each of the processors print `Hello World` from processor #, followed by the rank of the processor (we will talk more about rank). Compile `hello.c` using :

```
> mpicc hello.c -o hello
```

Compilation via compiler wrapper

After successful compilation, an executable file `hello` should be placed in the directory in which you did the compilation. The program is executed via the script `runMPI.sh` (or `run_mpi.sh`), either directly, or by use of the management command `qsub`:

<pre>&gt; runMPI.sh hello</pre>	Run directly under MPI
<pre>&gt; qsub runMPI.sh hello</pre>	Run under management system

This script sets up the running of the program on the 10 processors, with processor 0 the host and processors 1–9 the guests.

**Listing 22.2:** The C program `MPIhello.c` containing MPI calls that gets each processor to say hello.

```
// MPIhello.c           has each processor prints hello to screen

#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] ) {
    int myrank;
    MPI_Init( &argc, &argv );                                // Initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);                  // Get CPU's rank
    printf( "Hello World from processor %d\n", myrank );
    MPI_Finalize( );                                         // Finalize MPI
    return 0;
}
```

### 22.2.3

#### MPIhello.c Explained

Here is what is contained in `MPIhello.c`:

- The inclusion of the MPI headers via the `#include "mpi.h"` statement on lines 2–3. These are short files that assist the C compiler by telling it the type of arguments that MPI functions use for input and output, without giving any details about the functions. (In Fortran we used `include "/usr/local/cluster/mpich-2.1.6/include/mpif.h"` after the program line.)
- The main method is declared with `int main(int argc, char *argv[])` statement, where `argc` is a pointer to the number of arguments, and `argv` is a pointer to the argument vector passed to main when you run the program from the shell. (*Pointers* are variable types that give the locations in memory where the values of the variables reside, rather than the variables' actual values.) These arguments are passed to MPI to tell it how many processors you desire.
- The `int myrank` statement declares the variable `myrank`, which stands for the *rank* of the computer. Each processor running your program under is assigned a unique number called its *rank* by MPI. This is how you tell the difference among the identical programs running on different CPUs.
- The processor that the program is executed on is called the *host* or *master*, and all other machines are called *guests* or *slaves*. The host always has `myrank = 0`, while all the other processors, based on who responds first, have their processor numbers assigned to `myrank`. This means that `myrank = 1` for the first guest to respond, 2 for the second, and so on. Giving each processor a unique value for `myrank` is critical to how parallel processing works.

- Lines 5 and 8 of `hello2.c` contain the `MPI_Init()` and `MPI_Finalize()` commands that initialize and then terminate MPI. All MPI programs must have these lines, with the MPI commands always placed between them. The `MPI_Init(&argv, &argc)` function call takes two arguments, both beginning with a `&` indicating pointers. These arguments are used for communication between the operating system and MPI.
- The `MPI_Comm_rank(MPI_COMM_WORLD, &myrank)` call returns a different value for `rank` for each processor running the program. The first argument is a predefined constant which tells MPI which grouping of processors to communicate with. Unless one has set up groups of processors, just use the default `a communicator MPI_COMM_WORLD`. The second argument is an integer which gets returned with the rank of the individual program.

When `hello2.c` is executed, each processor prints its rank to the screen. You should notice that it does not print the ranks in order, and that the order will probably be different each time you run the program. Take a look at the output (in the file `output/MPI_job-xxxx`). It should look something like this, depending on how many nodes were assigned to your job:

```
"Hello, world!" from node 3 of 4 on eigen3.science.oregonstate.local
"Hello, world!" from node 2 of 4 on eigen2.science.oregonstate.local
Node 2 reporting "Hello, world!" from node 1 of 4 on
eigen1.science.oregonstate.local "Hello, world!" from node 0 of 4 on
eigen11.science.oregonstate.local
```

It might seem that the line `Node 2 reporting` should come last in the output, but it may not. In fact, if you run the program a few times, the lines in the output may appear in different orders each time because each node runs its own code without waiting for any other nodes to catch up. In the output above, node 2 finished outputting two lines before nodes 0 and 1.

If processing order matters to obtain proper execution, call `MPI_Barrier(MPI_COMM_WORLD)` to synchronize the processors. It is similar to inserting a starting line at a race; a processor will stop and wait at this line until all other processors reach it, and then they all set off at the same time. However, modern programming practice suggests that you try to design your programs so that the processors do not have to synchronize often. Having a processor stop and wait obviously slows down the number crunching, and consequently removes some of the advantage of parallel computing. However, as a scientist it is more important to have correct results than fast ones, and so do not hesitate inserting barriers if needed.

**Exercise:** Modify `hello2.c` so that only the guest processors say hello. *Hint:* What do the guest processors all have in common?

## 22.2.4

**Send/Receive Messages: MPImessage2.c**

**Listing 22.3:** The C program `MPIMessage2.c` uses MPI commands to both send and receive messages. Note the possibility of blocking, in which the program waits for a message.

```
// MPIMessage2.c: source node sends message to dest

#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rank;
    int msg_size = 6;
    MPI_Status status;
    int tag = 10;
    int source = 0, dest = 1;
    MPI_Init(&argc,&argv);                                // Initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);                  // Get CPU's rank
    if (rank == source) {
        char *msg = "Hello";
        printf("Host about to send message: %s\n",msg);
        // Send message, may block till dest receives message
        MPI_Send(msg, msg_size, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else if (rank == dest) {
        char buffer[msg_size+1];
        // Receive message, may block till dest receives message
        MPI_Recv( &buffer, msg_size, MPI_CHAR, source, tag,
                  MPI_COMM_WORLD, &status );
        printf("Message received by %d: %s\n", rank, buffer);
    }
    printf("NODE %d done.\n", rank);                      // All nodes print
    MPI_Finalize();                                       // Finalize MPI
    return 0;
}
```

Sending and receiving data constitute the heart of parallel computing. The guest processors need to transmit the data they have processed back to the host, and the host has to assemble the data and then assign new work to the guests. An important aspect of MPI communication is that if one processor sends data, another processor must receive those same data. Otherwise the sending processor waits indefinitely for a signal that its data have been received! Likewise, after a processor executes a `receive` command, it waits until those data arrive.

There is a basic MPI command `MPI_Send` to send a message from a *source* node, and another basic command `MPI_Recv` needed for a *destination* node to receive it. The message itself must be an array, even if there is only one element in the array. We see these commands in use in `message2.c` in Listing 22.3. This program accomplishes the same thing as `hello.c`, but with send and receive

**Tab. 22.2** The arguments for MPI\_Send and MPI\_Recv.

Argument name	Description
<b>msg</b>	Pointer (& in front) to array to send/receive
<b>msg_size</b>	Size of array sent; may be bigger than actual size
<b>MPI_TYPE</b>	Predefined constant indicating variable type within array other possible constants: MPI_INTEGER, MPI_DOUBLE
<b>dest</b>	Rank of processor receiving message
<b>tag</b>	A number that uniquely identifies message
<b>comm</b>	A <i>communicator</i> , e.g., predefined constant MPI_COMM_WORLD
<b>source</b>	Rank of processor sending message; if receiving messages from any source, use predefined constant MPI_ANY_SOURCE
<b>status</b>	Pointer to variable type MPI_Status containing status info

commands. The host sends the message and prints out a message, while the guests also print when they receive a message. The forms of the send/receive commands are:

```
MPI_Send(msg, msg_size, MPI_TYPE, dest, tag, MPI_COMM_WORLD);      Send
MPI_Recv(msg, msg_size, MPI_TYPE, source, tag, comm, status);      Receive
```

The arguments and their descriptions are given in Tab. 22.2.4. The criteria for successfully sending and receiving a message are as follows.

1. The sender must specify a valid destination rank, and the processor of that rank must call MPI\_recv.
2. The receiver must specify a valid source rank or MPI\_ANY\_SOURCE.
3. The send and receive *communicators* must be the same.
4. The tags must match.
5. The receiver's message array must be large enough to hold the array.

**Exercise:** Modify message2.c so that all processors say hello. □

### 22.2.5

#### Receive More Messages: MPImessage3.c

**Listing 22.4:** The C program MPImessage3.c containing MPI's commands that have each guest processor send a message to the host processor, who then prints that guest's rank.

```
// MPImessage3.c: guests send rank to the host, who prints them

#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
```

```

int rank, size;
int msg_size = 6;
MPI_Status status;
int tag = 10;
int host = 0;
MPI_Init(&argc,&argv);                                // Initialize MPI
MPI_Comm_rank(MPI_COMM_WORLD,&rank);                  // Get CPU's rank
MPI_Comm_size(MPI_COMM_WORLD,&size);                  // Get number of CPUs
if (rank != host) {
    int n[1];                                         // Array of 1 integer
    n[0] = rank;
    printf("node %d about to send message\n", rank);
    MPI_Send(&n, 1, MPI_INTEGER, host, tag, MPI_COMM_WORLD);
}
else {
    int r[1];
    int i;
    for(i=1; i < size; i++) {
        MPI_Recv(&r, 1, MPI_INTEGER, MPI_ANY_SOURCE, tag,
                  MPI_COMM_WORLD, &status);
        printf("Message received: %d\n", r[0]);
    }
}
MPI_Finalize();                                       // Finalize MPI
return 0;
}

```

A bit more advanced use of message passing is given by `message2.c` in Listing 22.4. It has each guest processor sending a message to the host processor, who then prints out the rank of the guest which sent the message. Observe how we have the host looping through all the guests; otherwise it would stop looking for more messages after the first one arrives. In order to know how the number of processors, the host calls `MPI_Comm_size`.

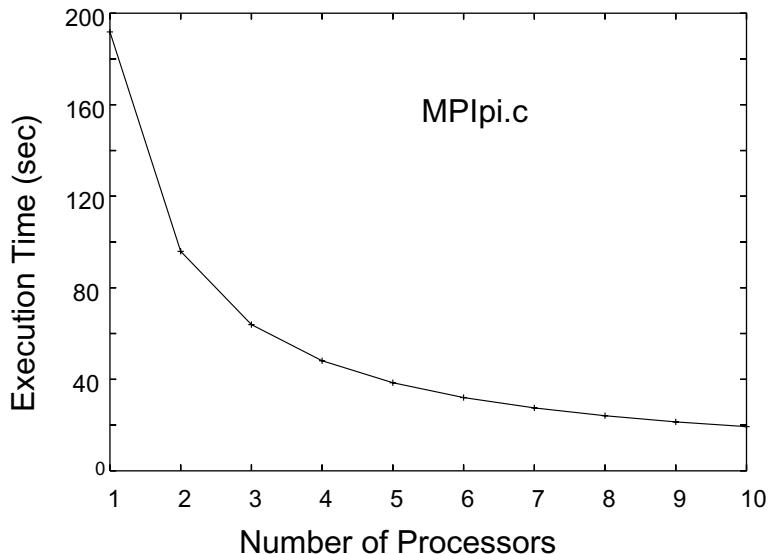
### 22.2.6

#### Broadcast Messages: `MPIpri.c`

If we used the same technique to send a message from one node to several other nodes, we would have to loop over calls to `MPI_Send`. In `MPIpri.c` in Listing 22.5 we see an easy way to send to all other nodes.

This is a simple program that computes  $\pi$  in parallel using the “stone throwing” technique discussed in Chap 11. Notice the new MPI commands:

- `MPI_Wtime` is used to return the wall time in seconds (the time as given by a clock on the wall). This is useful when computing speedup curves (Figs. 22.2 and 22.3).
- `MPI_Bcast` broadcasts sends out data from one processor to all others. In our case the host broadcasts the number of iterations to the guests, which, in turn, replace their current values of `n` with the one received from the host.



**Fig. 22.2** Execution time versus number of processors for the calculation of  $\pi$  with `MPipi.c`.

- **MPI\_Allreduce** is a glorified broadcast command. It collects the values of the variable `mypi` from each of the processors, performs an operation on them with `MPI_SUM`, and then broadcasts the result via the variable `pi`.

**Listing 22.5:** The C program `MPipi.c` uses a number of processors to compute  $\pi$  by a Monte Carlo rejection technique (stone throwing).

```
// MPipi.c computes pi in parallel by Monte Carlo Integration
#include "mpi.h" #include <stdio.h> #include <math.h>
double f(double);

int main(int argc, char *argv[]) {
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, startwtime=0., endwtime;
    int namelen;
    char processor_name [MPI_MAX_PROCESSOR_NAME];
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &myid );
    MPI_Get_processor_name( processor_name, &namelen );
    fprintf(stdout,"Process %d of %d is on %s\n", myid, numprocs,
            processor_name );
    fflush( stdout );
    n = 10000; // default # of rectangles
    if (myid == 0) startwtime = MPI_Wtime();
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD );
    h = 1. / (double) n;
    sum = 0.;
```

```

for (i = myid + 1; i <= n; i += numprocs) { // Better to work back
    x = h * ((double)i - 0.5);
    sum += f(x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0) {
    endwtime = MPI_Wtime();
    printf("pi is approximately %.16f, Error is %.16f\n",
           pi, fabs(pi - PI25DT));
    printf("wall clock time = %f\n", endwtime-startwtime);
    fflush(stdout);
}
MPI_Finalize();
return 0;
}

double f(double a) { return (4. / (1. + a*a));}           // Function f(a)

```

## 22.2.7

**Exercise**

In Fig. 22.2 we show our results for the speedup obtained by calculating  $\pi$  in parallel with `MPipi.c`. This exercise leads you through the steps in obtaining your own speedup curve:

1. Record how long each of your runs takes and how accurate the answers are. Does the roundoff error enter in? What could you do to get a more accurate value for  $\pi$ ?
2. There are two versions of a parallel program possible here. In the *active host* version the host acts just like a guest and does some work. In the *lazy host* version the host does no work but instead just controls the action. Does `MPipi.c` contain an active or lazy host? Change `MPipi.c` to the other version and record the difference in execution times.
3. Make a plot of the time versus number of processors for the calculation of  $\pi$ .
4. Make a *speedup* plot, that is, a graph of computation time divided by the time for one processor, versus the number of processors.  $\square$

**Listing 22.6:** The program `TuneMPI.c` is a parallel version of our program `Tune.java` used to test the effects of various optimization modifications.

```

/* TuneMPI.c: a matrix algebra program to be tuned for performance
   N X N Matrix speed tests using MPI */
#include "mpi.h"
#include <stdio.h>

```



```

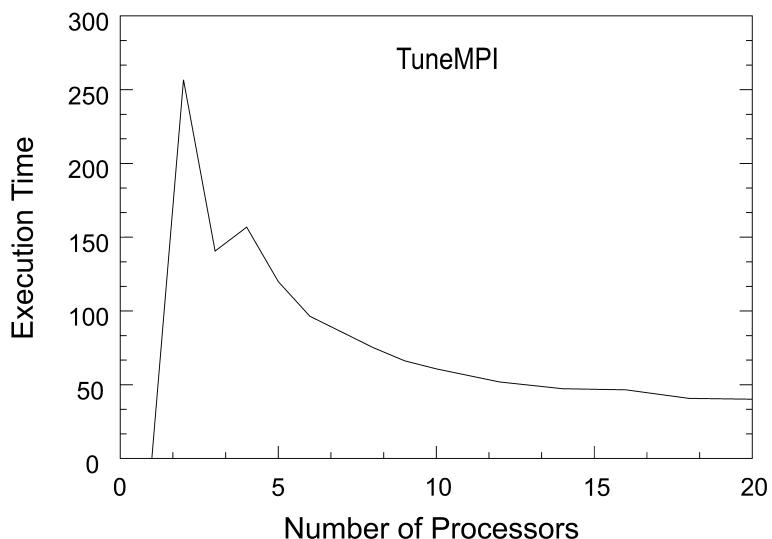
iter = iter + 1;
mycoef[0]=0.0;
ener[0] = 0.0; myener[0] = 0.0 ;
ovlp[0] = 0.0; myovlp[0] = 0.0 ;
err[0] = 0.0 ; myerr[0] = 0.0;
for (i= 1; i < N; i++) {
    h = (int)(i)%(nmach-1)+1 ;
    if (myrank == h){
        myovlp[0] = myovlp[0]+coef[i]*coef[i];
        mysigma[0] = 0.0;
        for ( j= 1; j < N; j++) {
            mysigma[0] = mysigma[0] + coef[j]*ham[j][i];
        }
        myener[0] = myener[0]+coef[i]*mysigma[0] ;
        MPI_Send(&mysigma, 1, MPI_DOUBLE, 0, h, MPI_COMM_WORLD);
    }
    if (myrank == 0) {
        MPI_Recv(&mysigma,1, MPI_DOUBLE,h,h,MPI_COMM_WORLD,&status );
        sigma[i]=mysigma[0];
    }
} // end of for(i...
MPI_Allreduce(&myener,&ener,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
MPI_Allreduce(&myovlp,&ovlp,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
MPI_Bcast(&sigma, N-1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ener[0] = ener[0]/(ovlp[0]);
for ( i = 1; i< N; i++) {
    h = (int)(i)%(nmach-1)+1 ;
    if (myrank == h) {
        mycoef[0] = coef[i]/sqrt(ovlp[0]);
        mysigma[0] = sigma[i]/sqrt(ovlp[0]);
        MPI_Send(&mycoef, 1, MPI_DOUBLE,0,nmach+h+1, MPI_COMM_WORLD);
        MPI_Send(&mysigma,1,MPI_DOUBLE,0,2*nmach+h+1,MPI_COMM_WORLD);
    }
    if (myrank == 0) {
        MPI_Recv(&mycoef, 1, MPI_DOUBLE, h, nmach+h+1,
        MPI_COMM_WORLD, &status );
        MPI_Recv(&mysigma, 1, MPI_DOUBLE, h, 2*nmach+h+1,
        MPI_COMM_WORLD, &status );
        coef[i]=mycoef[0];
        sigma[i]=mysigma[0];
    }
} // end of for(i...
MPI_Bcast(&sigma, N-1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&coef, N-1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i = 2; i < N ; i++) {
    h = (int)(i)%(nmach-1)+1;
    if (myrank == h) {
        step = (sigma[i] - ener[0]*coef[i])/(ener[0]-ham[i][i]);
        mycoef[0] = coef[i] + step;
        myerr[0] = myerr[0]+ pow(step,2);
        for ( k= 0; k <= N*N; k++) { // slowdown loop
            dummy = pow(dummy,dummy);
            dummy = pow(dummy,1.0/dummy);
        }
        MPI_Send(&mycoef,1,MPI_DOUBLE,0,3*nmach+h+1, MPI_COMM_WORLD);
    }
    if (myrank == 0) { // end of if(myrank...
}

```

```

        MPI_Recv(&mycoef, 1, MPI_DOUBLE, h, 3*nmach+h+1,
                  MPI_COMM_WORLD, &status);
        coef[i]=mycoef[0];
    }
} // end of for (i ...
MPI_Bcast(&coef, N-1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Allreduce(&myerr,&err, 1,MPI_DOUBLE,MPI_SUM, MPI_COMM_WORLD);
err[0] = sqrt(err[0]);
if (myrank==0) {
    printf("\t%d\t%g\t%g\n", iter, ener[0], err[0]);
    fflush(stdout);
}
} // end whileloop
// output elapsed time
if (myrank == 0) {
    systime_f = time(NULL);
    difftime = ((long) systime_f) - ((long) systime_i);
    printf("\n\tTotal wall time = %d s\n", difftime);
    fflush(stdout);
    timempi[1] = MPI_Wtime();
    printf("\n\tMPI time= %g s\n", (timempi[1]-timempi[0]));
    fflush(stdout);
}
MPI_Finalize();
}

```



**Fig. 22.3** Execution time versus number of processors for the solution of an eigenvalue problem with TuneMPI.c. Note, the single processor result here does *not* include the overhead of running MPI.

### 22.3

#### Parallel Tuning: TuneMPI.c

Recall the `Tune` program that we experimented with in Chap. 13 to determine how memory access for a large matrix affects the running time of programs. You may also recall that as the size of the matrix was made larger, the execution time increased more rapidly than did the number of operations the program had to perform. The extra increase came from the time it took to transfer the needed matrix elements in and out of central memory. Because parallel programming on a multiprocessor also involves a good deal of data transfer, `Tune` program is also a good teaching tool for seeing how communications costs affect parallel computations.

In Listing 22.6 is the program `TuneMPI.c`. In Fig. 22.3 is the speedup curve we obtained by running it. This is a modified version of the `Tune` program in which each row of the large matrix multiplication is performed on a different processor using MPI. Explicitly,

$$\begin{matrix} [H]_{N \times N} & \times & [\Psi]_{N \times 1} & (22.1) \\ \left[ \begin{array}{c} \Rightarrow \text{ rank 1 } \Rightarrow \\ \Rightarrow \text{ rank 2 } \Rightarrow \\ \Rightarrow \text{ rank 3 } \Rightarrow \\ \Rightarrow \text{ rank 1 } \Rightarrow \\ \Rightarrow \text{ rank 2 } \Rightarrow \\ \Rightarrow \text{ rank 3 } \Rightarrow \\ \Rightarrow \text{ rank 1 } \Rightarrow \\ \vdots \end{array} \right]_{N \times N} & \times & \left[ \begin{array}{c} \psi_1 \downarrow \\ \psi_2 \downarrow \\ \psi_3 \downarrow \\ \psi_4 \downarrow \\ \psi_5 \downarrow \\ \psi_6 \downarrow \\ \psi_7 \downarrow \\ \vdots \end{array} \right]_{N \times 1} & (22.2) \end{matrix}$$

where the arrows indicate how each row of  $H$  gets multiplied by the single column of  $\Psi$ , with the multiplication of each row performed on a different processor (rank). The assignment of rows to processors continues until we run out of processors, and then starts all over again. Since this multiplication gets repeated for a number of iterations, this is the most computationally intensive part of the program, and so it makes sense to parallelize it.

However, even if the matrix is large, the `Tune` program is not computationally intensive enough to overcome the cost of communication inherent in parallel computing. Consequently, to increase computing time we have inserted an inner `for` loop over  $k$  that takes up time but accomplishes nothing (we have all had days like that). Slowing down the program should help make the speedup curve more realistic.

##### 22.3.0.1 TuneMPI.c Exercise

1. Compile `TuneMPI.c`:

```
> mpicc TuneMPI.c -lm -o TuneMPI
```

Compilation

Here `-lm` loads the math library and `-o` places the object in `TuneMPI`. This is your base program. It will use one processor as the host and another one to do the work.

2. To determine the speedup with multiple processors, you need to change the `runMPI.sh` script. Open it with an editor and find the line of the form:

```
#$ -pe class_mpi 1-4
```

A line in `runMPI.sh` script

The last number on this line tells the cluster the maximum number of processors to use. Change this to the number of processors you want to use. Use a number from 2 up to 10; starting with one processor leads to an error message as that leaves no processor to do the work. After changing `runMPI.sh`, run the program on the cluster. With the SGE management system this is done via

```
> qsub runMPI.sh TuneMPI
```

Submit to queue via SGE

3. You are already familiar with the scalar version of the `Tune` program. Find the scalar version of `Tune.c` (and add the extra lines to slow the program down), or modify the present one so that it runs on only one processor. Run the scalar version of `TuneMPI` and record the time it takes. Because there is overhead associated with running MPI, we would expect the scalar program to be faster than an MPI program running on a single processor.

4. Open another window and watch the processing of your MPI jobs on the host computer. Check that all temporary files are removed.

5. You now want to collect data for a plot of running time versus number of machines. Make sure your matrix size is large, say with `N=200` and up. Run `TuneMPI` on a variable number of machines, starting at 2, until you find no appreciable speedup (or an actual slowdown) with an increasing number of machines.

6. *Warning:* While you will do no harm running on the Beowulf when others are also running on it, in order to get meaningful and repeatable speedup graphs, you need to have the cluster all to yourself. Otherwise, the time it takes to switch around jobs and to setup and drop communications may slow down your runs significantly. A management system should help with this. If you are permitted to log in directly to the Beowulf machines, you can check what is happening via `who`:

```
> rsh rose who
```

Who is running on rose?

```
> rsh rubin who
```

Who is running on rubin?

```
> rsh emma who
```

Who is running on emma?

7. Increase the matrix size in steps and record how this affects the speedups. Remember, once your code is communications bound due to memory access, distributing it over many processors probably will make things worse!

## 22.4

### A String Vibrating in Parallel

**Listing 22.7:** The solution of the sting equation on several processors via MPIstring.c.

```
// Code listing for MPIstring.c

#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define maxt 10000           // Number of time steps to take
#define L 10000              // Number of divisions of the string
#define rho 0.01              // Density per length (kg/m)
#define ten 40.0               // Tension (N)
#define deltat 1.0e-4          // Delta t (s)
#define deltax .01             // Delta x (m)
#define skip 50                // Number of time steps to skip before printing

/* Need sqrt(ten/rho) <= deltax/deltat for a stable solution
   Decrease deltat for more accuracy , c' = deltax/deltat */

main(int argc, char *argv[]) {

    const double scale = pow(deltat/deltax,2)*ten/rho;
    int i, j, k, myrank, numprocs, start, stop, avgwidth, maxwidth, len;
    double left, right, startwtime, init_string(int index);

    FILE *out;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);           // Get my rank
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);        // Number of processors
    MPI_Status status;
    if (myrank == 0) {
        startwtime = MPI_Wtime();
        out = fopen("eqstringmpi.dat","w");
    }
    // assign a string to each node
    // 1st and last points (0 and L-1) -must =0, else error
    // Thus L-2 segments for numprocs processors
    avgwidth = (L-2)/numprocs;
    start = avgwidth*myrank+1;
    if (myrank < numprocs - 1) stop = avgwidth*(myrank+1);
    else stop = L-2;
    if (myrank == 0) maxwidth = L-2 - avgwidth*(numprocs-1);
    else maxwidth = 0;
    double results[maxwidth];           // Holds print for master
    len = stop - start;                 // Length of the array - 1
    double x[3][len+1];
    for (i=start; i <= stop; i++) x[0][i-start] = init_string(i);
    x[1][0] = x[0][0]+0.5*scale*(x[0][1]
                                  +init_string(start-1)-2.*x[0][0]);
    x[1][len] = x[0][len]           // 1st time step
                + 0.5*scale*(init_string(stop+1)+x[0][len-1]-2.0*x[0][len]);
    for (i=1; i < len; i++)
        x[1][i] = x[0][i]+0.5*scale*(x[0][i+1]+x[0][i-1]-2.0*x[0][i]);
    for(k=1; k<maxt; k++) {           // Later time steps
        if (myrank == 0) {           // Send to R, get from L
            MPI_Scatter(x[0], len, MPI_DOUBLE, x[1], len, MPI_DOUBLE, 1, MPI_COMM_WORLD);
            MPI_Gatherv(x[1], len, MPI_DOUBLE, x[0], &len, &len, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }
        else MPI_Gatherv(x[1], len, MPI_DOUBLE, x[0], &len, &len, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }
}
```

```

        MPI_Send(&x[1][len],1,MPI_DOUBLE,1,1,MPI_COMM_WORLD);
        left = 0.0;
    }
    else if (myrank < numprocs - 1) MPI_Sendrecv(&x[1][len],1,
        MPI_DOUBLE, myrank+1,1,&left,1,MPI_DOUBLE,myrank-1,1,
        MPI_COMM_WORLD, &status);
    else MPI_Recv(&left,1,MPI_DOUBLE,myrank-1,
        1,MPI_COMM_WORLD,&status);
    if (myrank == numprocs - 1) { // Send to L & get from R
        MPI_Send(&x[1][0],1,MPI_DOUBLE,myrank-1,2,MPI_COMM_WORLD);
        right = 0.0;
    }
    else if (myrank>0) MPI_Sendrecv(&x[1][0],1,MPI_DOUBLE,myrank-1,
        2,&right,1,MPI_DOUBLE,myrank+1,2,MPI_COMM_WORLD,&status);
    else MPI_Recv(&right,1,MPI_DOUBLE,1,2,MPI_COMM_WORLD,&status);
    x[2][0] = 2.0*x[1][0]-x[0][0]+scale*(x[1][1]+left-2.0*x[1][0]);
    for(i = 1; i < len; i++) x[2][i] = 2.0*x[1][i]-x[0][i]
        + scale*(x[1][i+1]+x[1][i-1]-2.0*x[1][i]);
    x[2][len] = 2.0*x[1][len]-x[0][len]
        + scale*(right+x[1][len-1]-2.0*x[1][len]);
    for(i = 0; i <= len; i++) {
        x[0][i] = x[1][i];
        x[1][i] = x[2][i];
    }
    if ((k%skip) == 0) { // Print using gnuplot 3D grid format
        if (myrank != 0)
            MPI_Send(&x[2][0],len+1,MPI_DOUBLE,0,3,MPI_COMM_WORLD);
        else {
            fprintf(out,"%f\n",0.0); // Left edge of (always 0)
            for (i=0; i < avgwidth; i++) fprintf(out,"%f\n",x[2][i]);
            for (i=1; i < numprocs-1; i++) {
                MPI_Recv(results,avgwidth,MPI_DOUBLE,i,3,MPI_COMM_WORLD,
                    &status);
                for (j=0;j<avgwidth;j++) fprintf(out,"%f\n",results[j]);
            }
            MPI_Recv(results,maxwidth,MPI_DOUBLE,numprocs-1,3,
                MPI_COMM_WORLD,&status);
            for (j=0; j < maxwidth; j++)
                fprintf(out,"%f\n",results[j]);
            fprintf(out,"%f\n",0.0); // R edge
            fprintf(out,"\n"); // Empty line for gnuplot
        }
    }
    if (myrank == 0)
        printf("Data in eqstringmpi.dat\nComputation time: %f s\n",
            MPI_Wtime()-startwtime);
    MPI_Finalize();
    exit(0);
}

double init_string(int index) {
    if (index < (L-1)*4/5) return 1.0*index/((L-1)*4/5);
    return 1.0*(L-1-index)/((L-1)-(L-1)*4/5);
    // Half of a sine wave
}

```

The program `MPIstring.c` given in Listing 22.7 is a parallel version of the solution of the wave equation (`eqstring.c`) discussed in Chap. 25. The algorithm calculates the future ([2]) displacement of a given string element from the present ([1]) displacements immediately to the left and right of that section, as well as the present and past ([0]) displacements of that element. The program is parallelized by assigning different sections of the string to different nodes, and by having the nodes communicate the displacements at their endpoints via the `MPI_Send()` and `MPI_Recv()`. The first argument to `MPI_Send()` are *pointers* to the data being sent, and the last arguments are of the type `MPI_Status`. A pointer in C, denoted by the ampersand `&`, gives the address of the variable that follows. The `MPI_Recv()` and `MPI_Send()` commands require a pointer as their first argument, or an array element (which in C is the address of the first element in the array). When sending more than one element of an array to `MPI_Send()`, you send a pointer to the first element of the array as the first argument, and then the number of elements as the second argument.

Observe near the end of the program how the `MPI_Send()` call is used to send `len+1` elements of the two-dimensional array `x[][]`, starting with the element `x[2][0]`. MPI will send these elements in the order in which they are stored in memory. In C, arrays are stored in row-major order with the first element of the second row immediately following the last element of the first row, and so on. Accordingly, this `MPI_Send()` call sends `len+1` elements of row 2, starting with column 0, which means all of row 2. If we had specified `len+5` elements instead, MPI would have sent all of row 2, plus the first four elements of row 3.

In `MPIstring.c`, the calculated future position of the string is stored in row 2 of `x[3][len+1]`, with different sections of the string stored in different columns. Row 1 stores the present positions, and row 0 stores the past positions. This is different from column-based algorithm used in the serial program `eqstring.c`, which followed the original Fortran program, which was column, rather than row based. This was changed because MPI reads data by rows.

The initial displacement of the string is given by the user-supplied function `init_string()`. Because the first time step requires only the initial displacement, no message passing is necessary. For later times each node sends the displacement of its rightmost point to the node with the next highest rank. This means that the rightmost node (`rank = numprocs - 1`) does not send data, and that the master (`rank = 0`) does not receive data. Communication is established via the `MPI_Sendrecv()` command, with the different `sends` and `receives` using tags to ensure proper delivery of the messages.

Next in the program, the nodes (representing string segments) send to and receive data from the segment to their right. All of these sends and receives

have a tag of 2. After every 50 iterations, the master collects the displacement of each segment of the string and outputs it. Each slave sends the data for the future time with a tag of 3. The master first outputs its own data and then calls `MPI_Recv()` for each node, one at a time, printing the data it receives.

#### 22.4.1

#### **MPIstring.c Exercise**

1. Ensure that the input data (`maxt`, `L`, `scale`, `skip`) in `MPIstring.c` are the same as those in `eqstring.c` (`maxt` in `MPIstring.c` is called `max` in `eqstring.c`).
2. Ensure that `init_string()` returns the initial configuration that is used in `eqstring.c`.
3. Compile and run `eqstring.c` via

```
> gcc eqstring.c -o eqstring -lm                                         Compile C code
> ./eqstring                                                               Run in present directory
```

4. Run both programs to ensure that they produce the same output. (This is easy to check in Unix with the `diff` command.)

**Listing 22.8:** The MPI program `MPIdeadlock.c` illustrates deadlock (waiting for receive). The code `MPIdeadlock-fixed.c` in Listing 22.9 removes the block.

```
// Code listing for MPIdeadlock.c
#include <stdio.h>
#include "mpi.h"
#define MAXLEN 100
main(int argc, char *argv[]) {

    int myrank, numprocs, fromrank, torank;
    char tosend[MAXLEN], received[MAXLEN];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    if (myrank == 0) fromrank = numprocs - 1;
    else fromrank = myrank - 1;
    if (myrank == numprocs - 1) torank = 0;
    else torank = myrank + 1;
    // Save string to send in tosend:
    sprintf(tosend,"Message from node %d to %d\n", myrank,torank);
    MPI_Recv(received ,MAXLEN,MPI_CHAR,fromrank ,0 ,MPI_COMM_WORLD,
             &status);
    MPI_Send(tosend ,MAXLEN,MPI_CHAR,torank ,0 ,MPI_COMM_WORLD);
    printf("%s",received); // Print string after successful receive
    MPI_Finalize();
    exit(0);
}
```

## 22.5

### Deadlock

It is important to avoid *deadlock* when using the `MPI_Send()` and `MPI_Recv()` commands. Deadlock occurs when one or more nodes wait for a nonoccurring event to occur. This can arise if each node waits to receive a message that is not sent. Compile and execute `deadlock.c`:

```
> mpicc deadlock.c -o deadlock                                         Compile
> qsub run_mpi.sh deadlock                                              Execute
```

Take note of the job id returned, which we will call `xxxx`. Wait for a few seconds, and then look at the output of the program

```
> cat output/MPI_job-xxxx                                              Examine output
```

The output should list how many nodes ("slots") were assigned to the job. Because all these nodes are now deadlocked, we need to cancel this job:

```
> qdel xxxx                                                 Cancel deadlocked job
> qdel all                                                 Alternate cancel
```

There are a number of ways to avoid deadlock. The program `MPIstring.c` used the function `MPI_Sendrecv()` to handle much of the message passing, and this will not cause deadlock. It is possible to use `MPI_Send()` and `MPI_Recv()`, but you should be careful to avoid deadlock, as we do in `MPIdeadlock-fixed.c`.

**Listing 22.9:** The code `MPIdeadlock-fixed.c` removes the deadlock present in `MPIdeadlock.c`.

```
// deadlock-fixed.c: deadlock.c without deadlock by Phil. Carter

#include <stdio.h>
#include "mpi.h"
#define MAXLEN 100

main(int argc, char *argv[]) {
    int myrank, numprocs, torank, i;
    char tosend[MAXLEN], received[MAXLEN];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    if (myrank == numprocs - 1) torank = 0;
    else torank = myrank + 1;
    // Save the string to send in the variable tosend:
    sprintf(tosend,"Message from node %d to %d\n", myrank, torank);
    for (i = 0; i < numprocs; i++) {
        if (myrank == i)
            MPI_Send(tosend ,MAXLEN,MPI_CHAR,torank ,i ,MPI_COMM_WORLD);
        else if (myrank == i+1 || (i == numprocs - 1 && myrank == 0))
            MPI_Recv(received ,MAXLEN,MPI_CHAR,i ,i ,MPI_COMM_WORLD,&status);
    }
}
```

```

    printf("%s",received);           // Print after successful receive
    MPI_Finalize();
    exit(0);
}

```

### 22.5.1

#### Nonblocking Communication

`MPI_Send()` and `MPI_Recv()`, as we have seen, are susceptible to deadlock because they block the program from continuing until the send or receive is finished. This method of message passing, called *blocking communication*. One way to avoid deadlock is to use nonblocking communication such as `MPI_Isend()`, which returns before the send is complete and thus frees up the node. Likewise, a node can call `MPI_Irecv()`, which does not wait for the receive to complete. Note that a node can receive a message with `MPI_Recv()` even if were sent using `MPI_Isend()`, and similarly, receive a message using `MPI_Irecv()` even if were sent with `MPI_Send()`.

There are two ways to determine whether a nonblocking send or receive has finished. One is to call `MPI_Test()`. It is then your choice as to whether you want to wait for the communication to complete (for example, to ensure that all string segments are at current time and not past time). To wait, call `MPI_Wait()`, which blocks execution until communication is complete. When you start a nonblocking send or receive, you get a request handle of data type `MPI_Request` to identify the communication you may need to wait for. When using nonblocking communication, you do have to ensure that you do not use the data being communicated until the communication has completed. You can check for this using `MPI_Test()`, or wait for completion using `MPI_Wait()`.

**Exercise:** Rewrite `deadlock.c` so that it avoids deadlock by using nonblocking communication. *Hint:* replace `MPI_Recv()` by `MPI_Irecv()`. □

### 22.5.2

#### Collective Communication

Several MPI commands that automatically exchange messages among multiple nodes at the same time. This is called collective communication, as opposed to the point-to-point communication between just two nodes. The program `MPIp1.c` already introduced the `MPI_Reduce()` command. It receives data from multiple nodes, performs an operation on the data, and stores the result on one node. The program `tunempi.c` used a similar function `MPI_Allreduce()` that does the same thing, but stores the result on every

node. The latter program also used `MPI_Bcast()`, which allows a node to send the same message to multiple nodes.

Collective commands communicate among a group of nodes specified by a communicator, such as `MPI_COMM_WORLD`. For example, in `MPIp1.c` we called `MPI_Reduce()` to receive results from every node, including itself. Other collective communication functions include `MPI_Scatter()` that has one node send messages to every other node. This is similar to `MPI_Bcast()`, but the former sends a different message to each node. Specifically, it breaks an array up into pieces of specified lengths, and sends the different pieces to different nodes. Likewise, `MPI_Gather()` gathers data from every node (including the root node), and places the data into an array, with data from node 0 placed first, followed by node 1, etc. A similar function, `MPI_Allgather()`, stores the data on every node, rather than just the root node.

## 22.6

### Supplementary Exercises

1. **Bifurcation Plot:** If you have not yet done it, take the program you wrote to generate the bifurcation plot for bug populations and run different ranges of  $\mu$  values simultaneously on several CPUs.
2. **Processor Ring:** Write a program in which
  - (a) a set of processes are arranged in a ring,
  - (b) each process stores its rank in `MPI_COMM_WORLD` in an integer,
  - (c) each process passes this on to its neighbor on the right,
  - (d) each processor keeps passing until it receives its rank back.
3. **Ping Pong:** Write a program in which two processes repeatedly pass a message back and forth. Insert timing calls to measure the time taken for one message, and determine how the time taken varies with the size of the message.
4. **Broadcast:** Have processor 1 send the same message to all the other processors and then receive messages of the same length from all the other processors. How does the time taken vary with the size of the messages and the number of processors?

## 22.7

### List of MPI Commands

#### MPI Data Types and Operators

MPI defines some of its own data types. The following are data types used as arguments to the MPI commands.

- **MPI\_Comm:** A communicator, used to specify group of nodes, most commonly `MPI_COMM_WORLD` for all the nodes.
- **MPI\_Status:** A variable holding status information returned by functions such as `MPI_Recv()`.
- **MPI\_Datatype:** Predefined constant indicating type of data being passed in a function such as `MPI_Send()` (see below).
- **MPI\_O:** Predefined constant indicating operation you want performed on data in functions such as `MPI_Reduce()` (see below).
- **MPI\_Request:** Request handle to identify a nonblocking `send` or `receive`, for example, when using `MPI_Wait()` or `MPI_Test()>`

#### Predefined constants: MPI\_Op and MPI\_Datatype

For a more complete of constants used in MPI, see

<http://www-unix.mcs.anl.gov/mpi/www/www3/Constants.html>.

MPI_OP	Description	MPI_Datatype	C Data Type
<code>MPI_MAX</code>	Maximum	<code>MPI_CHAR</code>	char
<code>MPI_MIN</code>	Minimum	<code>MPI_SHORT</code>	short
<code>MPI_SUM</code>	Sum	<code>MPI_INT</code>	int
<code>MPI_PROD</code>	Product	<code>MPI_LONG</code>	long
<code>MPI LAND</code>	Logical and	<code>MPI_FLOAT</code>	float
<code>MPI_BAND</code>	Bitwise and	<code>MPI_DOUBLE</code>	double
<code>MPI_LOR</code>	Logical or	<code>MPI_UNSIGNED_CHAR</code>	unsigned char
<code>MPI_BOR</code>	Bitwise or	<code>MPI_UNSIGNED_SHORT</code>	unsigned short
<code>MPI_LXOR</code>	Logical exclusive or	<code>MPI_UNSIGNED</code>	unsigned int
<code>MPI_BXOR</code>	Bitwise exclusive or	<code>MPI_UNSIGNED_LONG</code>	unsigned long
<code>MPI_MINLOC</code>	Find node's min and rank		
<code>MPI_MAXLOC</code>	Find node's max and rank		

#### MPI Commands

Below we list and identify the MPI commands used in this chapter. For the syntax of each command, along with many more MPI commands, see <http://www-unix.mcs.anl.gov/mpi/www/>. There each command name is a hyperlink to a full description.

#### Basic MPI Commands

- **MPI\_Send.** Sends a message.

- **MPI\_Recv.** Receives a message.
- **MPI\_Sendrecv.** Sends and receives a message.
- **MPI\_Init.** Starts MPI at the beginning of program.
- **MPI\_Finalize.** Stops MPI at the end of program.
- **MPI\_Comm\_rank.** Determine a node's rank.
- **MPI\_Comm\_size.** Determine number of nodes in communicator.
- **MPI\_Get\_processor\_name.** Determines name of the processor.
- **MPI\_Wtime.** Returns wall time in seconds since arbitrary time in the past.
- **MPI\_Barrier.** Blocks until all nodes have called this function.

### Collective Communication

- **MPI\_Reduce.** Performs operation on all copies of variable and stores result on single node.
- **MPI\_Allreduce.** Like `MPI_Reduce`, but stores result on all nodes.
- **MPI\_Gather.** Gathers data from group of nodes and stores them on one node.
- **MPI\_Allgather.** Like `MPI_Gather`, but stores the result on all nodes
- **MPI\_Scatter.** Sends different data to all other nodes (opposite of `MPI_Gather`).
- **MPI\_Bcast.** Sends same message to all other processes.

### Nonblocking Communication

- **MPI\_Isend.** Begins a nonblocking send.
- **MPI\_Irecv.** Begins a nonblocking receive.
- **MPI\_Wait.** Waits for an MPI send or receive to complete.
- **MPI\_Test.** Tests for the completion of a send or receive.

**23****Electrostatics Potentials via Finite Differences (PDEs)****23.1****PDE Generalities**

Physical quantities such as temperature and pressure vary continuously in both space and time. Such being our world, the function or *field*  $U(x, y, z, t)$  used to describe these quantities must contain independent space and time variable. As time evolves, the changes in  $U(x, y, z, t)$  at any one position affects the field at neighboring points. This means that the dynamical equations describing the dependence of  $U$  on four independent variables must be written in terms of partial derivatives, and therefore the equations must be *partial differential equations* (PDEs), in contrast to ordinary differential equations (ODEs).

The most general form for a PDE with two independent variables is

$$A \frac{\partial^2 U}{\partial x^2} + 2B \frac{\partial^2 U}{\partial x \partial y} + C \frac{\partial^2 U}{\partial y^2} + D \frac{\partial U}{\partial x} + E \frac{\partial U}{\partial y} = F \quad (23.1)$$

where  $A$ ,  $B$ ,  $C$ , and  $F$  are arbitrary functions of the variables  $x$  and  $y$ . In the table below we define the classes of PDEs by the value of the discriminant  $d$  in the second row [68], with the next two rows being examples:

<i>Elliptic</i>	<i>Parabolic</i>	<i>Hyperbolic</i>
$d = AC - B^2 > 0$	$d = AC - B^2 = 0$	$d = AC - B^2 < 0$
$\nabla^2 U(x) = -4\pi\rho(x)$	$\nabla^2 U(\mathbf{x}, t) = a\partial U/\partial t$	$\nabla^2 U(\mathbf{x}, t) = c^{-2}\partial^2 U/\partial t^2$
Poisson's	Heat	Wave

We usually think of a parabolic equation as containing a first-order derivative in one variable and second-order in the other; a hyperbolic equation as containing second-order derivatives of all the variables, with opposite signs

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5

when placed on the same side of the equal sign; and an elliptic equation as containing second-order derivatives of all the variables, with all having the same sign when placed on the same side of the equal sign.

**Tab. 23.1** The relation between boundary conditions and uniqueness for PDEs.

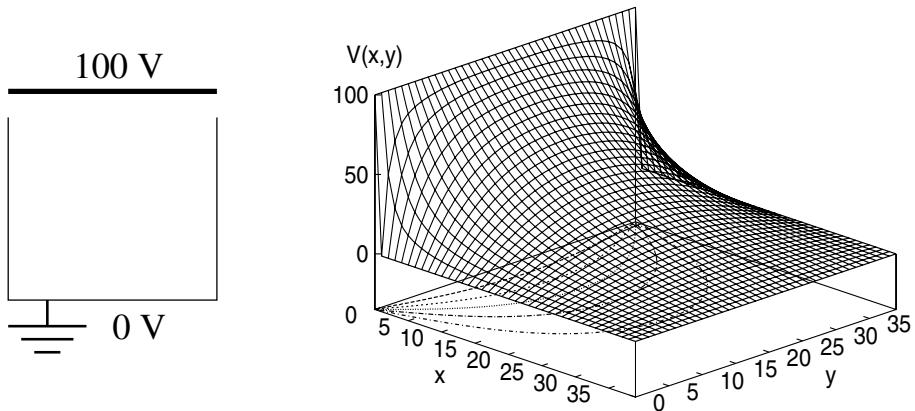
Boundary condition	Elliptic (Poisson equation)	Hyperbolic (Wave equation)	Parabolic (Heat equation)
Dirichlet open surface	Underspecified	Underspecified	<i>Unique and stable (1D)</i>
Dirichlet closed surface	<i>Unique and stable</i>	Overspecified	Overspecified
Neumann open surface	Underspecified	Underspecified	<i>Unique and stable (1D)</i>
Neumann closed surface	<i>Unique and stable</i>	Overspecified	Overspecified
Cauchy open surface	Unphysical	<i>Unique and stable</i>	Overspecified
Cauchy closed surface	Overspecified	Overspecified	Overspecified

After solving enough problems, one often develops some physical intuition as to whether one has sufficient *boundary conditions* for there to exist a unique solution for a given physical situation (this, of course, is in addition to requisite *initial conditions*). For instance, a string tied at both ends, or a heated bar placed in an infinite heat bath, are physical situations for which the boundary conditions are adequate. If the boundary condition is the value of the solution on a surrounding closed surface, we have a *Dirichlet boundary condition*. If the boundary condition is the value of the normal derivative on the surrounding surface, we have a *Neumann boundary condition*. If both the value of solution and its derivative are specified on a closed boundary, we have a *Cauchy boundary condition*. Although having an adequate boundary condition is necessary for a unique solution, having too many boundary conditions, for instance, both Neumann and Dirichlet, may be an overspecification for which no solution exists. (Although conclusions drawn for exact PDEs may differ from those drawn for the finite-difference equations, they are usually the same; in fact, Morse and Feshbach [69] use the finite difference form to derive the relations between boundary conditions and uniqueness for each type of equation shown in Tab. 23.1 [70].)

Solving PDEs numerically differs from solving ODEs in a number of ways. First, because we are able to write all ODEs in a standard form,

$$\frac{d\mathbf{y}(t)}{dt} = \mathbf{f}(\mathbf{y}, t) \quad (23.2)$$

with  $t$  the single independent variable, we are able to use a standard algorithm, rk4 in our case, to solve all such equations. Yet because PDEs have



**Fig. 23.1** *Left:* The region of space within a square in which we want to determine the electric potential. There is a wire at the top kept at a constant 100 V and a grounded wire at the sides and bottom. *Right:* The electric potential as a function of  $x$  and  $y$ . The projections onto the  $xy$  plane are equipotential surfaces or lines.

several independent variables, to illustrate,  $\rho(x, y, z, t)$ , we would have to apply (23.2) simultaneously and independently to each variable, which would be very complicated. Second, since there are more equations to solve with PDEs than ODEs, we need more information than just the two *initial conditions* [ $x(0)$ ,  $v(0)$ ]. Yet because each PDE often has its own particular set of boundary conditions, we have to develop a special algorithm for each particular problem.

## 23.2 Electrostatic Potentials

Your **problem** is to find the electric potential for all points *inside* the charge-free square shown in Fig. 23.1. The bottom and sides of the region are made up of wires that are “grounded” (kept at 0 V). The top wire is connected to a battery that keeps it at a constant 100 V.

### 23.2.1 Laplace's Elliptic PDE (Theory)

We consider the entire square in Fig. 23.1 as our boundary with voltages prescribed upon it. If we imagine infinitesimal insulators placed at the top corners of the box, then we have a closed boundary within which we will solve our problem. Since the values of the potential are given on all sides, we have Neumann conditions on the boundary, and, according to Tab. 23.1, a unique and stable solution.

It is known from classical electrodynamics, that the electric potential  $U(\mathbf{x})$  arising from static charges satisfies Poisson's PDE [70]:

$$\nabla^2 U(\mathbf{x}) = -4\pi\rho(\mathbf{x}) \quad (23.3)$$

where  $\rho(\mathbf{x})$  is the charge density. In charge-free regions of space, that is, regions where  $\rho(\mathbf{x}) = 0$ , the potential satisfies *Laplace's equation*:

$$\nabla^2 U(\mathbf{x}) = 0. \quad (23.4)$$

Both these equations are elliptic PDEs of a form that occurs in various applications. We solve them in 2D rectangular coordinates:

$$\frac{\partial^2 U(x, y)}{\partial x^2} + \frac{\partial^2 U(x, y)}{\partial y^2} = \begin{cases} 0 & \text{Laplace's equation} \\ -4\pi\rho(x) & \text{Poisson's equation} \end{cases} \quad (23.5)$$

In both cases, we see that the potential depends simultaneously on  $x$  and  $y$ . For Laplace's equation, the charges, which are source of the field, enter indirectly by specifying the potential values in some region of space; for Poisson's equation they enter directly.

### 23.3

#### Fourier Series Solution of PDE

For the simple geometry of Fig. 23.1, it is possible to find an analytic solution in the form of an infinite series. However, we will see that this is often not a good approach if numerical values are needed because the series may converge painfully slowly and may contain spurious oscillations. We start with Laplace's equation in Cartesian form,

$$\frac{\partial^2 U(x, y)}{\partial x^2} + \frac{\partial^2 U(x, y)}{\partial y^2} = 0 \quad (23.6)$$

with the boundary conditions given along a square of side  $L$ . If we assume that the potential is the product of independent functions of  $x$  and  $y$ , and substitute this product into (23.6), we obtain

$$U(x, y) = X(x)Y(y) \quad \Rightarrow \quad \frac{d^2X(x)/dx^2}{X(x)} + \frac{d^2Y(y)/dy^2}{Y(y)} = 0 \quad (23.7)$$

Because  $X(x)$  is a function of only  $x$  and  $Y(y)$  of only  $y$ , the derivatives in (23.7) are *ordinary* as opposed to *partial* derivatives. Since  $X(x)$  and  $Y(y)$  are assumed to be independent, the only way (23.7) can be valid for *all* values of

$x$  and  $y$  is for each term in (23.7) to be equal to a constant:

$$\frac{d^2Y(y)/dy^2}{Y(y)} = -\frac{d^2X(x)/dx^2}{X(x)} = k^2 \quad (23.8)$$

$$\Rightarrow \frac{d^2X(x)}{dx^2} + k^2 X(x) = 0 \quad \frac{d^2Y(y)}{dy^2} - k^2 Y(y) = 0 \quad (23.9)$$

We shall see that this choice of sign for the constant matches the boundary conditions and gives us periodic behavior in  $x$ . The other choice of sign would give periodic behavior in  $y$ , and that would not work.

It is worth pointing out here that even though an analytic solution in the form of the product of separate functions of  $x$  and  $y$  exists to Laplace's equation, this does not mean that the solution to a realistic problem will have this form. Indeed, we shall see that a realistic solution requires an infinite sum of such products, which is no longer separable into functions of  $x$  and  $y$ , and is, accordingly, not a good approach if numerical values of the potential are needed.

The solutions for  $X(x)$  are periodic and those for  $Y(y)$  are exponential:

$$X(x) = A \sin kx + B \cos kx \quad Y(y) = Ce^{ky} + De^{-ky} \quad (23.10)$$

The  $x = 0$  boundary condition  $U(x = 0, y) = 0$  can be met only if  $B = 0$ . The  $x = L$  boundary condition  $U(x = L, y) = 0$  can be met only for

$$kL = n\pi \quad n = 1, 2, \dots \quad (23.11)$$

Such being the case, for each value of  $n$  there is the solution

$$X_n(x) = A_n \sin\left(\frac{n\pi}{L}x\right) \quad (23.12)$$

For each value of  $k_n$  that satisfies the  $x$  boundary conditions,  $Y(y)$  must satisfy the  $y$  boundary condition  $U(x, 0) = 0$ , which requires  $D = -C$ :

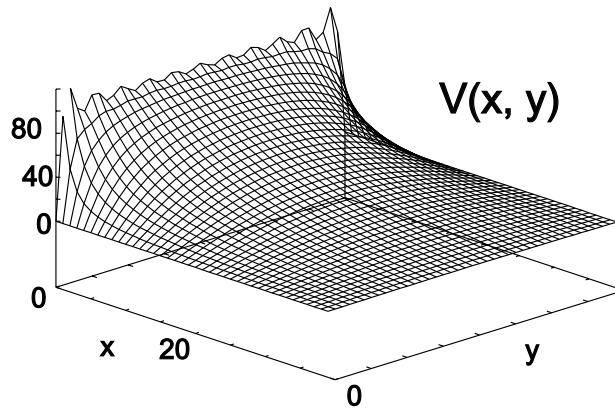
$$Y_n(y) = C(e^{k_n y} - e^{-k_n y}) \equiv 2C \sinh\left(\frac{n\pi}{L}y\right) \quad (23.13)$$

Because we are solving linear equations, the principle of linear superposition holds, which means that the most general solution is the sum of the products:

$$U(x, y) = \sum_{n=1}^{\infty} E_n \sin\left(\frac{n\pi}{L}x\right) \sinh\left(\frac{n\pi}{L}y\right) \quad (23.14)$$

The  $E_n$  values are arbitrary constants and are fixed by requiring the solution to satisfy the remaining boundary condition at  $y = L$ ,  $U(x, y = L) = 100$  V:

$$\sum_{n=1}^{\infty} E_n \sin\left(\frac{n\pi}{L}x\right) \sinh(n\pi) = 100 \text{ V} \quad (23.15)$$



**Fig. 23.2** The analytic (Fourier series) solution of Laplace's equation showing Gibbs's overshoot oscillations near  $y = 0$ . The solution shown here uses 21 terms, yet the oscillations remains even if a large number of terms is summed.

We determine the constants  $E_n$  by projection: multiply both sides of the equation by  $\sin m\pi/Lx$ , with  $m$  an integer, and integrate from 0 to  $L$ :

$$\sum_n E_n \sinh n\pi \int_0^L dx \sin \frac{n\pi}{L} x \sin \frac{m\pi}{L} x = \int_0^L dx 100 \sin \frac{m\pi}{L} x \quad (23.16)$$

The integral on the LHS is nonzero only for  $n = m$ , which yields

$$E_n = \begin{cases} 0 & \text{for } n \text{ even} \\ \frac{4(100)}{n\pi \sinh n\pi} & \text{for } n \text{ odd} \end{cases} \quad (23.17)$$

Finally, we obtain the potential at any point  $(x, y)$  as

$$U(x, y) = \sum_{n=1,3,5,\dots}^{\infty} \frac{400}{n\pi} \sin \left( \frac{n\pi x}{L} \right) \frac{\sinh(n\pi y/L)}{\sinh(n\pi)} \quad (23.18)$$

### 23.3.1

#### Shortcomings of Polynomial Expansions

It is interesting to observe that the solution via the numerical algorithm (23.25) starts with the values of the potential on the boundaries and then propagates them through all space via repeated iterations. In contrast, the “exact” solution must sum an infinite number of terms to be exact, which of course is never possible in a practical calculation, and so in practice it is too a numerical approximation—but not a good one! First, the oscillatory nature of the terms being summed leads to cancellation s, and so the sum of many terms suffers from the roundoff error. Second, the sinh functions in (23.18) overflows for large  $n$ , which can be avoided somewhat by expressing the quotient

of the two sinh functions in terms of exponentials, and then taking a large  $n$  limit:

$$\frac{\sinh(n\pi y/L)}{\sinh(n\pi)} = \frac{e^{n\pi(y/L-1)} - e^{-n\pi(y/L+1)}}{1 - e^{-2n\pi}} \rightarrow e^{n\pi(y/L-1)} \quad (23.19)$$

A third problem with the “analytic” solution is that a Fourier series converges only in the *mean square* (Fig. 23.2). This means that it converges to the *average* of the left- and right-hand limits in the regions where the solution is discontinuous [71], such as in the corners of the box. Explicitly, what you see in Fig. 23.2 is a phenomenon known as the **Gibb’s overshoot** that occurs when a Fourier series with a finite number of terms is used to represent a discontinuous function. Rather than fall off abruptly, the series develops large oscillations that tend to overshoot the function at the corner. To obtain a smooth solution, we had to sum 40,000 terms.

## 23.4

### Solution: Finite Difference Method

To solve our 2D PDE, we divide space up into a lattice (Fig. 23.3) and solve for  $U$  at each site on the lattice. Since we will express derivatives in terms of the finite differences in the values of  $U$  at the lattice sites, this is called a *finite difference* method. A numerically more efficient, but also more complicated approach, is the *finite-element* method (Unit II), which solves the PDE for small geometric elements, and then matches the elements.

To derive the finite-difference algorithm for the numeric solution of (23.5), we follow the same path taken in Section 6.1 to derive the forward-difference algorithm for differentiation. We start by adding the two Taylor expansions of the potential to the right and left of  $(x, y)$ :

$$U(x + \Delta x, y) = U(x, y) + \frac{\partial U}{\partial x} \Delta x + \frac{1}{2} \frac{\partial^2 U}{\partial x^2} (\Delta x)^2 + \dots \quad (23.20)$$

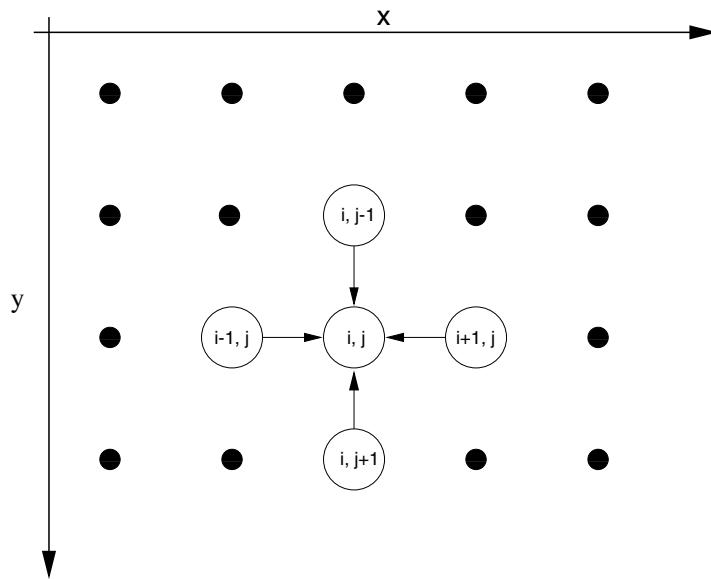
$$U(x - \Delta x, y) = U(x, y) - \frac{\partial U}{\partial x} \Delta x + \frac{1}{2} \frac{\partial^2 U}{\partial x^2} (\Delta x)^2 - \dots \quad (23.21)$$

All odd terms cancel when we add these equations together, and we obtain a central-difference approximation for the second partial derivative good to order  $(\Delta x)^4$ :

$$\Rightarrow \frac{\partial^2 U(x, y)}{\partial x^2} \simeq \frac{U(x + \Delta x, y) + U(x - \Delta x, y) - 2U(x, y)}{(\Delta x)^2} + \mathcal{O}(\Delta x^4)$$

Likewise, we add the two Taylor expansions of the potential above and below  $(x, y)$  to obtain

$$\frac{\partial^2 U(x, y)}{\partial y^2} \simeq \frac{U(x, y + \Delta y) + U(x, y - \Delta y) - 2U(x, y)}{(\Delta y)^2} + \mathcal{O}(\Delta y^4) \quad (23.22)$$



**Fig. 23.3** The algorithm for Laplace's equation in which the potential at the point  $(x, y) = (i, j)\Delta$  equals the average of the potential values at the four nearest-neighbor points. The nodes with white centers correspond to fixed values of the potential along the boundaries.

Substituting both these approximations into Poisson's PDE (23.5) leads to a finite-difference form of the equation:

$$\frac{U(x + \Delta x, y) + U(x - \Delta x, y) - 2U(x, y)}{(\Delta x)^2} + \frac{U(x, y + \Delta y) + U(x, y - \Delta y) - 2U(x, y)}{(\Delta y)^2} = -4\pi\rho \quad (23.23)$$

If the  $x$  and  $y$  grids are of equal spacings, this takes the simple form

$$U(x + \Delta x, y) + U(x - \Delta x, y) + U(x, y + \Delta y) + U(x, y - \Delta y) - 4U(x, y) = -4\pi\rho \quad (23.24)$$

The reader will notice that this equation is a relation among the solutions at five points in space. When  $U(x, y)$  is evaluated for the  $N_x$   $x$  values on the lattice, and for the  $N_y$   $y$  values, there results a set of  $N_x \times N_y$  simultaneous linear algebraic equations for  $U[i][j]$ . One approach is to solve these linear equations explicitly as a (big) matrix problem, using the computer to do the matrix algebra. This is attractive as it is a direct solution, but it requires a great deal of memory and book keeping.

The approach we follow here is a simple one. We solve (23.24) for  $U(x, y)$ :

$$\begin{aligned} U(x, y) \simeq & \frac{1}{4} [U(x + \Delta, y) + U(x - \Delta, y) + U(x, y + \Delta) + U(x, y - \Delta)] \\ & + \pi\rho(x, y)\Delta^2 \end{aligned} \quad (23.25)$$

where we would just leave off the  $\rho(x)$  term for Laplace's equation. In terms of discrete locations on our lattice, the  $x$  and  $y$  variables are:

$$x = x_0 + i\Delta, \quad y = y_0 + j\Delta \quad i, j = 0, \dots, N_{\max-1} \quad (23.26)$$

$$\Delta x = \Delta y = \Delta = L/(N_{\max} - 1) \quad (23.27)$$

where we have placed our lattice in a square of side  $L$ . The potential is represented by the array  $U[N_{\max}][N_{\max}]$ , and the finite-difference algorithm (23.25) becomes

$$U_{i,j} = \frac{1}{4} [U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}] + \pi\rho(i\Delta, j\Delta)\Delta^2 \quad (23.28)$$

This equation says that when we have a proper solution it will be the average of the potential at the four nearest neighbors (Fig. 23.3), plus a contribution from the local charge density. As an algorithm, (23.28) does not provide a direct solution to Poisson's equation, but rather must be repeated many times to converge upon the solution. We start with an initial guess for the potential, improve it by sweeping through all space taking the average over nearest neighbors at each node, and keep repeating the process until the solution no longer changes to some level of precision, or until failure is evident. When converged, the initial guess is said to have *relaxed* into the solution.

A reasonable question with this simple approach is "Does it always converge, and if so, does it converge fast enough to be useful?" In some sense the answer to the first question is not an issue; if the method does not converge then we will know it, else, we have ended up with a solution and the path we followed to get there does not matter! The answer to the question of speed is that relaxation methods may converge slowly (although still faster than a Fourier series), yet we will show you two clever tricks to accelerate the convergence.

At this point it is important to remember that our algorithm arose from expressing the Laplacian  $\nabla^2$  in rectangular coordinates. While this does not restrict us from solving problems with circular symmetry, there may be geometries where it is better to develop an algorithm based on expressing the Laplacian in the cylindrical or spherical coordinates in order to have grids that fit the geometry better.

### 23.4.1

#### **Relaxation and Over-Relaxation**

There are a number of variations in how to iterate the algorithm (23.25), and so continually convert the initial boundary conditions into a solution. Its most

basic form is the **Jacobi method**, and is one in which the potential values are not updated until an entire sweep of applying (23.25) at each point is completed. This maintains the symmetry of the initial guess and boundary conditions.

A rather obvious improvement of the Jacobi method employs the updated guesses for the potential in (23.25) as soon as they are available. As a case in point, if the sweep starts in the upper left-hand corner of Fig. 23.3, then the left-most ( $i-1, j$ ) and top-most ( $i, j-1$ ) values of the potential used will be from the present generation of guesses, while the other two values of the potential will be from the previous generation:

$$U_{i,j}^{(\text{new})} = \frac{1}{4} \left[ U_{i+1,j}^{(\text{old})} + U_{i-1,j}^{(\text{new})} + U_{i,j+1}^{(\text{old})} + U_{i,j-1}^{(\text{new})} \right] \quad (\text{GS}) \quad (23.29)$$

This technique, known as the **Gauss–Seidel method**, usually leads to accelerated convergence, which in turn leads to less roundoff error. It also uses less memory as there is no need to store two generations of guesses. However, it does distort the symmetry of the boundary conditions or the initial guess, which one hopes is insignificant when convergence is reached.

A less obvious improvement to the relaxation technique, known as **successive over-relaxation** (SOR), starts by writing the algorithm (23.25) in a form that determines the new values of the potential  $U^{(\text{new})}$  as the old values  $U^{(\text{old})}$  plus a correction or residual  $r$ :

$$U_{i,j}^{(\text{new})} = U_{i,j}^{(\text{old})} + r_{i,j} \quad (23.30)$$

While the Gauss–Seidel technique may still be used to incorporate the updated values in  $U^{(\text{old})}$  to determine  $r$ , we just rewrite the algorithm in the general form:

$$\begin{aligned} r_{i,j} &\equiv U_{i,j}^{(\text{new})} - U_{i,j}^{(\text{old})} \\ &= \frac{1}{4} \left[ U_{i+1,j}^{(\text{old})} + U_{i-1,j}^{(\text{new})} + U_{i,j+1}^{(\text{old})} + U_{i,j-1}^{(\text{new})} \right] - U_{i,j}^{(\text{old})} \end{aligned} \quad (23.31)$$

The successive over-relaxation technique [9, 72] proposes that if convergence is obtained by adding  $r$  to  $U$ , then even more rapid convergence might be obtained by adding more of  $r$ :

$$U_{i,j}^{(\text{new})} = U_{i,j}^{(\text{old})} + \omega r_{i,j} \quad (\text{SOR}) \quad (23.32)$$

where  $\omega$  is a parameter that amplifies, or reduces, the effect of the residual. The nonaccelerated relaxation algorithm (23.29) is obtained with  $\omega = 1$ , accelerated convergence (over relaxation) is obtained with  $\omega \geq 1$ , and under relaxation occurs for  $\omega < 1$ . Values of  $\omega \leq 2$  often work well, yet  $\omega > 2$  may lead to numerical instabilities. Although a detailed analysis of the algorithm

is needed to predict the optimal value for  $\omega$ , we suggest that you explore different values for  $\omega$  to see which one works best for your particular problem.

**Listing 23.1:** `LaplaceLine.java` is the framework for the solution of Laplace's equation via relaxation. The various parameters need to be adjusted for a good and realistic solution.

```
/* LaplaceLine.java: Laplace eqn via finite difference mthd
   wire in grounded box, Output for 3D gnuplot */

import java.io.*;

public class LaplaceLine {
    static int Nmax = 100;                                // Size of box

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {

        double V[][] = new double[Nmax][Nmax];
        int i, j, iter;

        PrintWriter w = new PrintWriter           // Save data in file
            (new FileOutputStream("LaplaceLine.dat"), true);
            // Initialize

        for (i=0; i<Nmax; i++)
            { for (j=0; j<Nmax; j++) V[i][j] = 0.; }
        for (i=0; i < Nmax; i++) V[i][0] = 100.;          // V(wire) = 100 V
                                                        // Iterations
        for (iter=0; iter < 1000; iter++) {
            // x, then y directions
            for (i=1; i < (Nmax-1); i++) {
                for (j=1; j < (Nmax-1); j++)           // THE ALGORITHM
                    V[i][j] = (V[i+1][j]+V[i-1][j]+V[i][j+1]+V[i][j-1])/4.;
            }
            for (i=0; i < Nmax; i=i+2) {           // Data in gnuplot format
                for (j=0; j < Nmax; j=j+2) w.println(" " + V[i][j] + " ");
                w.println("");                      // Blank line separates rows
            }
            System.out.println("data stored in LaplaceLine.dat");
        }
    }
}
```

### 23.4.2

#### Lattice PDE Implementation

In Listing 23.1 we present the code `LaplaceLine.java` that solves the square-wire problem (Fig. 23.1). Here we have kept the code simple by setting the length of the box  $L = N_{\text{max}}\Delta = 100$ , taking  $\Delta = 1$ :

$$\begin{array}{lll} U(i, N_{\text{max}}) & = & 99 \quad (\text{top}) \\ U(N_{\text{max}}, j) & = & 0 \quad (\text{right}) \end{array} \quad \begin{array}{lll} U(1, j) & = & 0 \quad (\text{left}) \\ U(i, 1) & = & 0 \quad (\text{bottom}) \end{array} \quad (23.33)$$

We also run the algorithm (23.28) for a fixed number, 1000 iterations. A better code would vary  $\Delta$  and the dimensions, and would quit iterating once the solution converges. Study, compile, and execute the basic code.

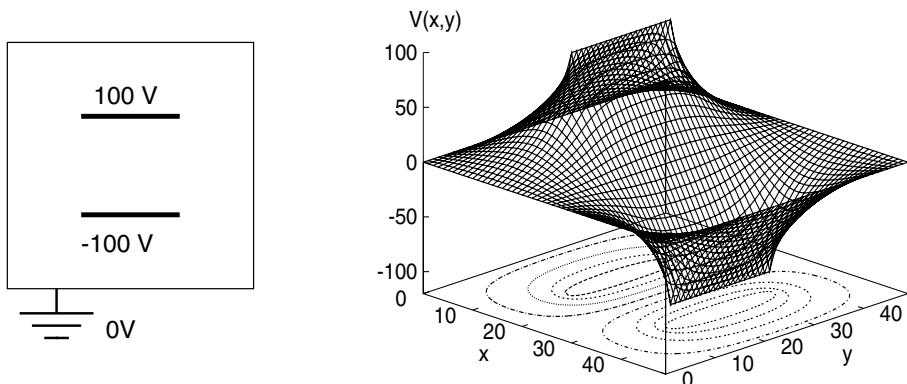
### 23.5

#### Assessment via Surface Plot

After executing `LaplaceLine.java`, you should have a file with data in the format appropriate for a surface plot like Fig. 23.1. Seeing that it is important to visualize your output to ensure the reasonableness of the solution, you should learn how to make such a plot before exploring more interesting problems. The 3D surface plots we show in this chapter were made with *gnuplot*. Here we repeat the commands used for Gnuplot with the output file of `LaplaceLine.java`:

> <b>gnuplot</b>	Start gnuplot system from a shell
gnuplot> <b>set hidden3d</b>	Hide surface whose view is blocked
gnuplot> <b>set unhidden3d</b>	Show surface though hidden from view
gnuplot> <b>splot 'Laplace.dat' with lines</b>	Surface plot of Laplace.dat with lines
gnuplot> <b>set view 65, 45</b>	Set $x$ and $y$ rotation viewing angles
gnuplot> <b>replot</b>	See effect of your change
gnuplot> <b>set contour</b>	Project contours onto the $x$ - $y$ plane
gnuplot> <b>set cntrparam levels 10</b>	10 contour levels
gnuplot> <b>set terminal PostScript</b>	Output in PostScript format for printing
gnuplot> <b>set output "Laplace.ps"</b>	Output to file <code>Laplace.ps</code>
gnuplot> <b>splot 'Laplace.dat' w l</b>	Plot again, output to file
gnuplot> <b>set terminal x11</b>	To see output on screen again
gnuplot> <b>set title 'Potential V(x,y) vs x,y'</b>	Title graph
gnuplot> <b>set xlabel 'x Position'</b>	Label $x$ axis
gnuplot> <b>set ylabel 'y Position'</b>	Label $y$ axis
gnuplot> <b>set zlabel 'V(x,y)'; replot</b>	Label $z$ axis and replot
gnuplot> <b>help</b>	Tell me more
gnuplot> <b>set nosurface</b>	Do not draw surface, leave contours
gnuplot> <b>set view 0, 0, 1</b>	Look down directly onto base
gnuplot> <b>replot</b>	Draw plot again; may want to write to file
gnuplot> <b>quit</b>	Get out of gnuplot

Here we have explicitly stated the viewing angle for the surface plot. Because gnuplot 4 permits you to rotate surface plots interactively, we recommend that you do just that to find the best viewing angle. Changes made to a plot are seen when you redraw the plot using the `replot` command. For this sample session, the default output for your graph is your terminal screen. To print a paper copy of your graph we recommend first saving it to a file as a *PostScript*



**Fig. 23.4** *Left:* A simple model of a parallel-plate capacitor (or of a vacuum-tube diode). A realistic model would have the plates closer together, in order to condense the field, and the enclosing, grounded box so large that it has no effect on the field near the capacitor. *Right:* A numerical solution for the electric potential for this geometry. The projection on the  $xy$  plane gives the equipotential lines.

document (suffix `.ps`), and then printing out that file to a PostScript printer. You create the PostScript file by changing the terminal type to `Postscript`, setting the name of the file, and then issuing the subcommand `splot` again. This plots the result out to a file. If you want to see plots on your screen again, you need to set the terminal type back to `x11` again (for Unix's *X Windows System*), and then plot it again.

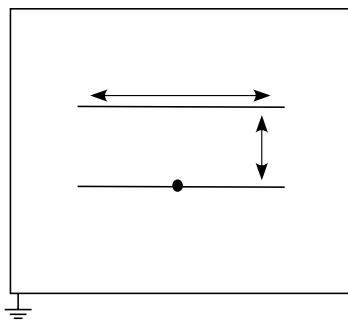
## 23.6

### Three Alternate Capacitor Problems

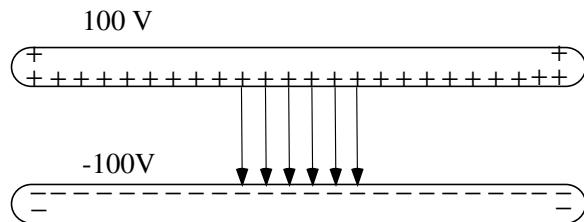
*We give you (or your instructor) a choice now. You can carry out the assessment using our wire-plus-grounded box problem, or you can replace that problem with the more interesting one of a realistic capacitor. We now describe the capacitor problem, and then get on to the assessment and exploration.*

The standard solution for a capacitor's field is for the region between two infinite plates, where the field is uniform and completely confined. The field in a finite capacitor will vary near the edges ("edge effects"), and extend beyond the edges of the capacitor ("fringe fields"). We model the realistic capacitor in a grounded box (Fig. 23.4) as two plates (wires) of finite length. Write your program such that it is convenient to vary the grid spacing  $\Delta$  and the geometry of the box and plate, without having to get into the details of the program. We pose three versions of this problem, each displaying some different physics. In each case the boundary conditions  $V = 0$  along the box must be satisfied for all steps during the solution process.

1. For the simplest version, assume that the plates are very thin sheets of conductors, with the top plate maintained at 100 V and the bottom at -100 V. Because the plates are conductors, they must be equipotential surfaces, and a battery can maintain them at constant voltages. Write or modify the given program to solve Laplace's equation such that the plates have fixed voltages.
2. For the next version of this problem, assume that the plates are made of a thin dielectric material with uniform charge densities  $\rho$  on the top and  $-\rho$  on the bottom. Solve Poisson's equation (23.3) in the region including the plates, and Laplace's equation elsewhere. Experiment until you find a numerical value for  $\rho$  that gives a similar potential to that shown in Fig. 23.4 for plates with fixed voltages.
3. For the final version of this problem we would like to investigate how the charges on the conducting plates of the capacitor distributes themselves. To do that, assume that the plates have a finite thickness (Fig. 23.6). Since the plates are conductors, we can still assume that they are equipotential surfaces at 100 and -100 V, only now we want them to have a thickness of at least  $2\Delta$  (so we can see the difference between the potential near the top and the bottom of the plates). Such being the case, we solve Laplace's equation (23.4) much as before to determine  $U(x, y)$ . Once we have  $U(x, y)$ , substitute it into Poisson's equation (23.3) and determine how the charge density distributes itself along the top and bottom surfaces of the plates. *Hint:* Since the electric field is no longer uniform, we know that the charge distribution also will no longer be uniform. In addition, since the electric field now extends beyond the ends of the capacitor, and since field lines begin and end on charge, we suspect that some charge may end up on the edges and outer surfaces of the plates (Fig. 23.5).



**Fig. 23.5** A suggested scheme for labeling the geometry of a parallel-plate capacitor within a box.



**Fig. 23.6** An indication of how the electric charge may rearrange itself on a capacitor with conducting plates.

### 23.7 Implementation and Assessment

1. Write a program or modify the one on the CD to find the electric potential for a capacitor within a grounded box. (In Listing 23.2 we list our version of successive over relaxation.) Use the labeling scheme in Fig. 23.5 and make your program general enough so that you can increase the size of the box, or the geometry of the capacitor, without having to rewrite the program.
2. Our sample program undertakes 1000 iterations and quits. You probably want to stick with this assumption until you have your program working. At least during debugging, examine how the potential changes in some key locations as you iterate toward a solution.
3. Repeat the process for different step sizes  $\Delta$  and judge if the process is stable and convergent for all sizes.
4. Once your program produces a reasonable graph, modify the program so it stops iterating after convergence is reached (or after a very large number of iterations, in case there is no convergence). Rather than try to discern small changes in highly compressed surface plots, use a numerical measure of precision. While it might be best to test for convergence throughout the entire box, this could require comparing millions of values, most of which are so close to zero that their precision does not matter anyway. Instead, look at the surface plot of the potential in Fig. 23.4, and observe how the line along the diagonal samples the whole range of the potential:

$$\text{trace} = \sum_i |\mathbf{v}[i][i]| \quad (23.34)$$

Print out the value of `trace` for each iteration and note how much precision is obtained. You may well need hundreds or thousands of iterations to obtain stable and accurate answers; this is part of the price paid for simplicity of algorithm.

5. Once you know the value of `trace` corresponding to the best precision you can obtain, incorporate it in an automated test of convergence. To illustrate, define `tol = 0.0001` as the desired relative precision and modify `LaplaceLine.java` so that it stops iterating when the relative change from one iteration to the next is less than `tol`:

$$\left| \frac{\text{trace} - \text{traceOld}}{\text{traceOld}} \right| < \text{tol} \quad (23.35)$$

The `break` command or a `while` loop is useful for this type of iteration.

6. Equation (23.32) expresses the successive over-relaxation technique in which convergence is accelerated by using a judicious choice of  $\omega$ . Determine by trial and error the approximate best value of  $\omega$ . Often  $\omega \leq 2$  speeds up the convergence by a factor of approximately two.
7. Now that you know the code is accurate, modify it to simulate a more realistic capacitor in which the plate separation is approximately 1/10th of the plate length. You should find the field more condensed and more uniform between the plates.
8. If you are working with the wire-in-the-box problem, compare your numerical solution to the analytic one (23.18). Do not be surprised if you need to sum thousands of terms before the “analytic” solution converges!

**Listing 23.2:** `LaplaceSOR.java` solves Laplace’s equation with successive over relaxation.

```
// LaplaceSOR.java: Solves Laplace eqn using SOR for convergence

import java.io.*;

public class LaplaceSOR {
    static int max = 40; // Number of grid points

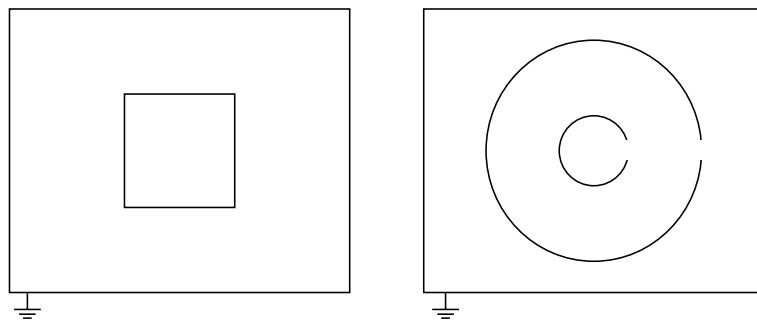
    public static void main(String[] argv) throws IOException,
                                               FileNotFoundException {
        double x, tol, aux, omega, r, pi = 3.141592653;
        double p[][] = new double[max][max];
        int i, j, iter;

        PrintWriter w = new PrintWriter(
            new FileOutputStream("laplR.dat"), true);
        omega = 1.8; // SOR parameter
        long timeStart=System.nanoTime();
        System.out.println(""+timeStart+"");
        for (i=0; i<max; i++) for (j=0; j<max; j++) p[i][j] = 0.; // Init
    }
}
```

```

for ( i = 0; i<max; i++) p[i][0] = +100.0; // Tolerance
tol = 1.0;
iter = 0; // Iterations
while ( ( tol > 0.000001) && (iter <= 140) ) {
    tol = 0.0; // x, then y directions
    for (i = 1; i<(max-1); i++) {
        for (j = 1; j<(max-1); j++) { // SOR ALGORITHM
            r = omega * ( p[i][j+1] + p[i][j-1] + p[i+1][j] +
                           p[i-1][j] - 4.0 * p[i][j] ) / 4.0;
            p[i][j] += r;
            if ( Math.abs(r) > tol ) tol = Math.abs(r);
        }
        iter++;
    }
}
long timeFinal=System.nanoTime();
System.out.println(""+timeFinal+"");
System.out.println("Nanoseconds="+ (timeFinal-timeStart));
for (i = 0; i<max ; i++) {
    for (j = 0; j<max; j++) w.println(""+p[i][j]+");
    w.println(""); // Empty line for gnuplot
}
System.out.println("data stored in laplR.dat");
}
}

```



**Fig. 23.7** *Left:* The geometry of a capacitor formed by placing two long, square cylinders within each other. *Right:* The geometry of a capacitor formed by placing two long, circular cylinders within each other. The cylinders are cracked on the side so that wires can enter the region.

**23.8****Other Geometries and Boundary Conditions (Exploration)**

The numerical solution to our PDE can be applied to arbitrary boundary conditions. Two boundary conditions to explore are triangular and sinusoidal:

$$U(x) = \begin{cases} 200\frac{x}{w} & \text{for } x \leq w/2 \\ 100(1 - \frac{x}{w}) & \text{for } x \geq w/2 \end{cases} \quad U(x) = 100 \sin\left(\frac{2\pi x}{w}\right)$$

**Square Conductors:** You have designed a piece of equipment consisting of a small metal box at 100 V within a larger, grounded one (Fig. 23.7). You find that sparking occurs between the boxes, which means that the electric field is too large. You need to determine where the field is greatest so that you can change the geometry and eliminate the sparking.

Modify the program to satisfy these boundary conditions and to determine the field between the boxes. Gauss's law tells us that the electric field vanishes within the inner box because it contains no charge. Plot the potential and equipotential surfaces, and sketch in the electric field lines. Deduce where the electric field is most intense and try redesigning the equipment to reduce the field.

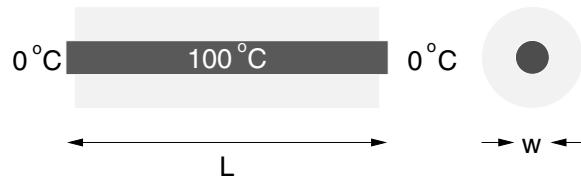
**Cracked Cylindrical Capacitor:** You have designed the cylindrical capacitor containing a long outer cylinder surrounding a thin inner cylinder (Fig. 23.7 right). The cylinders have a small crack in them in order to connect them to the battery that maintains the inner cylinder at  $-100$  V and outer cylinder at  $100$  V. Determine how this small crack affects the field configuration. In order for a unique solution to exist for this problem, place both cylinders within a large grounded box. Note, since our algorithm is based on the expansion of the Laplacian in rectangular coordinates, you cannot just convert it into a radial and angle grid.

**Thinking Outside the Box** : Find the electric potential for all points *outside* the charge-free square shown in Fig. 23.1. Is your solution unique?

## 24

### Heat Flow

**Problem:** You are given an aluminum bar of length  $L = 1 \text{ m}$  and width  $w$  aligned along the  $x$  axis (Fig. 24.1). It is insulated along its length but not its ends. Initially the bar is at a uniform temperature of  $100^\circ\text{C}$ , and then both ends are placed in contact with ice water at  $0^\circ\text{C}$ . Heat flows only out of the noninsulated ends. Your **problem** is to determine how the temperature will vary as we move along the length of the bar at later times.



**Fig. 24.1** A metallic bar insulated along its length with its ends kept at  $0^\circ\text{C}$ .

#### 24.1 The Parabolic Heat Equation (Theory)

A basic fact of nature is that heat flows from hot to cold, that is, from regions of high temperature to regions of low temperature. We give these words mathematical expression by stating that the rate of heat flow  $\mathbf{H}$  through a material is proportional to the gradient of the temperature  $T$  within that material:

$$\mathbf{H} = -K \nabla T(\mathbf{x}, t) \quad (24.1)$$

where  $K$  is the thermal conductivity of the material. The total amount of heat  $Q(t)$  in the material at any one time is proportional to the integral of the temperature over the volume of the material:

$$Q(t) = \int d\mathbf{x} C \rho(\mathbf{x}) T(\mathbf{x}, t) \quad (24.2)$$

where  $C$  is the specific heat of the material and  $\rho$  its density. Because energy is conserved, the rate of decrease of  $Q$  with time must equal the amount of

heat flowing out of the material. When this energy balance is struck, and the divergence theorem applied, the *heat equation* results:

$$\frac{\partial T(\mathbf{x}, t)}{\partial t} = \frac{K}{C\rho} \nabla^2 T(\mathbf{x}, t) \quad (24.3)$$

The heat equation (24.3) is a parabolic PDE with space and time as independent variables. The specification of this problem implies that there is no temperature variation in directions perpendicular to the bar ( $y$  and  $z$ ), and so we have only one spatial coordinate in our PDE:

$$\frac{\partial T(x, t)}{\partial t} = \frac{K}{C\rho} \frac{\partial^2 T(x, t)}{\partial x^2} \quad (24.4)$$

We are given the initial temperature of the bar, and the boundary conditions:

$$T(x, t = 0) = 100^\circ\text{C} \quad T(x = 0, t) = T(x = L, t) = 0^\circ\text{C} \quad (24.5)$$

## 24.2

### Solution: Analytic Expansion

As with Laplace's equation, the analytic "solution" starts with the assumption that the solution is the product of functions of space and time:

$$T(x, t) = X(x)\mathcal{T}(t) \quad (24.6)$$

When (24.6) is substituted into the heat equation (24.4), and the resulting equation is divided by the  $X(x)\mathcal{T}(t)$ , there results two noncoupled ODEs:

$$\frac{d^2X(x)}{dx^2} + k^2 X(x) = 0 \quad \frac{d\mathcal{T}(t)}{dt} + k^2 \frac{C}{C\rho} \mathcal{T}(t) = 0 \quad (24.7)$$

where  $k$  is a constant to be determined. The boundary condition that the temperature equals zero at  $x = 0$  requires a sine function for  $X$ :

$$X(x) = A \sin kx \quad (24.8)$$

The boundary condition that the temperature equals zero at  $x = L$  requires the sine function to vanish there:

$$\sin kL = 0 \Rightarrow k = k_n = n\pi/L \quad n = 1, 2, \dots \quad (24.9)$$

The time function is a decaying exponential with  $k_n$  in the exponent:

$$\mathcal{T}(t) = e^{-k_n^2 t/C\rho} \Rightarrow T(x, t) = A_n \sin k_n x e^{-k_n^2 t/C\rho} \quad (24.10)$$

where  $n$  can be any integer, and  $A_n$  is an arbitrary constant.

Equation (24.10) is a particular solution. The most general solution is the linear superposition of all values of  $n$ :

$$T(x, t) = \sum_{n=1}^{\infty} A_n \sin k_n x e^{-k_n^2 t / C\rho} \quad (24.11)$$

The expansion coefficients  $A_n$  are determined by the initial condition that at time  $t = 0$  the entire bar has temperature  $T = 100^\circ\text{C}$ :

$$T(x, t = 0) = 100^\circ\text{C} \quad \Rightarrow \quad \sum_{n=1}^{\infty} A_n \sin k_n x = 100^\circ\text{C} \quad (24.12)$$

As with Laplace's equation, projecting out the sine functions determines  $A_n = 4T_0 / n\pi$  for  $n$  odd:

$$T(x, t) = \sum_{n=1,3,\dots}^{\infty} \frac{4T_0}{n\pi} \sin k_n x e^{-k_n^2 Kt / (C\rho)} \quad (24.13)$$

### 24.3

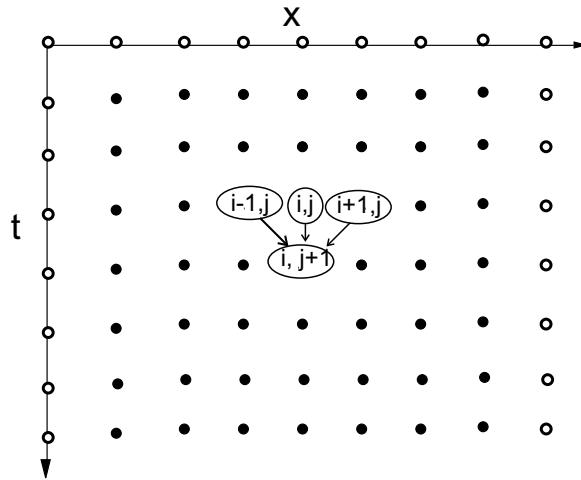
#### Solution: Finite Time Stepping (Leap Frog)

As we did with Laplace's equation, the numerical solution is based on converting the differential equation into a finite-difference ("difference") equation. We discretize space and time on a lattice (Fig. 24.2), and look for a solution along the nodes. The horizontal nodes with white centers correspond to the known values of the temperature for the initial time, while the vertical white nodes correspond to the fixed temperature along the boundaries. If we also knew the temperature for times along the bottom row, then we could use a relaxation algorithm, as we did for Laplace's equation. However, with only the top row known, we shall end up with an algorithm that steps forward in time, one row at a time, as in the children's game *leapfrog*.

The algorithm is customized for the equation being solved and for the constraints imposed by the particular set of initial and boundary conditions. With only one row of times to start with, we use a forward-difference approximation for the time derivative of the temperature:

$$\frac{\partial T(x, t)}{\partial t} \simeq \frac{T(x, t + \Delta t) - T(x, t)}{\Delta t} \quad (24.14)$$

Because we know the spatial variation of the temperature along the entire top row, as well as along the left and right sides, we are not as constrained with the space derivative as with the time derivative. Consequently, as we did with the



**Fig. 24.2** The algorithm for the heat equation in which the temperature at the location  $x = i\Delta x$  and time  $t = (j + 1)\Delta t$  is computed from the temperature values at three points of an earlier time. The nodes with white centers correspond to known initial and boundary conditions. (The boundaries are placed artificially close for illustrative purposes.)

Laplace equation, we use the more-accurate central-difference approximation for the (second) space derivative:

$$\frac{\partial^2 T(x, t)}{\partial x^2} \simeq \frac{T(x + \Delta x, t) + T(x - \Delta x, t) - 2T(x, t)}{(\Delta x)^2} \quad (24.15)$$

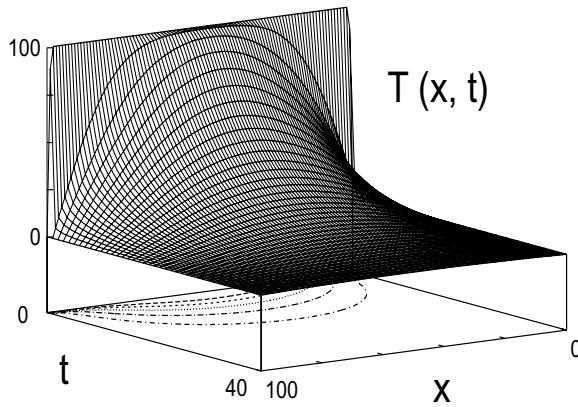
When we substitute these approximations for the derivatives into the heat equation (24.4), we obtain the heat difference equation:

$$\frac{T(x, t + \Delta t) - T(x, t)}{\Delta t} = \frac{K}{C\rho} \frac{T(x + \Delta x, t) + T(x - \Delta x, t) - 2T(x, t)}{\Delta x^2} \quad (24.16)$$

We reorder this equation to a form in which the solution can be stepped forward in time:

$$T_{i,j+1} = T_{i,j} + \eta [T_{i+1,j} + T_{i-1,j} - 2T_{i,j}] \quad \eta = \frac{K\Delta t}{C\rho\Delta x^2} \quad (24.17)$$

where  $x = i\Delta x$  and  $t = j\Delta t$ . This algorithm is called *explicit* because it provides a solution in terms of known values of the temperature. If we tried to solve for the temperature at all lattice sites simultaneously, then we would have an *implicit* algorithm that requires us to solve equations involving unknown values of the temperature (Fig. 24.2). We see that the temperature at space-time point  $(i, j + 1)$  is computed from the three temperature values at an earlier time  $j$ , and at adjacent space values  $i \pm 1, i$ . We start the solution at the top row, moving it forward in time for as long as we want, keeping the temperature along the ends fixed at 0°C. Fig. 24.3 shows the solution so obtained.



**Fig. 24.3** A numeric calculation of the temperature versus position and versus time, with isotherm contours projected onto horizontal plane.

## 24.4

### von Neumann Stability Assessment

When we solve a PDE by converting it to a difference equation, we hope that the solution to the difference equation is a good approximation to the solution of the PDE. If the difference-equation solution diverges, then we know we have a bad approximation. If the difference solution converges, then it generally provides a good approximation to the PDE. The *von Neumann stability analysis* is based on the assumption that eigenmodes of the difference equation can be written as

$$T_{m,j} = \xi(k)^j e^{ikm\Delta x} \quad (24.18)$$

where  $x = m\Delta x$ ,  $t = j\Delta t$ , but  $i = \sqrt{-1}$  is the imaginary number. The constant  $k$  is an unknown wave vector ( $2\pi/\lambda$ ), and  $\xi(k)$  is an unknown complex function of  $k$ . View (24.18) as a basis function that oscillates in space (the exponential), with an amplitude or *amplification factor*  $\xi(k)^j$  that changes by an additional power of  $\xi$  for each time step. If the general solution to the difference equation can be expanded in terms of these eigenmodes, then the general solution will be stable if the eigenmodes are stable. Clearly, for an eigenmode to be stable, the amplitude  $\xi$  cannot grow in time  $j$ , which means  $|\xi(k)| < 1$  for all values of the parameter  $k$  [9, 73].

Application of the stability analysis is more straightforward than it might seem. We substitute the expansion (24.18) into the difference equation (24.17):

$$\xi^{j+1} e^{ikm\Delta x} = \xi^j e^{ikm\Delta x} + \eta \left[ \xi^j e^{ik(m+1)\Delta x} + \xi^j e^{ik(m-1)\Delta x} - 2\xi^j e^{ikm\Delta x} \right]$$

After cancelling some common factors, it is easy to solve for  $\xi$ :

$$\xi(k) = 1 + 2\eta[\cos(k\Delta x) - 1] \quad (24.19)$$

In order for  $|\xi(k)| < 1$  for all possible  $k$  values, we must have

$$\eta = \frac{K\Delta t}{C\rho\Delta x^2} < \frac{1}{4} \quad (24.20)$$

Equation (24.20) tells us that if we make the time step  $\Delta t$  smaller, we always improve stability. But if we decrease the space step  $\Delta x$ , without a simultaneous quadratic *increase* in the time step, we worsen stability. The lack of space-time symmetry arises from our use of stepping in time, but not space.

In general, you should perform a stability analysis for every PDE you have to solve, although it can get complicated [9]. Yet even if you do not, the lesson here is that you may have to try out different *combinations* of  $\Delta x$  and  $\Delta t$  variations until a stable and reasonable solution is obtained. You may expect, nonetheless, that there are choices for  $\Delta x$  and  $\Delta t$  for which the numeric solution fails, and that simply decreasing an individual  $\Delta x$  or  $\Delta t$ , in the hope that this will increase precision, may not improve the solution.

#### 24.4.1

##### Heat Equation Implementation

Recall, we want to solve for the temperature distribution within an aluminum bar of length  $L = 1$  m subject to the boundary and initial conditions

$$T(x = 0, t) = T(x = L, t) = 0^\circ\text{C} \quad T(x, t = 0) = 100^\circ\text{C} \quad (24.21)$$

The International system constants for iron's specific heat, thermal conductivity, and density are:

$$C = 0.113 \text{ cal}/^\circ\text{C g} \quad K = 0.12 \text{ cal}^\circ\text{C g s} \quad \rho = 7.8 \text{ g/cm}^3 \quad (24.22)$$

1. Write or modify `EqHeat.java` in Listing 24.1 to solve the heat equation.
2. Define a 2D array `T[101][2]` for the temperature as a function of space and time. The first index is for the 100 space divisions of the bar, and the second index for present and past times (because thousands of time steps may be made, we save memory by saving only two times).
3. For time  $t = 0$ , ( $j=1$ ), initialize `T` so that all points on the bar except the end points are at  $100^\circ\text{C}$ . Set the temperatures of the ends to  $0^\circ\text{C}$ .
4. Apply (24.14) to obtain the temperature at the next time step.
5. Assign the present-time values of the temperature to the past values:  
`T[i][1] = T[i][2], i=1, ..., 101.`

**Listing 24.1:** EqHeat.java solves the heat equation for a 1D space and time, by leapfrogging (time stepping) the initial conditions forward in time. You will need to adjust the parameters to obtain a solution like those in the figures.

```
// EqHeat.java: Solve heat equation via finite differences

import java.io.*; // Import IO library

public class EqHeat { // Class constants in MKS units

    public static final int Nx = 11, Nt = 300; // Grid sizes
    public static final double Dx = 0.01, Dt = 0.1; // Step sizes
    public static final double KAPPA = 210.; // Thermal conductivity
    public static final double SPH = 900.; // Specific heat
    public static final double RHO = 2700.; // Al density

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {

        int ix, t;
        double T[][] = new double[Nx][2];
        double cons;

        PrintWriter q = new PrintWriter // File for gnuplot
            (new FileOutputStream("EqHeat.dat"), true);
        for (ix=1; ix < Nx-1; ix++) T[ix][0] = 100.; // Initialize
        T[0][0] = 0.; // Except ends
        T[0][1] = 0.;
        T[Nx-1][0] = 0.;
        T[Nx-1][1] = 0.;
        cons = KAPPA/(SPH*RHO)*Dt/(Dx*Dx); // Integration factor
        System.out.println("constant = " + cons); // t loop
        for (t=1; t <= Nt; t++) {
            for (ix=1; ix < Nx-1; ix++) T[ix][1] = T[ix][0]
                + cons*(T[ix+1][0] + T[ix-1][0] - 2.*T[ix][0]); // x loop
            if (t%10==0 || t==1) { // Save every N steps
                for (ix = 0; ix < Nx; ix++) q.println(T[ix][1]);
                q.println(); // Blank line ends row
            }
            for (ix = 1; ix < Nx-1; ix++) T[ix][0] = T[ix][1]; // New to old
        }
        System.out.println("data stored in EqHeat.dat");
    } // End main
} // End class
```

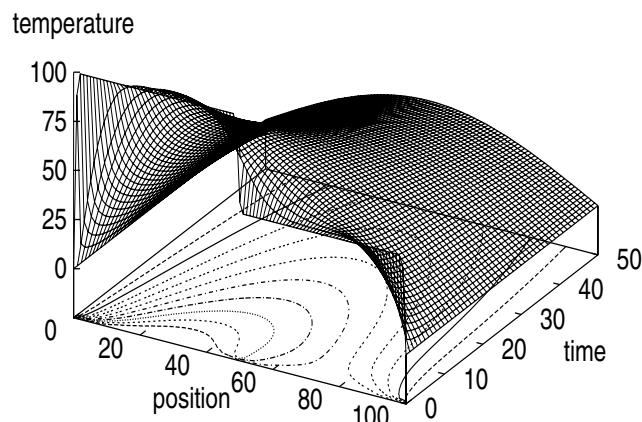
6. Start with 50 time steps. Once you are confident the program is running properly, use thousands of steps to see the bar cool smoothly with time. For every  $\sim$ 500 time steps, print the time and temperature along the bar.

## 24.5

### Assessment and Visualization

1. Check that your program gives a temperature distribution that varies smoothly along the bar, and which agrees with the boundary conditions.
2. Check that your program gives a temperature distribution that varies smoothly with time and attains equilibrium. You may have to vary the time and space steps to obtain well-behaved solutions.
3. Compare the analytic and numeric solutions (and the wall times needed to compute them). If the solutions differ, suspect the one which does not appear smooth and continuous.
4. Make surface plots of the temperature vs. position for several times.
5. Better yet, make a surface plot of the temperature vs. position vs. time.
6. Plot the *isotherms* (contours of constant temperature).
7. **Stability test:** Check (24.20) that the temperature diverges in  $t$  if  $\eta > 1/4$ .
8. **Material dependence:** Repeat the calculation for iron. Take note that the stability condition requires you to change the size of the time step.
9. **Initial sinusoidal distribution**  $\sin(\pi x/L)$ : Compare to analytic solution,

$$T(x, t) = \sin(\pi x/L) e^{-\pi^2 K t / (L^2 C \rho)} \quad (24.23)$$



**Fig. 24.4** Temperature versus position and time when two bars of differing temperature are placed in contact at  $t = 0$ . The projected contours show the isotherms.

10. **Two bars in contact:** Two identical bars 0.25 m long are placed in contact along one of their ends with their other ends kept at 0°C (Fig. 24.4). One is kept in a heat bath at 100°C, and the other at 50°C. Determine how the temperature varies with time and location.
11. **Radiating bar (Newton's cooling):** Imagine now, that instead of being insulated along its length, a bar is in contact with an environment at a temperature  $T_e$ . Newton's law of cooling (radiation) says that the rate of temperature change due to radiation is

$$\frac{\partial T}{\partial t} = -h(T - T_e) \quad (24.24)$$

where  $h$  is a positive constant. This leads to the modified heat equation

$$\frac{\partial T(x, t)}{\partial t} = \frac{K}{C\rho} \frac{\partial^2 T}{\partial x^2} - hT(x, t) \quad (24.25)$$

Modify the algorithm to include Newton's cooling, and compare the cooling of this bar with that of the insulated bar.

**25****PDE Waves on Strings and Membranes**

*In this chapter we explore the numerical solution of wave equations. We have two purposes in mind. First, and especially if you have skipped the discussion of the heat equation in Chap. 24, we wish to give another example of how initial conditions in time are treated with a time stepping or leapfrog algorithm. Second, we wish to demonstrate that once we have a working algorithm for solving a wave equation, we can include considerably more physics than is possible with analytic treatments. Indeed, we will see that solutions can be found even after including friction, variable density, gravity, dispersion and nonlinearities.*

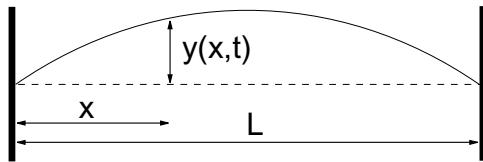
**Problem:** Recall the demonstration from elementary physics in which a string tied down at both ends is plucked “gently” at one location and a pulse is observed to travel along the string. Likewise, if the string has one end free and you shake it just right, a standing-wave pattern is set up in which the nodes remain in place and the antinodes move just up and down. Your **problem** is develop accurate models for wave propagation on a string, and to see if you can set up traveling- and standing-wave patterns.<sup>1</sup>

**25.1****The Hyperbolic Wave Equation (Theory)**

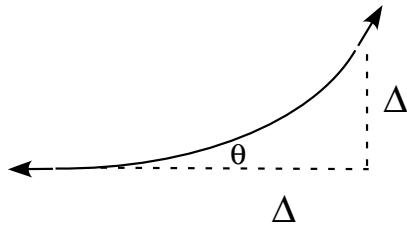
Consider a string of length  $L$  tied down at both ends (Fig. 25.1). The string has a constant density  $\rho$  per unit length, a constant tension  $T$ , is subject to no frictional forces, and the tension is high enough that we may ignore the sagging of the string due to gravity. We assume that the displacement of the string  $y(x, t)$  from its rest position is in the vertical direction only, and that it is a function of the horizontal location along the string  $x$  and the time  $t$ .

To obtain a simple, linear equation of motion (nonlinear wave equations are discussed in Unit II), we assume that the string’s relative displacement  $y(x, t)/L$  and slope  $\partial y / \partial x$  are small. We isolate an infinitesimal section  $\Delta x$  of the string (Fig. 25.2). We see that the difference in the vertical components of

<sup>1</sup> Some similar, but independent, studies can also be found in [74].



**Fig. 25.1** A stretched string of length  $L$  tied down at both ends, under high enough tension to ignore gravity. The vertical disturbance of the string from its equilibrium position is  $y(x, t)$ .



**Fig. 25.2** A differential element of the string showing how the string's displacement leads to the restoring force.

the tension on either end of the string produces the restoring force that accelerates this section of the string in the vertical direction. By applying Newton's laws to this section, we obtain the familiar wave equation:

$$\sum F_y = \rho \Delta x \frac{\partial^2 y}{\partial t^2} \quad (25.1)$$

$$\sum F_y = T \sin[\theta(x + \Delta x)] - T \sin[\theta(x)] \quad (25.2)$$

$$= T \frac{\partial y}{\partial x} \Big|_{x+\Delta x} - T \frac{\partial y}{\partial x} \Big|_x \simeq T \frac{\partial^2 y}{\partial x^2} \quad (25.3)$$

$$\Rightarrow \frac{\partial^2 y(x, t)}{\partial x^2} = \frac{1}{c^2} \frac{\partial^2 y(x, t)}{\partial t^2} \quad c = \sqrt{T/\rho} \quad (25.4)$$

where we have assumed that  $\theta$  is small enough for  $\sin \theta \simeq \tan \theta = \partial y / \partial x$ . The existence of two independent variables  $x$  and  $t$  makes this a PDE. The constant  $c$  is the velocity with which a disturbance travels along the wave, and is seen to decrease for a heavier string, and increase for a tighter one. Note, this signal velocity  $c$  is *not* the same as the velocity of a string element  $\partial y(x, t) / \partial t$ .

The initial conditions for our problem is that the string is plucked gently and released. We assume that the "pluck" places the string into a triangular shape with the center of triangle 8/10th of the way down the string and with

a height of 1:

$$y(x, t=0) = \begin{cases} 1.25x/L & x \leq 0.8L \\ (5 - 5x/L), & x > 0.8L \end{cases} \quad (\text{initial condition 1}) \quad (25.5)$$

Because (25.4) is second-order in time, a second initial condition (beyond initial displacement) is needed to determine the solution. We interpret the “gentleness” of the pluck to mean the string is released from rest:

$$\frac{\partial y}{\partial t}(x, t=0) = 0 \quad (\text{initial condition 2}) \quad (25.6)$$

The boundary conditions for our problem are that both ends of the string are tied down for all times:

$$y(0, t) \equiv 0 \quad y(L, t) \equiv 0 \quad (\text{boundary conditions}) \quad (25.7)$$

### 25.1.1

#### Solution via Normal Mode Expansion

The “analytic solution” to (25.4) is obtained via the familiar separation-of-variables technique. We assume that the solution is the product of a function of space times a function of time:

$$y(x, t) = X(x)T(t) \quad (25.8)$$

We substitute (25.8) into (25.4), divide by  $y(x, t)$ , and are left with an equation that has a solution only if there are solutions to the two ODEs:

$$\frac{d^2T(t)}{dt^2} + \omega^2 T(t) = 0 \quad (25.9)$$

$$\frac{d^2X(x)}{dx^2} + k^2 X(x) = 0 \quad k \stackrel{\text{def}}{=} \frac{\omega}{c} \quad (25.10)$$

The angular frequency  $\omega$  and the wave vector  $k$  are determined by demanding that the solutions satisfy the boundary conditions. Specifically, the string being attached at both ends demands

$$X(x=0, t) = X(x=L, t) = 0 \quad (25.11)$$

$$\Rightarrow X_n(x) = A_n \sin k_n x \quad k_n = \frac{\pi(n+1)}{L} \quad n = 0, 1, \dots \quad (25.12)$$

The time solution is

$$T_n(t) = C_n \sin \omega_n t + D_n \cos \omega_n t \quad \omega_n = nck_0 = n \frac{2\pi c}{L} \quad (25.13)$$

where the frequency of this  $n$ th *normal mode* is also fixed. In fact, it is the single frequency of oscillation that defines a normal mode.

The *initial condition* (25.5) of zero velocity  $\partial y/\partial t(t = 0) = 0$ , requires the  $C_n$  values in (25.13) to be zero. Putting the pieces together, the normal-mode solutions are

$$y_n(x, t) = \sin k_n x \cos \omega_n t \quad n = 0, 1, \dots \quad (25.14)$$

Since the wave equation (25.4) is linear in  $y$ , the principle of linear superposition holds and the most general solution for waves on a string with fixed ends can be written as the sum of normal modes:

$$y(x, t) = \sum_{n=0}^{\infty} B_n \sin k_n x \cos \omega_n t \quad (25.15)$$

The Fourier coefficient  $B_n$  is determined by using the second initial condition (25.5), which describes how the wave is plucked. We start with

$$y(x, t = 0) = \sum_n B_n \sin nk_0 x, \quad (25.16)$$

multiply both sides by  $\sin mk_0 x$ , substitute the value of  $y(x, 0)$  from (25.5), and integrate from 0 to  $l$  to obtain

$$B_m = 6.25 \frac{\sin(0.8m\pi)}{m^2\pi^2}. \quad (25.17)$$

You will be asked to compare the Fourier series (25.15) to our numerical solution. While it is in the nature of the approximation that the precision of the numerical solution depends on the choice of step sizes, it is also revealing to realize that the precision of the “analytic” solution depends on summing an infinite number of terms, which can only be done only approximately.

### 25.1.2

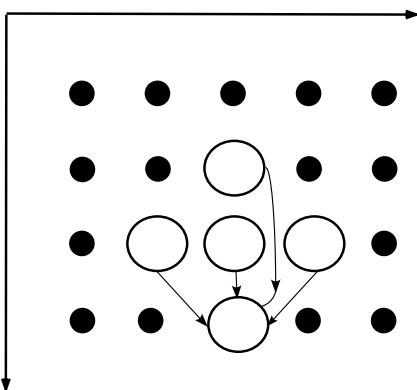
#### Algorithm: Time Stepping (Leapfrog)

As with the Laplace and heat equations, we look for a solution  $y(x, t)$  only for discrete values of the independent variables  $x$  and  $t$  on a grid (Fig. 25.3):

$$x = i\Delta x \quad i = 1, \dots, N_x \quad t = j\Delta t \quad j = 1, \dots, N_t \quad (25.18)$$

$$y(x, t) = y(i\Delta x, j\Delta t) \stackrel{\text{def}}{=} y_{i,j} \quad (25.19)$$

In contrast to Laplace’s equation where the grid was in two space dimensions, the grid in Fig. 25.3 is in both space and time. That being the case, moving across a row corresponds to increasing  $x$  values along the string for a fixed



**Fig. 25.3** The solutions of the wave equation for four earlier space–time points are used to obtain the solution at the present time.

time, while moving down a column corresponds to increasing time steps for a fixed position. Even though the grid in Fig. 25.3 may be square, we cannot use a relaxation technique for the solution because we do not know the solution on all four sides. The boundary conditions determine the solution along the right and left sides, while the initial time condition determines the solution along the top.

As with the Laplace equation, we use the central difference approximation to *discretize* the wave equation into a difference equation. First we express the second derivatives in terms of finite differences:

$$\frac{\partial^2 y}{\partial t^2} \simeq \frac{y_{i,j+1} + y_{i,j-1} - 2y_{i,j}}{(\Delta t)^2} \quad (25.20)$$

$$\frac{\partial^2 y}{\partial x^2} \simeq \frac{y_{i+1,j} + y_{i-1,j} - 2y_{i,j}}{(\Delta x)^2} \quad (25.21)$$

Substituting (25.20) in the wave equation (25.4) yields the difference equation

$$\frac{y_{i,j+1} + y_{i,j-1} - 2y_{i,j}}{c^2(\Delta t)^2} = \frac{y_{i+1,j} + y_{i-1,j} - 2y_{i,j}}{(\Delta x)^2}. \quad (25.22)$$

Notice that this equation contains three time values:  $j+1$  = the future,  $j$  = the present, and  $j-1$  = the past. Consequently, we rearrange it into a form that permits us to predict the future solution from the present and past solutions:

$$y_{i,j+1} = 2y_{i,j} - y_{i,j-1} + \frac{c^2}{c'^2} [y_{i+1,j} + y_{i-1,j} - 2y_{i,j}] \quad (25.23)$$

$$c' = \Delta x / \Delta t \quad (25.24)$$

Here  $c'$  is a combination of numerical parameters with the dimension of velocity, whose size relative to  $c$  determines the stability of the algorithm. The algorithm (25.23) propagates the wave from the two earlier times,  $j$  and  $j - 1$ , and from three nearby positions,  $i - 1$ ,  $i$ , and  $i + 1$ , to a later time  $j + 1$  and a single space position  $i$  (Fig. 25.3).

As you have seen in our discussion of the heat equation, a leapfrog method is quite different from a relaxation technique. We start with the solution along the topmost row, and then move down forward, one step at a time. If we write out the solution for present times to a file, then we need to store only three time values on the computer, which this saves memory. In fact, because the time steps must often be quite small to obtain high precision, you may only want to store the solution for every fifth or tenth times.

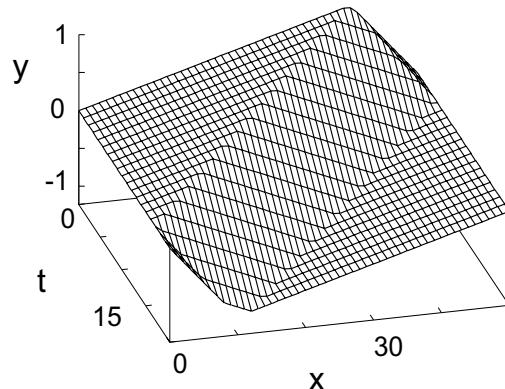
Initializing the recurrence relation is a bit tricky because it requires displacements from two earlier times, whereas the initial conditions are only for one time. Nonetheless, the rest condition (25.5), when combined with the *forward-difference* approximation, lets us extrapolate to negative time:

$$\frac{\partial y}{\partial t}(x, 0) \simeq \frac{y(x, 0) - y(x, -\Delta t)}{\Delta t} = 0 \quad (25.25)$$

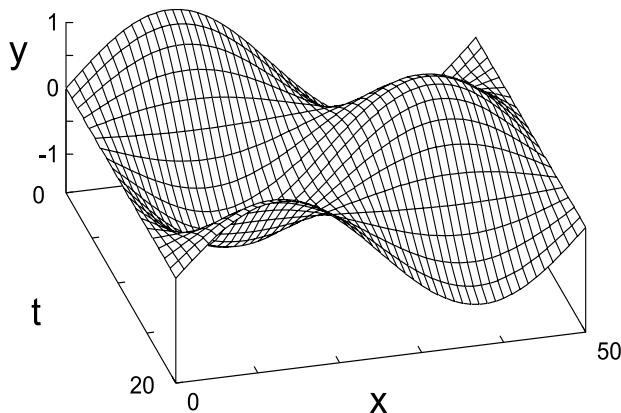
$$\Rightarrow y_{i,0} = y_{i,1} \quad (25.26)$$

Here we take the initial time as  $j = 1$ , and so  $j = 0$  corresponds to  $t = -\Delta t$ . Substituting this relation into (25.23) yields for the initial step

$$y_{i,2} = y_{i,1} + \frac{c^2}{c'^2} [y_{i+1,1} + y_{i-1,1} - 2y_{i,1}] \quad (t = \Delta t \text{ only}) \quad (25.27)$$



**Fig. 25.4** The vertical displacement as a function of position  $x$  and time  $t$  for a string initially plucked near its right end forms a pulse that divides into waves traveling to the right and to the left.)



**Fig. 25.5** The vertical displacement as a function of position  $x$  and time  $t$  for a string initially placed in a normal mode. Notice how the standing wave moves up and down with time. (In this and Fig. 25.4,  $y/L$  should be small, but that is harder to show.)

Equation (25.27) uses the solution throughout all of space at the initial time  $t = 0$  to propagate (leapfrog) it forward to a time  $\Delta t$ . Subsequent time steps use (25.23) and are continued for as long as you like.

As is also true with the heat equation, the success of the numerical method depends on the relative sizes of the time and space steps. If we apply a von Neumann stability analysis to this problem by substituting  $y_{m,j} = \xi^j \exp(ikm\Delta x)$ , as we did in Section 24.4, a more complicated equation results. Nonetheless, [9] shows that the difference-equation solution will be stable for the general class of transport equations if

$$c \leq c' = \Delta x / \Delta t \quad (\text{Courant condition}). \quad (25.28)$$

Equation (25.28) means that the solution gets better with smaller *time* steps, but gets worse for smaller space steps (unless you simultaneously make the time step smaller). Having different sensitivities to the time and space steps may appear surprising because the wave equation (25.4) is symmetric in  $x$  and  $t$ , yet the symmetry is broken by the nonsymmetric initial and boundary conditions.

**Exercise:** Figure out a procedure for solving for the wave equation for all times in just one step. Estimate how much memory would be required for that.  $\square$

**Exercise:** Can you figure out a procedure solving for the wave motion with a relaxation technique? What would you take as your initial guess, and how would you know when the procedure has converged?  $\square$

## 25.1.3

**Wave Equation Implementation**

The program `EqString.java` in Listing 25.1 solves the wave equation for a string of length  $L = 1$  m with its ends fixed and with the gently-plucked initial conditions. Note,  $L = 1$  violates the assumption that  $y/L \ll 1$ , but makes it easy to display the results; you should try  $L = 1000$  to be realistic. The values of density and tension are entered as constants,  $\rho = 0.01$  kg/m,  $T = 40$  N, with the space grid set at 101 points, corresponding to  $\Delta = 0.01$  cm.

## 25.1.4

**Assessment and Exploration**

1. Run the simulation and make a surface plot of the results.
2. Explore a number of space and time step combinations. In particular, try steps that satisfy and that do not satisfy the Courant condition (25.28). Does your exploration agree with the stability condition?
3. Compare the analytic and numeric solutions, summing at least 200 terms in the “analytic” solution.
4. Use the time dependence of your graph to estimate the peak’s propagation velocity  $c$ . Compare the deduced  $c$  to (26.6).
5. Our solution of the wave equation for a plucked string leads to the formation of a wave packet, which corresponds to multiple normal modes of the string. In Fig. 25.5 we show the motion of a string for the initial conditions,

$$y(x, t = 0) = 0.001 \sin 2\pi x \quad \frac{\partial y}{\partial t}(x, t = 0) = 0 \quad (25.29)$$

which excite just one normal mode. Modify the program to incorporate this initial condition and see if a normal mode results.

6. Observe the motion of the wave for initial conditions corresponding to the sum of two adjacent normal modes. Does beating occur?
7. When a string is plucked near its end, a pulse reflects off the ends and bounces back and forth. Change the initial conditions of the model program to one corresponding to a string plucked exactly in its middle, and see if a traveling or a standing wave results.
8. (Optional) Figs. 25.6 and 25.7 show the wave packets that result as a function of time for initial conditions corresponding to the double pluck indicated on the left of the figure. Verify that initial conditions of the form

**Listing 25.1:** EqString.java solves the wave equation via time stepping for a string of length  $L = 1$  m with its ends fixed and with the gently-plucked initial conditions. You will need to modify this code to include new physics.

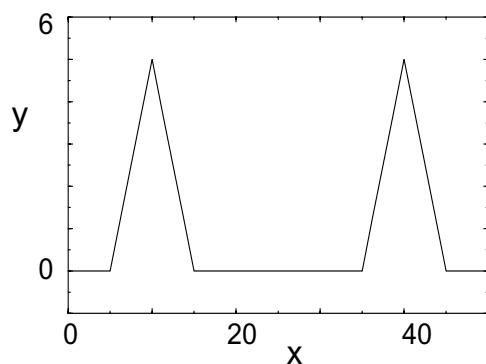
```
// Eqstring.java: Solution of wave equation via time stepping
// Output in 3D gnuplot format

import java.io.*;

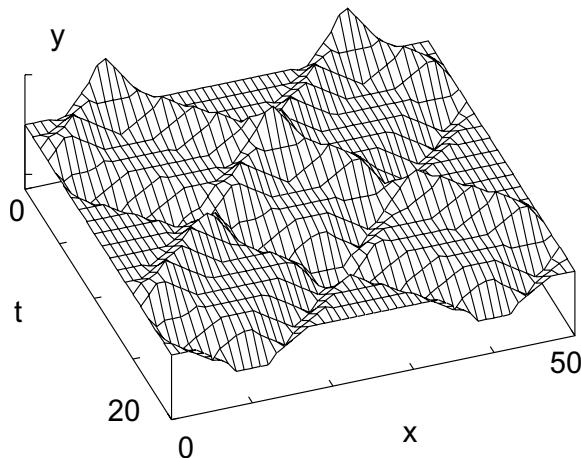
public class Eqstring {
    final static double rho = 0.01, ten = 40., max = 100.;

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
        int i, k;
        double x[][] = new double[101][3];
        double ratio, c, c1;
        PrintWriter w = new PrintWriter
            (new FileOutputStream("EqString.dat"), true);
        c = Math.sqrt(ten/rho); // Propagation speed
        c1 = c; // CFL criteria
        ratio = c*c/(c1*c1); // Initial configuration
        for (i=0; i < 81; i++) x[i][0] = 0.00125*i;
        for (i=81; i < 101; i++) x[i][0] = 0.1 - 0.005*(i-80); // First time step
        for (i=1; i < 100; i++) {
            x[i][1] = x[i][0] + 0.5 * ratio * (x[i+1][0] + x[i-1][0] - 2*x[i][0]); // Later time steps
            for (k=1; k < max; k++) {
                for (i=1; i < 100; i++) x[i][2] = 2.*x[i][1] - x[i][0] + ratio * (x[i+1][1] + x[i-1][1] - 2*x[i][1]);
                for (i=0; i < 101; i++) {
                    x[i][0] = x[i][1];
                    x[i][1] = x[i][2];
                }
                if ((k%5) == 0) { // Print every 5th point
                    for (i=0; i < 101; i++) {
                        w.println("") + x[i][2] + ""); } // Gnuplot 3D format
                    w.println(""); // Empty line for gnuplot
                }
            }
        System.out.println("data in EqString.dat, gnuplot format");
    }
}
```

$$\frac{y(x, t=0)}{0.005} = \begin{cases} 0 & 0.0 \leq x \leq 0.1 \\ 10x - 1 & 0.1 \leq x \leq 0.2 \\ -10x + 3 & 0.2 \leq x \leq 0.3 \\ 0, & 0.3 \leq x \leq 0.7 \\ 10x - 7 & 0.7 \leq x \leq 0.8 \\ -10x + 9 & 0.8 \leq x \leq 0.9 \\ 0 & 0.9 \leq x \leq 1.0 \end{cases} \quad (25.30)$$



**Fig. 25.6** The initial configuration of a string plucked in two places simultaneously.

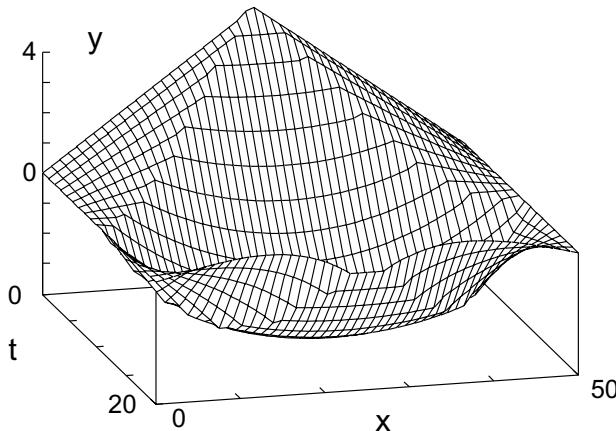


**Fig. 25.7** The vertical displacement as a function of position and time of a string initially plucked simultaneously at two points, as shown on the left. Note that each initial peak breaks up into waves traveling to the right and to the left. The traveling waves invert on reflection from the fixed end. As a consequence of these inversions, the  $t = 15$  wave is an inverted  $t = 0$  wave.

leads to this type of a repeating pattern. In particular, observe whether the pulses move or just oscillate up and down.

### 25.1.5 Including Friction (Extension)

The string problem we have investigated so far can be handled by either numerical or analytic techniques. We now wish to extend the theory to include some more realistic physics. These extensions have only numerical solutions.



**Fig. 25.8** The vertical displacement as a function of position and time of a string with friction initially plucked at its middle.

Real plucked strings do not vibrate forever because the real world contains friction. Consider again the element of string between  $x$  and  $x + dx$  (Fig. 25.2), but imagine now that this element is moving in a viscous fluid, such as air. An approximate model for the frictional force is to have it point in a direction opposite to the (vertical) velocity of the string, proportional to that velocity, as well as proportional to the length of the element:

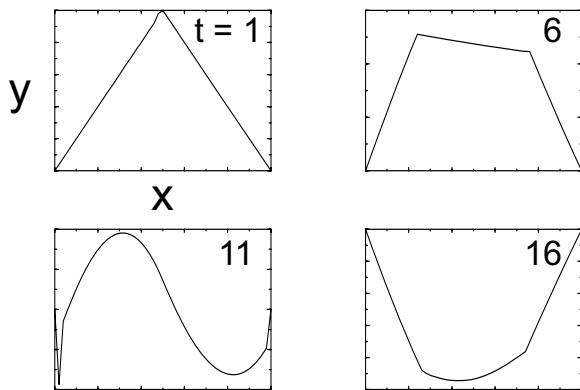
$$F_f \simeq -2\kappa\Delta x \frac{\partial y}{\partial t} \quad (25.31)$$

Here  $\kappa$  is a constant that is proportional to the viscosity of the medium in which the string is vibrating. Including this force in the equation of motion changes the wave equation to

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2} - \frac{2\kappa}{\rho} \frac{\partial y}{\partial t} \quad (25.32)$$

In Figs. 25.8 and 25.9 we show the resulting motion of a string plucked in the middle when friction is included. Observe how the initial pluck breaks up into waves traveling to the right and left that get reflected and inverted by the fixed ends. Because those parts of the wave with the greatest velocity experience the greatest friction, the peak tends to get smoothed out the most as time progresses.

**Exercise:** Generalize the algorithm for the wave equation to include friction and observe the change in wave's behavior. Start off with  $T = 40$  N,  $\rho = 10$  kg/m, and pick a value of  $\kappa$  that is large enough to cause a noticeable effect, but not so large as to stop the oscillations. As a check, reverse the sign of  $\kappa$  and



**Fig. 25.9** Disturbance versus position for a string with variable density that is initially plucked at its center. The disturbances at four times are given. At  $t = 6$  we see that the wave moves faster in the denser region to the right, but that its amplitude decreases because the string is heavier there.

see if the wave grows in time (which would eventually violate our assumption of small oscillations).  $\square$

### 25.1.6

#### Variable Tension and Density (Extension)

We have derived the propagation velocity for waves on a string as  $c = \sqrt{T/\rho}$ . This says that waves move slower in regions of high density, and faster in regions of high tension. If the density of the string varies, to illustrate, by having the ends thicker in order to support the weight of the middle, then  $c$  will no longer be a constant and our wave equation needs fixing. In addition, if the density varies, then the tension would too because it takes a greater tension to support a greater mass. If gravity acts, then we would also expect that the tension at the ends of the string to be higher because they must support the entire weight of the string.

To derive the proper equation for wave motion, consider again an element of string (Fig. 25.2), and our derivation of the wave equation. If we ignore friction but do not assume the tension  $T$  is constant, then Newton's second law gives

$$F = ma \Rightarrow \frac{\partial}{\partial x} \left[ T(x) \frac{\partial y(x, t)}{\partial x} \right] \Delta x = \rho(x) \Delta x \frac{\partial^2 u(x, t)}{\partial t^2} \quad (25.33)$$

$$\Rightarrow \frac{\partial T(x)}{\partial x} \frac{\partial y(x, t)}{\partial x} + T(x) \frac{\partial^2 y(x, t)}{\partial x^2} = \rho(x) \frac{\partial^2 y(x, t)}{\partial t^2} \quad (25.34)$$

If  $\rho(x)$  and  $T(x)$  are known functions, then these equations can be solved with just a small modification of our algorithm.

Deducing a  $T(x)$  for a given  $\rho(x)$  in the presence of gravity is a hard statics problem because it also requires the solution for equilibrium shape of the string. While we solve that problem below, for those interested in an easier problem that still shows the new physics, you may assume that the density and tension are proportional:

$$\rho(x) = \rho_0 e^{\alpha x} \quad T(x) = T_0 e^{\alpha x}. \quad (25.35)$$

While we would expect the tension to be greater in regions of higher density (more mass to move), being proportional is just an approximation. Substitution of these relations into (25.34) yields the new wave equation:

$$\frac{\partial^2 y(x,t)}{\partial x^2} + \alpha \frac{\partial y(x,t)}{\partial x} = \frac{1}{c^2} \frac{\partial^2 y(x,t)}{\partial t^2} \quad c^2 = \frac{T_0}{\rho_0}. \quad (25.36)$$

Here  $c$  is a constant that would be the wave velocity if  $\alpha = 0$ . This equation is similar to the wave equation with friction, only now the first derivative is with respect to  $x$ , and not  $t$ . The corresponding difference equations follow from using the central difference approximations for the derivatives:

$$\begin{aligned} y_{i,j+1} &= 2y_{i,j} - y_{i,j-1} + \frac{\alpha c^2 (\Delta t)^2}{2\Delta x} [y_{i+1,j} - y_{i,j}] + \frac{c^2}{\Delta x^2} [y_{i+1,j} + y_{i-1,j} - 2y_{i,j}] \\ y_{i,2} &= y_{i,1} + \frac{c^2}{\Delta x^2} [y_{i+1,1} + y_{i-1,1} - 2y_{i,1}] + \frac{\alpha c^2 (\Delta t)^2}{2\Delta x} [y_{i+1,1} - y_{i,1}] \end{aligned} \quad (25.37)$$

## 25.2

### Realistic 1D Wave Exercises

Do these exercises for the assumed density and tension given by (25.35). Include friction in order to make the simulation realistic. Assume  $\alpha = 0.5$ ,  $T_0 = 40$  N, and  $\rho_0 = 0.01$  kg/m.

1. Modify the algorithm in your program to handle variable tension and density, and friction. Run some typical cases and create surface plots of the results.
2. Explain in words how the waves dampen and how a wave's velocity appears to change. The behavior you obtain may look something like that shown in Fig. 25.8.
3. **Normal Modes:** Search for normal mode solutions of the variable-tension wave equation, that is, solutions that vary like

$$u(x,t) = A \cos(\omega t) \sin(\gamma x)$$

Try using this form to start off your algorithm and see if you can find standing waves. Use large values for  $\omega$ .

4. When conducting physics demonstrations, we set up standing wave patterns by driving one end of the string periodically. Try doing the same with your algorithm; that is, build into your code the conditions that

$$y(x = 0, t) = A \sin \omega t$$

for all times. Try to vary  $A$  and  $\omega$  until a normal mode (standing wave) is obtained.

5. If you were able to find standing waves, then verify that this string acts like a high-frequency filter, namely, that there is a frequency below which no waves occur. (This is for exponential density case.)
6. For the catenary problem, plot up your results showing *both* the disturbance  $u(x, t)$  about the catenary and the actual height  $y(x, t)$  above the horizontal for a plucked string initial condition.
7. Try the first two normal modes for a uniform string as the initial conditions for the catenary. These should be close to, but not exactly normal modes.
8. We derived the normal modes for a uniform string after assuming that  $k(x) = \omega/c(x)$  is a constant. For a catenary without too much  $x$  variation of the tension, we should be able to make the approximation

$$c(x)^2 \simeq \frac{T(x)}{\rho} = \frac{T_0 \cosh(x/d)}{\rho}$$

See if you get a better representation of the first two normal modes if you include some  $x$  dependence to  $k$ .

### 25.3

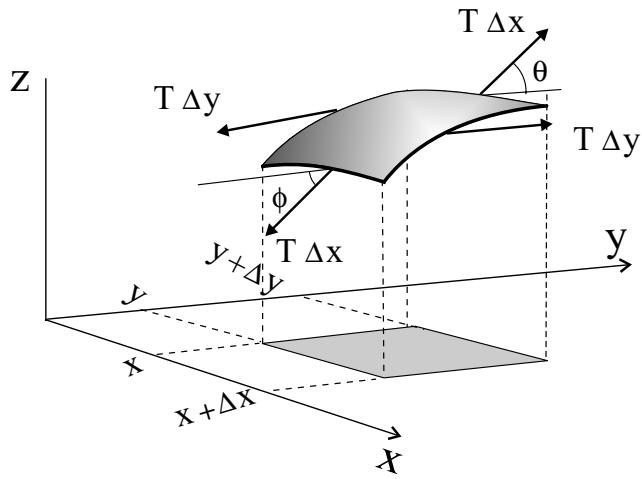
#### Vibrating Membrane (2D Waves)

**Problem:** An elastic membrane is stretched across the top of a square box of sides  $\pi$  and attached securely. The tension per unit length in the membrane is  $T$ . Initially the membrane is placed in the asymmetrical shape,

$$u(x, y, t = 0) = \sin 2x \sin y \quad 0 \leq x \leq \pi \quad 0 \leq y \leq \pi \quad (25.38)$$

where  $u$  is the vertical displacement from equilibrium. Your **problem** is to describe the motion of the membrane when it is released from rest [71].

The description of wave motion on a membrane is basically the same as that of 1D waves on a string discussed in Section 25.1, only now we have wave propagation in two directions. Consider Fig. 25.10 showing a square section of the membrane under tension  $T$ . The membrane only moves vertically in the  $z$  direction, yet because the restoring force arising from the tension in the membrane varies in both the  $x$  and  $y$  directions, there is wave motion along the surface of the membrane.



**Fig. 25.10** A small part of an oscillating membrane and the forces that act on it.

Although the tension is constant over the small area in Fig. 25.10, there will be a net vertical force on the segment if the angle of incline of the membrane varies as we move through space. Accordingly, the net force on the membrane in the  $z$  direction due to the change in  $y$  is

$$\sum F_z(x = \text{constant}) = T\Delta x \sin \theta - T\Delta x \sin \phi \quad (25.39)$$

where  $\theta$  is the angle of incline at  $y + \Delta y$  and  $\phi$  the angle at  $y$ . Yet if we assume that the angles are small (small displacements), then we can make the approximations:

$$\sin \theta \approx \tan \theta = \frac{\partial u}{\partial y} \Big|_{y+\Delta y} \quad (25.40)$$

$$\sin \phi \approx \tan \phi = \frac{\partial u}{\partial y} \Big|_y$$

$$\Rightarrow \sum F_z(x = \text{constant}) = T\Delta x \left( \frac{\partial u}{\partial y} \Big|_{y+\Delta y} - \frac{\partial u}{\partial y} \Big|_y \right) \approx T\Delta x \frac{\partial^2 u}{\partial y^2} \Delta y$$

Similarly, the net force in the  $z$  direction due to the variation in  $y$  is

$$\sum F_z(y = \text{constant}) = T\Delta y \left( \frac{\partial u}{\partial x} \Big|_{x+\Delta x} - \frac{\partial u}{\partial x} \Big|_x \right) \quad (25.41)$$

$$\approx T\Delta y \frac{\partial^2 u}{\partial x^2} \Delta x \quad (25.42)$$

The membrane section has mass  $\rho\Delta x\Delta y$ , where  $\rho$  is the membrane's mass per unit area. We now apply Newton's second law to determine the acceleration of the membrane section in the  $z$  direction motion due to the sum of the net forces arising from both the  $x$  and  $y$  variations:

$$\begin{aligned} \rho\Delta x\Delta y \frac{\partial^2 u}{\partial t^2} &= T\Delta x \frac{\partial^2 u}{\partial y^2} \Delta y + T\Delta y \frac{\partial^2 u}{\partial x^2} \Delta x, \\ \Rightarrow \quad \frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} &= \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \quad c = \sqrt{T/\rho} \end{aligned} \quad (25.43)$$

This is the 2D version of the wave equation (25.4) that we studied previously in one dimension. Here  $c$ , the propagation velocity, is still the square root of tension over density, only now it is tension per unit length and mass per unit area.

## 25.4 Analytical Solution

The analytic or numerical solution of the partial differential equation (25.43) requires us to know both the boundary conditions and the initial conditions. The boundary conditions for our problem hold for all times and are given when we are told that the membrane is attached securely to a square box:

$$\begin{aligned} u(x = 0, y, t) &= u(x = \pi, y, t) = 0 \\ u(x, y = 0, t) &= u(x, y = \pi, t) = 0 \end{aligned} \quad (25.44)$$

The initial conditions has two parts, the shape of the membrane at time  $t = 0$ , and the velocity of each point of the membrane. The initial configuration is

$$u(x, y, t = 0) = \sin 2x \sin y \quad 0 \leq x \leq \pi \quad 0 \leq y \leq \pi \quad (25.45)$$

Second, we are told that the membrane is released from rest, which means:

$$\frac{\partial u}{\partial t} \Big|_{t=0} = v(x, y, t = 0) = 0, \quad (25.46)$$

where we write partial derivative because there are also spatial variations.

The analytic solution is based on the guess that because the wave equation (25.43) has the derivatives with respect to each coordinate and time separate, the full solution  $u(x, y, t)$  is the product of separate functions of  $x$ ,  $y$  and  $t$ :

$$u(x, y, t) = X(x) Y(y) T(t) \quad (25.47)$$

After substituting into (25.43) and dividing by  $X(x)Y(y)T(t)$ , we obtain:

$$\frac{1}{c^2} \frac{1}{T(t)} \frac{d^2 T(t)}{dt^2} = \frac{1}{X(x)} \frac{d^2 X(x)}{dx^2} + \frac{1}{Y(y)} \frac{d^2 Y(y)}{dy^2} \quad (25.48)$$

The only way that the LHS of (25.48) can depend only on time while the RHS depends only on coordinates, is if both sides are constant:

$$\frac{1}{c^2} \frac{1}{T(t)} \frac{d^2 T(t)}{dt^2} = -\xi^2 = \frac{1}{X(x)} \frac{d^2 X(x)}{dx^2} + \frac{1}{Y(y)} \frac{d^2 Y(y)}{dy^2} \quad (25.49)$$

$$\Rightarrow \frac{1}{X(x)} \frac{d^2 X(x)}{dx^2} = -k^2 \quad (25.50)$$

$$\frac{1}{Y(y)} \frac{d^2 Y(y)}{dy^2} = -q^2 \quad \text{where } q^2 + k^2 = \xi^2 \quad (25.51)$$

In (25.50) and (25.51) we have included the further deduction that since each term on the RHS of (25.49) depends on either  $x$  or  $y$ , then the only way their sum can be constant is if each term is a constant, in this case  $-k^2$ .

The solutions of these equations are standing waves in the  $x$  and  $y$  directions, which of course are all sinusoidal function,

$$\begin{aligned} X(x) &= A \sin kx + B \cos kx \\ Y(y) &= C \sin qy + D \cos qy \\ T(t) &= E \sin c\xi t + F \cos c\xi t \end{aligned}$$

We now apply the boundary conditions:

$$u(x = 0, y, t) = u(x = \pi, y, z) = 0 \Rightarrow B = 0 \quad k = m = 1, 2, \dots$$

$$u(x, y = 0, t) = u(x, y = \pi, t) = 0 \Rightarrow D = 0 \quad y = n = 1, 2, \dots$$

$$\Rightarrow X(x) = A \sin mx \quad Y(y) = C \sin ny$$

The fixed values for the eigenvalues  $m$  and  $n$  describing the modes for the  $x$  and  $y$  standing waves are equivalent to fixed values for the constants  $q^2$  and  $k^2$ . Yet since  $q^2 + k^2 = \xi^2$ , we must also have a fixed value for  $\xi^2$ :

$$\xi^2 = q^2 + k^2 \Rightarrow \xi_{mn} = \pi \sqrt{m^2 + n^2} \quad (25.52)$$

The full space–time solution now takes the form

$$u_{mn} = [G_{mn} \cos c\xi t + H_{mn} \sin c\xi t] \sin mx \sin ny \quad (25.53)$$

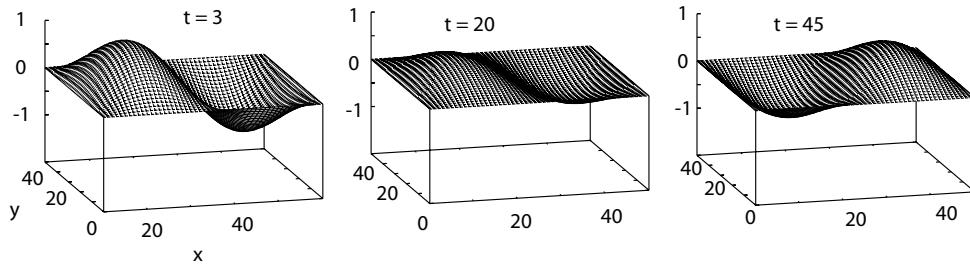
where  $n$  and  $m$  are integers. Because the wave equation is linear in  $u$ , its most general solution is a linear combination of the eigenmodes (25.53):

$$u(x, y, t) = \sum_{m=1}^{\infty} \sum_{n=1}^{\infty} [G_{mn} \cos c\xi t + H_{mn} \sin c\xi t] \sin mx \sin ny \quad (25.54)$$

While an infinite series is not generally a good algorithm, our choice of initial and boundary conditions means that only the  $m = 2, n = 1$  term contributes:

$$u(x, y, t) = \cos c\sqrt{5} \sin 2x \sin y \quad (25.55)$$

where  $c$  is the wave velocity. You should verify that initial and boundary conditions are indeed satisfied.



**Fig. 25.11** The standing wave pattern on a square box top at three different times.

## 25.5

### Numerical Solution for 2D Waves

The development of an algorithm for the solution of the 2D wave equation (25.43) follows that of the 1D equation in Section 25.1.2. We start by expressing the second derivatives in terms of central differences:

$$\frac{\partial^2 u(x, y, t)}{\partial t^2} = \frac{u(x, y, t + \Delta t) + u(x, y, t - \Delta t) - 2u(x, y, t)}{(\Delta t)^2} \quad (25.56)$$

$$\frac{\partial^2 u(x, y, t)}{\partial x^2} = \frac{u(x + \Delta x, y, t) + u(x - \Delta x, y, t) - 2u(x, y, t)}{(\Delta x)^2} \quad (25.57)$$

$$\frac{\partial^2 u(x, y, t)}{\partial y^2} = \frac{u(x, y + \Delta y, t) + u(x, y - \Delta y, t) - 2u(x, y, t)}{(\Delta y)^2} \quad (25.58)$$

After discretizing the variables,  $u(x = i\Delta, y = i\Delta y, t = k\Delta t) \equiv u_{i,j}^k$ , we obtain our time-stepping algorithm by solving for the future solution in terms of the present and past ones:

$$u_{i,j}^{k+1} = 2u_{i,j}^k - u_{i,j}^{k-1}c^2\left(\frac{\Delta t}{\Delta x}\right)^2 \left[ u_{i+1,j}^k + u_{i-1,j}^k - 4u_{i,j}^k + u_{i,j+1}^k + u_{i,j-1}^k \right] \quad (25.59)$$

Whereas the present ( $k$ ) and past ( $k - 1$ ) solutions are known after the first step, to get the algorithm going we need to know the solution at  $t = -\Delta t$ , that is, before the initial time. To find that, we use the fact that the membrane is released from rest:

$$0 = \frac{\partial u(t=0)}{\partial t} \approx \frac{u_{i,j}^1 - u_{i,j}^{-1}}{2\Delta t} \Rightarrow u_{i,j}^{-1} = u_{i,j}^1 \quad (25.60)$$

After substitution into (25.59) and solving for  $u^1$ , we obtain the algorithm for the first step:

$$u_{i,j}^1 = u_{i,j}^0 + \frac{1}{2}c^2\left(\frac{\Delta t}{\Delta x}\right)^2 \left[ u_{i+1,j}^0 + u_{i-1,j}^0 - 4u_{i,j}^0 + u_{i,j+1}^0 + u_{i,j-1}^0 \right] \quad (25.61)$$

Since the displacement  $u_{i,j}^0$  is given at time  $t = 0$  ( $k = 0$ ), we compute the solution for the first time step with (25.61) and with (25.59) for subsequent steps.

**Listing 25.2:** Wave2D.java

```
// Wave2D.java: 2D Wave eq for vibrating membrane by Manuel J. Paez
import java.io.*;

public class Wave2D {
    final static double den = 390.0, ten = 180.0;      // Density, T, step

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
        int i, j, k;           // i, j: membrane grid positions, k: for time
        int max = 45;          // Final time of oscillation
        double c, cprime, x, y; // vel; cprime = delta u/delta t
        double covercp, incr, incry; // c, cprime, increments
        double u[][][] = new double[101][101][3];
        PrintWriter w = new PrintWriter(new FileOutputStream("Helmholtz.dat")); // Output in Helmholtz.dat
        w.println("Helmholtz.dat");
        double ratio;           // (c/cprime)^2.
        incr = Math.PI/100.0;
        incry = Math.PI/100;
        c = Math.sqrt(ten/den); // Propagation speed
        cprime = c;             // For simplicity
        covercp = c/cprime;     // c / cprime
        ratio = 0.5*covercp*covercp; // 0.5 for stability
        System.out.println("ratio "+ratio);
        y = 0.0;
    }
}
```

```

    for( j=0; j<101; j++ ) {           // Initial condition: position
        x = 0.0;
        for( i=0; i<101; i++ ) {
            u[i][j][0] = Math.sin(2.0*x)*Math.sin(y);
            x = x+incr_x;
        }
        y = y+incr_y;
    }
    for ( j=1; j<100; j++ ) {           // First time step
        for ( i=1; i<100; i++ ) {
            u[i][j][1] = u[i][j][0] + 0.5* ratio *(u[i+1][j][0]+u[i-1][j][0]
                +u[i][j+1][0]+u[i][j-1][0]-4.0*u[i][j][0]);
        }
    }
    for ( k=1; k<=max; k++ ) {          // Later times
        for ( j=1; j<100; j++ ) {
            for ( i=1; i<100; i++ ) {
                u[i][j][2] = 2.*u[i][j][1] - u[i][j][0]+ratio *(u[i+1][j][1]
                    + u[i-1][j][1] + u[i][j+1][1]+u[i][j-1][1] - 4.*u[i][j][1]);
            }
        }
        for ( j=0; j<101; j++ ){
            for( i=0; i<101; i++ ){
                u[i][j][0] = u[i][j][1];           // New past
                u[i][j][1] = u[i][j][2];           // New present
            }
        }
        if ( k == max) {
            for ( j=0; j<101; j=j+2 ) {
                for( i=0; i<101; i=i+2 ) {
                    w.println(" " +u[i][j][2] );
                } //for gnuplot
                w.println("");
            }
        }
    }
    System.out.println("data stored in Helmholtz.dat");
}
}

```

The program `Wave2D.java` in Listing 25.2 solves the 2D wave equation using the time-stepping (leapfrog) algorithm. It continues iterating in time up to `max` steps. The shape of the membrane at three different times are shown in Fig. 25.11.

**26****Solitons; KdV and Sine-Gordon**

*This chapter examines how the inclusion of dispersion and nonlinearity affect wave behavior. Because these subjects are not often covered in traditional physics classes, we give more background materials than we normally do. We start with a linear chain of coupled pendulums, which should make it clear how the dispersion and nonlinearity arise physically, and then go to the continuum limit to obtain a differential equation of motion. Next we see how soliton waves arise in both one and two dimensions. Although solitons were originally discovered analytically, they were rediscovered computationally in recent times, and are now an active area of research.*

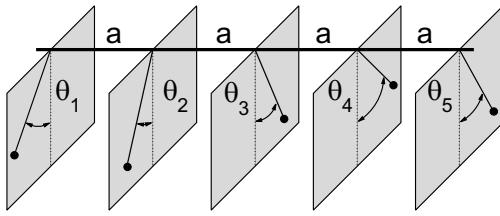
**26.1****Chain of Coupled Pendulums (Theory)**

In 1955, Fermi, Ulam, and Pastu were investigating how a 1D chain of coupled oscillators disperses waves. Since waves of differing frequencies traveled through the chain with differing speeds, a pulse broadens as time progresses, as each of its components travel with a different speed. Surprisingly, when the oscillators were made more realistic by introducing a nonlinear term into Hooke's law

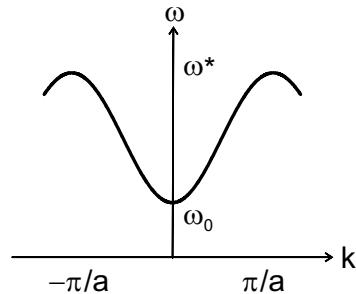
$$F(x) \simeq -k(x + \alpha x^2) \quad (26.1)$$

they found that even in the presence of dispersion, a sharp pulse in the chain would survive indefinitely. Your **problem** is to explain how this combination of dispersion and nonlinearity can combine to produce a stable pulse.

Since we have already studied nonlinear effects in a single pendulum (Chapter 19), we take as our model a 1D chain of identical pendulums connected by a torsion bar (Fig. 26.1). The angle  $\theta_i$  measures the displacement of pendulum  $i$  from its equilibrium position. If all the pendulums are set off swinging together,  $\theta_i \equiv \theta_j$ , the coupling torques would vanish and we would have our old friend, the equation for a realistic (albeit, very thick) pendulum. We assume that three torques act on each pendulum, a gravitational torque trying to return the pendulum to its equilibrium position, and the two torques from the twisting of the bar to the right and to the left of the pendulum. The



**Fig. 26.1** A 1D chain of pendulums coupled with a torsion bar on top. The pendulums swing in planes perpendicular to the length of the bar.



**Fig. 26.2** The dispersion relation for a linearized chain of pendulums.

equation of motion for pendulum  $j$  follows from Newton's law for rotational motion:

$$\sum_{j \neq i} \tau_{ji} = I \frac{d^2\theta_j(t)}{dt^2} \quad (26.2)$$

$$-\kappa(\theta_j - \theta_{j-1}) - \kappa(\theta_j - \theta_{j+1}) - mgL \sin \theta_j = I \frac{d^2\theta_j(t)}{dt^2} \quad (26.3)$$

$$\Rightarrow \kappa(\theta_{j+1} - 2\theta_j + \theta_{j-1}) - mgL \sin \theta_j = I \frac{d^2\theta_j(t)}{dt^2} \quad (26.4)$$

where  $I$  is the moment of inertia of each pendulum,  $L$  is the length of the pendulum, and  $\kappa$  is the torque constant of the bar. The nonlinearity in (26.4) arises from the  $\sin \theta \simeq \theta - \theta^3/6$  dependence of the gravitational torque. As it stands, (26.4) is a set of coupled nonlinear equations, with the number of equations equal to the number of oscillators.

## 26.2 Wave Dispersion

Consider a surfer remaining on the crest of a wave. Since she does not see the wave form change with time, her position is given by a function of the form

$f(kx - \omega t)$ . Consequently, to her the wave has a constant phase

$$kx - \omega t = \text{constant} \quad \Rightarrow \quad x = \omega t/k = \text{constant} \quad (26.5)$$

The surfer's (phase) velocity is the rate of change of  $x$  with respect to time,

$$v_p = \frac{dx}{dt} = \frac{\omega}{k} \quad (26.6)$$

which is constant. In general, the frequency  $\omega$  may be a nonlinear function of  $k$ , in which case the phase velocity varies with frequency and we have *dispersion*. If the wave contained just one frequency, then you would not observe any dispersion, but if wave was a pulse composed of many Fourier components, then it would broaden and change shape in time as each frequency moved with a differing phase velocity. There is no loss of energy due to dispersion, but the energy does disperse itself into more frequencies with time.

The functional relation between frequency  $\omega$  and the wave vector  $k$  is called a *dispersion relation* (Fig. 26.2). Information in a wave is often transmitted via pulses, with each pulse containing a group of Fourier components. If the Fourier components are centered around a mean frequency  $\omega_0$ , then the pulse (information) travels, not with the phase velocity, but with the *group velocity*

$$v_g = \left. \frac{\partial \omega}{\partial k} \right|_{\omega_0} \quad (26.7)$$

When there is dispersion, the group and phase velocities may differ.

To isolate the dispersive aspect of (26.4), we examine at its linear version

$$\frac{d^2\theta_j(t)}{dt^2} + \omega_0^2\theta_j(t) = \frac{\kappa}{I}(\theta_{j+1} - 2\theta_j + \theta_{j-1}) \quad (26.8)$$

where  $\omega_0 = \sqrt{mgL/I}$  is the natural frequency for any one pendulum. Because we want to determine if a wave with a single frequency propagates on this chain, we test if a traveling-wave with frequency  $\omega$  and wavelength  $\lambda$ ,

$$\theta_j(t) = Ae^{i(\omega t - kx_j)} \quad k = \frac{2\pi}{\lambda} \quad (26.9)$$

is a solution. Substitution of (26.9) into the wave equation (26.8) produces the *dispersion relation* (Fig. 26.2):

$$\omega^2 = \omega_0^2 - \frac{2\kappa}{I}(1 - \cos ka) \quad (\text{dispersion relation}) \quad (26.10)$$

To have dispersionless propagation (all frequencies propagate with the same velocity), we need a linear relation between  $\omega$  and  $k$ :

$$\lambda = c \frac{2\pi}{\omega} \quad \Rightarrow \quad \omega = ck, \quad (\text{dispersionless propagation}) \quad (26.11)$$

This is true for the chain only if  $ka$  is small, since then  $\cos ka \simeq 1$  and  $\omega \simeq \omega_0$ .

Not only does the dispersion relation (26.10) change the speed of waves, it actually limits which frequencies can propagate (have real frequencies  $\omega$ ) on the chain. In order to have real  $k$  solutions,  $\omega$  must lie in the range

$$\omega_0 \leq \omega \leq \omega^* \quad (\text{waves propagation}) \quad (26.12)$$

The minimum frequency  $\omega_0$  and the maximum frequency  $\omega^*$  are related through the limits of  $\cos ka$  in (26.10),

$$(\omega^*)^2 = \omega_0^2 + \frac{4\kappa}{I} \quad (26.13)$$

Waves with  $\omega < \omega_0$  do not propagate, while waves with  $\omega > \omega^*$  are non-physical because they correspond to wavelengths  $\lambda < 2a$ , that is, oscillations where there are no particles. It should be clear that these high and low  $\omega$  cutoffs will change the shape of a propagating pulse.

### 26.2.1

#### Continuum Limit, the Sine-Gordon Equation

If the wavelengths in a pulse are much longer than the repeat distance  $a$ , that is, if  $ka \ll 1$ , the chain can be approximated as a continuous medium. In this limit,  $a$  becomes the continuous variable  $x$ , and the system of coupled ordinary differential equations becomes a single, partial differential equation:

$$\begin{aligned} \theta_{j+1} &\simeq \theta_j + \frac{\partial \theta}{\partial x} \Delta x \\ \Rightarrow (\theta_{j+1} - 2\theta_j + \theta_{j-1}) &\simeq \frac{\partial^2 \theta}{\partial x^2} \Delta x^2 = \frac{\partial^2 \theta}{\partial x^2} a^2 \\ \Rightarrow \frac{\partial^2 \theta}{\partial t^2} - \frac{\kappa a^2}{I} \frac{\partial^2 \theta}{\partial x^2} &= \frac{mgL}{I} \sin \theta \end{aligned} \quad (26.14)$$

If we measure time in units of  $\sqrt{I/mgL}$  and distances in units of  $\sqrt{\kappa a/(mgL)}$ , we obtain the standard form of the sine-Gordon equation (SGE)<sup>1</sup>:

$$\frac{1}{c^2} \frac{\partial^2 \theta}{\partial t^2} - \frac{\partial^2 \theta}{\partial x^2} = \sin \theta \quad (\text{Nonlinear SGE}) \quad (26.15)$$

where the  $\sin \theta$  on the RHS introduces the nonlinear effects.

<sup>1</sup> The name “sine-Gordon” is either a reminder that the SGE is like the Klein–Gordon equation of relativistic quantum mechanics with a  $\sin u$  added to the RHS, or a reminder of how clever one can be in thinking up names.

### 26.3

#### Analytic SGE Solution

The nonlinearity of the sine-Gordon equation (26.15) makes it hard to solve analytically. The trick is to guess a functional form of a traveling wave and that converts the PDE into an ODE:

$$\theta(x, t) \stackrel{?}{=} \theta(\xi = t \pm x/v) \quad \Rightarrow \quad \frac{d^2\theta}{d\xi^2} = \frac{v^2}{v^2 - 1} \sin \theta \quad (26.16)$$

You should recognize (26.16) as old friend, the equation of motion for the realistic pendulum with no driving force and no friction. The constant  $v$  is a velocity in natural units, and separates different regimes of the motion:

$$\begin{aligned} v < 1 : & \text{ pendula initially down } \downarrow\downarrow\downarrow\downarrow \quad (\text{stable}), \\ v > 1 : & \text{ pendula initially up } \uparrow\uparrow\uparrow\uparrow \quad (\text{unstable}) \end{aligned} \quad (26.17)$$

Even though the equation may be familiar, which does not mean that an analytic solution exists. However, for motion along the separatrix ( $E = \pm 1$ ) we obtain the characteristic *soliton* form,

$$\theta(x - vt) = \begin{cases} 4\tan^{-1} \left( \exp \left[ +\frac{x-vt}{\sqrt{1-v^2}} \right] \right), & \text{for } E = 1 \\ 4\tan^{-1} \left( \exp \left[ -\frac{x-vt}{\sqrt{1-v^2}} \right] \right) + \pi, & \text{for } E = -1 \end{cases} \quad (26.18)$$

This soliton corresponds to a solitary *kink* traveling with  $v = -1$  that flips the pendulums around by  $2\pi$  as it moves down the chain. There is also an *antikink* in which the initial  $\theta = \pi$  values are flipped to final  $\theta = -\pi$ .

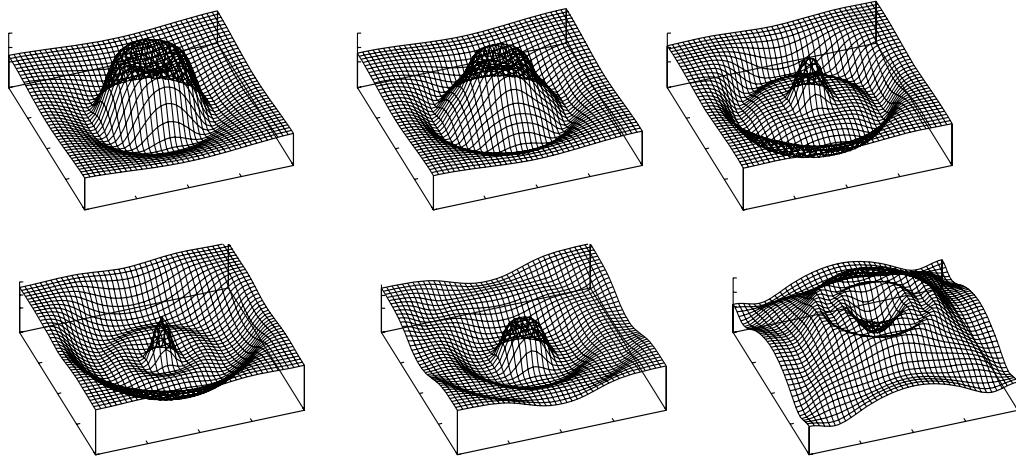
### 26.4

#### Numeric Solution: 2D SGE Solitons

Although we can solve the 1D SGE equation for soliton-like solutions, we will solve for 1D solitons in our study of the KdV equation in Section 26.8, and so here solve for 2D solitons. The 2D solitons occur as solutions of the 2D generalization of the SGE equation (26.15):

$$\frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = \sin u \quad (\text{2D SGE}) \quad (26.19)$$

Whereas the 1D SGE describes wave propagation along a chain of connected pendulums, the 2D form describes wave propagation in nonlinear elastic media. Interestingly enough, the same 2D SGE also occurs in quantum field theory, where the soliton solutions have been suggested as models for elementary particles [75–77]. The idea is that, like elementary particles, the solutions are



**Fig. 26.3** A circular ring soliton at times 8, 20, 40, 60, 80, and 120. This has been proposed as a model for an elementary particle.

confined to a region of space for a long period of time and do not radiate away their energy.

We now have a wave equation containing nonlinear terms to solve. Although we can follow the same procedure used for the linear equation, we want to ensure that any unusual behavior we find arises from the physics and not the algorithm, and so we follow the procedure used in the research literature. We solve in a finite region of 2D space and for positive times:

$$-x_0 < x < x_0 \quad -y_0 < y < y_0 \quad 0 \leq t \quad (26.20)$$

We take  $x_0 = y_0 = 7$  and impose the *boundary conditions* that the derivative of the displacement vanishes at the ends of the region:

$$\frac{\partial u}{\partial x}(-x_0, y, t) = \frac{\partial u}{\partial x}(x_0, y, t) = \frac{\partial u}{\partial y}(x, -y_0, t) = \frac{\partial u}{\partial y}(x, y_0, t) = 0 \quad (26.21)$$

We also impose the *initial condition* that at time  $t = 0$  the waveform is that of a pulse (Fig. 26.3) with its surface at rest:

$$u(x, y, t = 0) = 4 \tan^{-1}(e^{3-\sqrt{x^2+y^2}}) \quad \frac{\partial u}{\partial t}(x, y, t = 0) = 0 \quad (26.22)$$

We discretize the equation first by looking for solutions on a space-time lattice:

$$x = m\Delta x \quad y = l\Delta x \quad t = n\Delta t \quad (26.23)$$

$$u_{m,l}^n \stackrel{\text{def}}{=} u(m\Delta x, l\Delta x, n\Delta t) \quad (26.24)$$

Next we replace the derivatives in (26.19) by their finite-difference approximations to obtain the finite difference SGE:

$$\begin{aligned} u_{m,l}^{n+1} &\simeq -u_{m,l}^{n-1} + 2 \left[ 1 - 2 \left( \frac{\Delta t}{\Delta x} \right)^2 \right] u_{m,l}^n \\ &+ \left( \frac{\Delta t}{\Delta x} \right)^2 \left( u_{m+1,l}^n + u_{m-1,l}^n + u_{m,l+1}^n + u_{m,l-1}^n \right) \\ &- \Delta t^2 \sin \left[ \frac{1}{4} \left( u_{m+1,l}^n + u_{m-1,l}^n + u_{m,l+1}^n + u_{m,l-1}^n \right) \right] \end{aligned} \quad (26.25)$$

To make the algorithm simpler and ensure stability, if we make the time and space steps proportional,  $\Delta t = \Delta x / \sqrt{2}$ , then all the  $u_{m,l}^n$  terms drop out:

$$\begin{aligned} u_{m,l}^2 &\simeq \frac{1}{2} \left( u_{m+1,l}^1 + u_{m-1,l}^1 + u_{m,l+1}^1 + u_{m,l-1}^1 \right) \\ &- \frac{\Delta t^2}{2} \sin \left[ \frac{1}{4} \left( u_{m+1,l}^1 + u_{m-1,l}^1 + u_{m,l+1}^1 + u_{m,l-1}^1 \right) \right] \end{aligned} \quad (26.26)$$

Likewise, the discrete form of vanishing initial velocity (26.22) becomes

$$\partial u(x, y, 0) / \partial t = 0 \Rightarrow u_{m,l}^2 = u_{m,l}^0 \quad (26.27)$$

This will be useful in getting the time propagation started.

The lattice points on the edges and corners cannot be obtained from these relations. They are obtained by applying the boundary conditions (26.21):

$$\frac{\partial u}{\partial z}(x_0, y, t) = \frac{u(x + \Delta x, y, t) - u(x, y, t)}{\Delta x} = 0 \quad (26.28)$$

$$\Rightarrow u_{1,l}^n = u_{2,l}^n \quad (26.29)$$

Similarly, the other derivatives in (26.21) give

$$u_{N_{\max},l}^n = u_{N_{\max}-1,l}^n \quad u_{m,2}^n = u_{m,1}^n \quad u_{m,N_{\max}}^n = u_{m,N_{\max}-1}^n \quad (26.30)$$

where  $N_{\max}$  is the number of grid points used for one space dimension.

## 26.5

### 2D Soliton Implementation

**Listing 26.1:** TwoDsol.java solves the 2D space plus time SGE for 2D solitons.

```
// TwoDsol.java: solves Sine-Gordon equation for 2D soliton

import java.io.*;
import java.util.*;

public class TwoDsol {
    public static int D = 201;
    public static double u[][][] = new double[D + 1][D + 1][4];

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
        int nint;
        // input positive integer proportional to time of wave packet
        Scanner sc = new Scanner(System.in); // Connect Scanner to input
        System.out.printf("Enter a pos integer from 1 (initial time)\n");
        System.out.printf("to 100 for wave packet at that time:\n");
        nint = sc.nextInt(); // Read int
        initial(u); // Initialization
        solution(u, nint); // Solve equation
    }

    public static void initial(double u[][][]) {
        double dx, dy, dt, xx, yy, dts, time, tmp;
        int i, j, k;
        dx = 14./200.;
        dy = dx;
        dt = dx/Math.sqrt(2.);
        dts = (dt/dx)*(dt/dx);
        yy = -7.;
        time = 0.;
        for ( i=0; i <= D-1; i++ ) {
            xx = -7.;
            for ( j=0; j <= D-1; j++ ) {
                tmp = 3.-Math.sqrt(xx*xx + yy*yy);
                u[i][j][0] = 4.*Math.atan(tmp);
                xx = xx + dx;
            }
            yy = yy + dy;
        }
    }

    public static void solution(double u[][][], int nint)
        throws IOException, FileNotFoundException {
        PrintWriter w =
            new PrintWriter(new FileOutputStream("2Dsol.dat"), true);
        double dx, dy, dt, time, a2, zz, dts, a1, tmp;
        int l, m, mn, k, j, i;

        dx = 14./200.; dy = dx;
        dt = dx/Math.sqrt(2.);
        time = 0.;
        time = time + dt;
    }
}
```

```

dts = (dt/dx)*(dt/dx);
tmp = 0.;
for ( m=1; m <= D-2; m++ ) {
    for ( l=1; l <= D-2; l++ ) {
        a2 = u[m+1][1][0]+u[m-1][1][0] + u[m][l+1][0] + u[m][l-1][0];
        tmp = .25*a2;
        u[m][l][1] = 0.5*(dts*a2-dt*dt*Math.sin(tmp));
    }
}
for ( mm=1; mm <= D-2; mm++ ) { // Borders in second iteration
    u[mm][0][1] = u[mm][1][1];
    u[mm][D-1][1] = u[mm][D-2][1];
    u[0][mm][1] = u[1][mm][1];
    u[D-1][mm][1] = u[D-2][mm][1];
}
u[0][0][1] = u[1][0][1]; // Still undefined terms
u[D-1][0][1] = u[D-2][0][1];
u[0][D-1][1] = u[1][D-1][1];
u[D-1][D-1][1] = u[D-2][D-1][1];
tmp = 0.;

for ( k=0; k <= nint; k++ ) { // Following iterations
    for ( m=1; m <= D-2; m++ ) {
        for ( l=1; l <= D-2; l++ ) {
            a1 = u[m+1][1][1]+u[m-1][1][1]+u[m][l+1][1]+u[m][l-1][1];
            tmp = .25*a1;
            u[m][l][2] = -u[m][l][0] + dts*a1-dt*dt*Math.sin(tmp);
            u[m][0][2] = u[m][1][2];
            u[m][D-1][2] = u[m][D-2][2];
        }
    }
    for ( mm=1; mm <= D-2; mm++ ) {
        u[mm][0][2] = u[mm][1][2];
        u[mm][D-1][2] = u[mm][D-2][2];
        u[0][mm][2] = u[1][mm][2];
        u[D-1][mm][2] = u[D-2][mm][2];
    }
    u[0][0][2] = u[1][0][2];
    u[D-1][0][2] = u[D-2][0][2];
    u[0][D-1][2] = u[1][D-1][2];
    u[D-1][D-1][2] = u[D-2][D-1][2];
    for ( l=0; l <= D-1; l++ ) { // New iterations now
        old
        for ( m=0; m <= D-1; m++ ) {
            u[l][m][0] = u[l][m][1];
            u[l][m][1] = u[l][m][2];
        }
    }
    if (k==nint) {
        for ( i=0; i <= D-1; i=i+5) {
            for ( j=0; j <= D-1; j=j+5)
                { w.println(" " + (float) Math.sin(u[i][j][2]/2.) + " ");
                  w.println(" " );
        }
    }
    time = time + dt;
}
}

```

1. Define an array  $u[N_{\max}][N_{\max}[3]$  with  $N_{\max} = 201$  for the space slots and 3 for the time slots.
2. The solution (26.22) for the initial time  $t = 0$  is placed in  $u[m][l][1]$ .
3. The solution for the second time  $\Delta t$  is placed in  $u(m, l, 2)$ , and the solution for the next time,  $2\Delta t$ , is placed in  $u[m][l][3]$ .
4. Assign the constants,  $\Delta x = \Delta y = \frac{7}{100}$ ,  $\Delta t = \Delta x / \sqrt{2}$ ,  $y_0 = x_0 = 7$ .
5. Start off at  $t = 0$  with the initial conditions and impose the boundary conditions to this initial solution. This is the solution for the first time step, defined over the entire  $201 \times 201$  grid.
6. For the second time step, increase time by  $\Delta t$  and use (26.26) for all points in the plane. Do not include the edge points.
7. At the edges, for  $i = 1, 2, \dots, 200$ , set

$$\begin{aligned} u[i][1][2] &= u[i][2][2] & u[i][N_{\max}][2] &= u[i][N_{\max-1}][2] \\ u[1][i][2] &= u[2][i][2] & u[N_{\max}][i][2] &= u[N_{\max-1}][i][2] \end{aligned}$$

8. To find values for the four points in the corners for the second time step, again use initial condition (26.26):

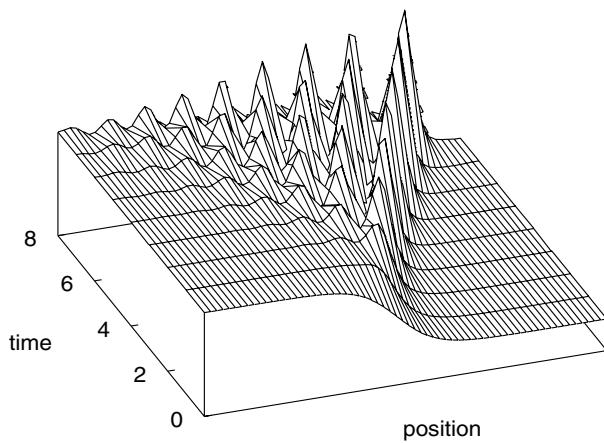
$$\begin{aligned} u[1][1][2] &= u[2][1][2] & u[N_{\max}][N_{\max}][2] &= u[N_{\max-1}][N_{\max-1}][2] \\ u[1][1][N_{\max}] &= u[2][N_{\max}][2] & u[N_{\max}][1][2] &= u[N_{\max-1}][1][2] \end{aligned}$$

9. For the third time step (the future), use (26.26).
10. Continue the propagation forward in time, reassigning the future to the present, and so forth. In this way the solutions for only three time steps need to be stored.

## 26.6

### SGE Soliton Visualization

We see in Fig. 26.3 the time evolution of a circular ring soliton for the stated initial conditions (these results are not critically dependent on the initial conditions). We note that the ring at first shrinks in size, then expands, and then shrinks back into another (but not identical) ring soliton. A small amount of the particle does radiate away, and in the last frame we can notice some interference between the radiation and the boundary conditions. An animation of this sequence can be found on the CD.



**Fig. 26.4** A single two-level waveform at time zero progressively breaks up into eight solitons (labeled) as time increases. The tallest soliton (1) is narrower and faster in its motion to the right.

## 26.7

### Shallow Water (KdV) Solitons ⊖

In this section we look at soliton water waves. In Section 26.2.1 we looked at soliton solutions of the sine-Gordon equation. We have marked this section as optional because the material is more advanced. Nevertheless, we recommend that everyone at least read through this material because it is fascinating and because the computer has been absolutely essential in the discovery and understanding of solitons. In addition, we recommend that you look at some of the soliton animation we have placed in the *Animations* folder on the CD. In recognition of the possible newness of this material to many readers, we give additional background and explanatory materials.

Your **problem** is to discover whether nonlinear and dispersive systems can support waves with “particle-like” properties. In a practical sense, your problem is to determine how a tsunami can form from a sudden change in the level of the ocean floor, and then travel over long distances without dispersion or attenuation until it reeks havoc on a distant shore. While a logical response is that systems with dispersion have solutions that broaden in time and thereby lose their identity, consider Fig. 26.4 and the following experimental observation as a **problem** you need to explain. In 1834, J. Scott Russell observed a phenomenon on the Edinburgh–Glasgow canal [78]:

*I was observing the motion of a boat which was rapidly drawn along a narrow channel by a pair of horses, when the boat suddenly stopped—not so the mass of water in the channel which it had put in motion; it accumulated round the prow of the vessel in a state of violent agitation, then suddenly leaving it behind, rolled forward with great velocity, assuming the form of a large solitary elevation, a rounded, smooth and well-defined heap of water, which*

*continued its course along the channel apparently without change of form or diminution of speed. I followed it on horseback, and overtook it still rolling on at a rate of some eight or nine miles an hour, preserving its original figure some thirty feet long and a foot to a foot and a half in height. Its height gradually diminished, and after a chase of one or two miles I lost it in the windings of the channel. Such, in the month of August 1834, was my first chance interview with that singular and beautiful phenomenon ....*

Russell also noticed that an initial, arbitrary waveform set in motion in the channel evolves into two or more waves that move at different velocities and progressively move apart until they form individual solitary waves. In Fig. 26.4 we see a single step-like wave breaking up into approximately eight solitons (this shows why these eight solitons are considered the normal modes for this nonlinear systems).

Russell went on to produce these solitary waves in a laboratory and empirically deduced that their speed  $c$  is related to the depth  $h$  of the water in the canal and to the amplitude  $A$  of the wave by

$$c^2 = g(h + A) \quad (26.31)$$

where  $g$  is the acceleration due to the gravity. Equation (26.31) implies an effect not found for linear systems, namely, that the waves with greater amplitudes travel faster than those with smaller amplitudes. Notice that this is different from *dispersion* in which waves of different wavelengths have different velocities, but similar to what we have seen with shock waves. The former effect is illustrated in Fig. 26.5, where we see a tall soliton catching up with and passing through a short one.

## 26.8

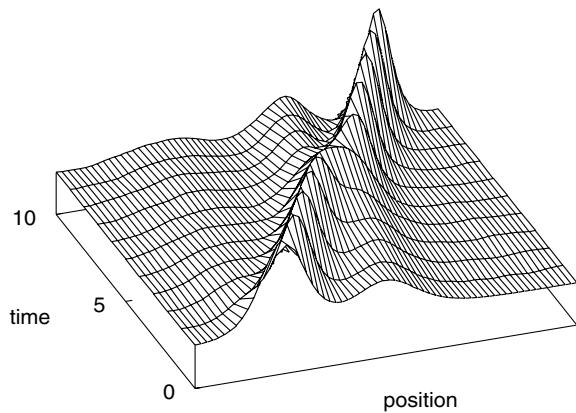
### Theory: The Korteweg–de Vries Equation

We want to understand these unusual water waves that occur in shallow, narrow channels such as canals [24, 39]. The analytic description of this “heap of water” was given by Korteweg and deVries (KdeV) [79] with the partial differential equation:

$$\frac{\partial u(x, t)}{\partial t} + \varepsilon u(x, t) \frac{\partial u(x, t)}{\partial x} + \mu \frac{\partial^3 u(x, t)}{\partial x^3} = 0 \quad (26.32)$$

The nonlinear term,  $\varepsilon u \partial u / \partial t$  leads to a sharpening of the wave and ultimately a *shock* wave. In contrast, the  $\partial^3 u / \partial x^3$  term in (26.32) produces broadening. For the proper parameters and initial conditions, the dispersive broadening exactly balances the nonlinear narrowing, and a stable wave is formed.

KdeV solved (26.32) and proved that the speed given by Russell, (26.31), is in fact correct. Seventy years after its discovery, the KdeV equation was rediscovered by Zabusky and Kruskal [80], who solved it numerically and found



**Fig. 26.5** Two shallow-water solitary waves crossing each other. The taller soliton on the left catches up with and overtakes the shorter one at  $t \simeq 5$ .

that a  $\cos(x/L)$  initial condition broke up into eight solitary waves (Fig. 26.4). They also found that the parts of the wave with larger amplitudes move faster than those with smaller amplitudes, which is why the higher peaks tend to be on the right in Fig. 26.4. As if wonders never cease, Zabusky and Kruskal, who coined the name *soliton* for the solitary wave, also observed that the faster peaks actually passed through the slower one unscathed (Fig. 26.5).

### 26.8.1

#### Analytic Solution: KdV Solitons

The trick in analytic approaches to these types of nonlinear equations is to substitute a guessed solution that has the form of a traveling wave,

$$u(x, t) = u(\xi) \quad (26.33)$$

This form means that if we move with a constant speed  $c$ , we see a constant wave form (yet now the speed will depend on the magnitude of  $u$ ). There is no guarantee that this form of a solution exists, but it is a lucky guess because substitution into the KdV equation produces a solvable ODE:

$$-c \frac{\partial u}{\partial \xi} + \epsilon u \frac{\partial u}{\partial \xi} + \mu \frac{d^3 u}{d \xi^3} = 0 \quad (26.34)$$

While you may find solving this equation for  $u(\xi)$  challenging, mathematicians are good at that sort of thing and have come up with the solution

$$u(x, t) = \frac{-c}{2} \operatorname{sech}^2 \left[ \frac{1}{2} \sqrt{c} (x - ct - \xi_0) \right] \quad (26.35)$$

where  $\xi_0$  is the initial phase. We see in (26.35) an amplitude that is proportional to the wave speed  $c$ , and a  $\text{sech}^2$  function which gives a single lump-like wave. This is a typical analytic form for a soliton.

### 26.8.2

#### Algorithm: KdV Soliton Solution

The KdV equation is solved numerically using a finite difference scheme with the time derivative given by a central difference centered at  $t$ :

$$\frac{\partial u(x, t)}{\partial t} \simeq \frac{u(x, t + \Delta t) - u(x, t - \Delta t)}{2\Delta t}$$

Likewise, the lowest order expansions of  $u(x, t + \Delta t)$  and  $u(x, t - \Delta t)$  give

$$\frac{\partial u}{\partial t} \simeq \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta t} \quad \frac{\partial u}{\partial x} \simeq \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \quad x = i\Delta x, t = j\Delta t$$

To approximate  $\partial^3 u(x, t)/\partial x^3$ , we expand  $u(x, t)$  to  $\mathcal{O}(\Delta t)^3$  about the four points  $u(x \pm 2\Delta x, t)$  and  $u(x \pm \Delta x, t)$ , for example,

$$u(x \pm \Delta x, t) \simeq u(x, t) \pm (\Delta x) \frac{\partial u}{\partial x} + \frac{(\Delta x)^2}{2!} \frac{\partial^2 u}{\partial^2 x} \pm \frac{(\Delta x)^3}{3!} \frac{\partial^3 u}{\partial x^3} \quad (26.36)$$

We solve this for  $\partial^3 u(x, t)/\partial x^3$ . Finally, the factor  $u(x, t)$  in the second term of (26.32) is taken as the average of three  $x$  values all with the same  $t$ :

$$u(x, t) \simeq \frac{u_{i+1,j} + u_{i,j} + u_{i-1,j}}{3} \quad (26.37)$$

These substitutions yield the algorithm for the KdV equation:

$$\begin{aligned} u_{i,j+1} &\simeq u_{i,j-1} - \frac{\epsilon \Delta t}{3 \Delta x} [u_{i+1,j} + u_{i,j} + u_{i-1,j}] \\ &\quad \times [u_{i+1,j} - u_{i-1,j}] - \mu \frac{\Delta t}{(\Delta x)^3} [u_{i+2,j} + 2u_{i-1,j} - 2u_{i+1,j} - u_{i-2,j}] \end{aligned} \quad (26.38)$$

To apply this algorithm to predict future times, we need to know  $u(x, t)$  at present and past times. The initial-time  $u_{i,1}$  solution is known for all positions  $i$  via the initial condition. To find  $u_{i,2}$ , we use a forward difference scheme in which we expand  $u(x, t)$ , keeping only two terms for the time derivative:

$$\begin{aligned} u_{i,2} &\simeq u_{i,1} - \frac{\epsilon \Delta t}{6 \Delta x} [u_{i+1,1} + u_{i,1} + u_{i-1,1}] [u_{i+1,1} - u_{i-1,1}] \\ &\quad - \frac{\mu}{2} \frac{\Delta t}{(\Delta x)^3} [u_{i+2,1} + 2u_{i-1,1} - 2u_{i+1,1} - u_{i-2,1}] \end{aligned} \quad (26.39)$$

The keen observer will note that there are still some undefined columns of points, namely,  $u_{1,j}$ ,  $u_{2,j}$ ,  $u_{N_{\max}-1,j}$ , and  $u_{N_{\max},j}$ , where  $N_{\max}$  is the total number of grid points. A simple technique for determining their values is to assume that  $u_{1,2} = 1$  and  $u_{N_{\max},2} = 0$ . To obtain  $u_{2,2}$  and  $u_{N_{\max}-1,2}$ , assume that  $u_{i+2,2} = u_{i+1,2}$  and  $u_{i-2,2} = u_{i-1,2}$  (avoid  $u_{i+2,2}$  for  $i = N_{\max} - 1$ , and  $u_{i-2,2}$  for  $i = 2$ ). To carry out these steps, approximate (26.39) so that

$$u_{i+2,2} + 2u_{i-1,2} - 2u_{i+1,2} - u_{i-2,2} \rightarrow u_{i-1,2} - u_{i+1,2}$$

The truncation error and stability condition for our algorithm are related by

$$\mathcal{E}(u) = \mathcal{O}[(\Delta t)^3] + \mathcal{O}[\Delta t(\Delta x)^2] \quad \frac{1}{(\Delta x/\Delta t)} \left[ \epsilon|u| + 4 \frac{\mu}{(\Delta x)^2} \right] \leq 1 \quad (26.40)$$

The first equation shows that smaller time and space steps lead to smaller truncation error, yet because the roundoff error increases with more steps, the total error does not necessarily decrease (Chapter 3 on errors). Yet we are also limited in how small the steps can be made by the stability condition, which indicates that making  $\Delta x$  too small always leads to instability. Care and experimentation are clearly required to get the algorithm to work just right.

### 26.8.3

#### Implementation: KdV Solitons

Modify or run the program `Soliton.java` that solves the KdV equation (26.32) for the initial condition:

$$u(x, t = 0) = \frac{1}{2} \left[ 1 - \tanh \left( \frac{x - 25}{5} \right) \right]$$

with parameters  $\epsilon = 0.2$  and  $\mu = 0.1$ . Start with  $\Delta x = 0.4$  and  $\Delta t = 0.1$ . These constants are chosen to satisfy (26.40) with  $|u| = 1$ .

**Listing 26.2:** `Soliton.java` solves the KdV equation for 1D solitons corresponding to a “bore” initial conditions.

```
// Soliton.java: Solves Korteweg-deVries Equation

import java.io.*;

public class Soliton {
    static double ds = 0.4;                                // Delta x
    static double dt = 0.1;                                // Delta t
    static int max = 2000;                                 // Time steps
    static double mu = 0.1;                                // Mu from KdV equation
    static double eps = 0.2;                                // Epsilon from KdV eq

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
```

```

int i, j, k;
double a1, a2, a3, fac, time;
double u[][] = new double[131][3];

PrintWriter w =                                     // Save data in soliton.dat
    new PrintWriter(new FileOutputStream("soliton.dat"), true);
for ( i=0; i < 131; i++ )                      // Initial wave form
{ u[i][0] = 0.5*(1.-(Math.exp(2*(0.2*ds*i - 5.))-1)
                  /(Math.exp(2*(0.2*ds*i - 5.))+ 1))); }
u[0][1]   = 1.;
u[0][2]   = 1.;
u[130][1] = 0.;
u[130][2] = 0.;
fac = mu*dt/(ds*ds*ds);
time = dt;                                         // First time step

for ( i=1; i < 130; i++ ) {
    a1 = eps*dt*(u[i + 1][0] + u[i][0] + u[i-1][0]) / (ds*6.);
    if ((i>1) && (i < 129))
        {a2 = u[i + 2][0] + 2.*u[i-1][0] - 2.*u[i + 1][0]-u[i-2][0];}
    else a2 = u[i-1][0] - u[i + 1][0];
    a3 = u[i + 1][0]-u[i-1][0];
    u[i][1] = u[i][0] - a1*a3 - fac*a2/3.;}
}                                                 // Other time steps

for ( j=1; j < max; j++ ) {
    time += dt;
    for ( i=1; i < 130; i++ ) {
        a1 = eps*dt*(u[i + 1][1] + u[i][1] + u[i-1][1]) / (3.*ds);
        if (i>1 && i < 129) a2 = u[i+2][1] + 2.*u[i-1][1]
                               - 2.*u[i+1][1] - u[i-2][1];
        else a2 = u[i-1][1] - u[i + 1][1];
        a3   = u[i + 1][1] - u[i-1][1];
        u[i][2] = u[i][0] - a1*a3 - 2.*fac*a2/3.;}
    } for (k=0; k < 131; k++) {u[k][0] = u[k][1]; u[k][1] = u[k][2];}
    if ((j%200)==0) {
        for ( k=0; k < 131; k += 2) w.println(" " + u[k][2] + " ");
        w.println( " ");                                // Empty line for gnuplot
    }
}
System.out.println("data stored in soliton.dat");
}
}

```

1. Define a 2D array  $u[131][3]$  with the first index corresponding to the position  $x$  and the second to the time  $t$ . With our choice of parameters, the maximum value for  $x$  is  $130 \times 0.4 = 52$ .
2. Initialize the time to  $t = 0$  and assign values to  $u[i][1]$ .
3. Assign values to  $u[i][2], i=3, 4, \dots, 129$  corresponding to the next time interval. Use (26.39) to advance the time, but note that you cannot start

at  $i = 1$  nor end at  $i = 131$  because (26.39) would include  $u[132][2]$  and  $u[-1][1]$ , which are beyond the limits of the array.

4. Increment the time and assume that  $u[1][2]=1$  and  $u[131][2]=0$ . To obtain  $u[2][2]$  and  $u[130][2]$ , assume that  $u[i+2][2]=u[i+1][2]$  and  $u[i-2][2]=u[i-1][2]$ . Avoid  $u[i+2][2]$  for  $i=130$ , and  $u[i-2][2]$  for  $i=2$ . To do this, approximate (26.39) so that (26.40) is satisfied.
5. Increment time and compute  $u[i][j]$  for  $j=3$  and for  $i=3, 4, \dots, 129$ , using Eq. (26.38). Again follow the same procedures to obtain the missing array elements  $u[2][j]$  and  $u[130][j]$  (set  $u[1][j]=1$  and  $u[131][j]=0$ ). As you print out the numbers during the iterations, you will be convinced that it was a good choice.
6. Set  $u[i][1] = u[i][2]$  and  $u[i][2]=u[i][3]$  for all  $i$ . In this way you are ready to find the next  $u[i][j]$  in terms of the previous two rows.
7. Repeat the previous two steps some 2000 times. Write your solution out to a file after every  $\sim 250$  iterations.
8. Use your favorite graphics tool (we used gnuplot) to plot your results as a 3D graph of disturbance  $u$  versus position and versus time.
9. Observe the wave profile as a function of time and try to confirm Russell's observation that a taller soliton travels faster than a smaller one.

#### 26.8.4

##### **Exploration: Two KdV Solitons Crossing**

Explore what happens when a tall soliton collides with a short one. Do they bounce off each other? Do they go through each other? Do they interfere? Do they destroy each other? Does the tall soliton still move faster than the short one after collision (Fig. 26.5)? Start off by placing a tall soliton of height 0.8 at  $x = 12$ , and a smaller soliton in front of it at  $x = 26$ :

$$u(x, t = 0) = 0.8 \left[ 1 - \tanh^2 \left( \frac{3x}{12} - 3 \right) \right] + 0.3 \left[ 1 - \tanh^2 \left( \frac{4.5x}{26} - 4.5 \right) \right]$$

#### 26.8.5

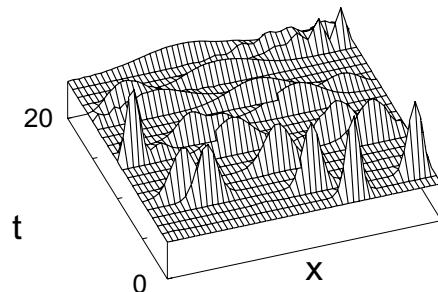
##### **Phase-Space Behavior (Exploration)**

Construct phase-space plots [ $\dot{u}(t)$  versus  $u(t)$ ] of the KdV equation for various parameter values. Note that only very specific sets of parameters produce solitons. In particular, by correlating the behavior of the solutions with your phase-space plots, show that the soliton solutions correspond to the *separatrix* solutions to the KdV equation.

## 27

## Quantum Wave Packets ◉

**Problem:** An electron is initially confined to a 1D region of space the size of an atom. Your **problem** is to determine the behavior in time and space that results. This is different from the problem of a particle confined to a box considered in Chapter 15. There we had a time-independent situation in which we had to solve for the spatial wave function; here we have a time-dependent problem in which the state is not an eigenstate or stationary state of the Hamiltonian, and so does not have a fixed form for its time dependence.



**Fig. 27.1** The position as a function of time of a localized electron confined to a square well. The electron is initially on the right with a Gaussian wave packet. In time, the wave packet spreads out and collides with the walls.

## 27.1

## Time-Dependent Schrödinger Equation (Theory)

We model an electron initially localized in space at  $x = 5$  with momentum  $k_0$  ( $\hbar = 1$ ) by a Gaussian wave function (packet) multiplying a plane wave:

$$\psi(x, t = 0) = \exp \left[ -\frac{1}{2} \left( \frac{x - 5}{\sigma_0} \right)^2 \right] e^{ik_0 x} \quad (27.1)$$

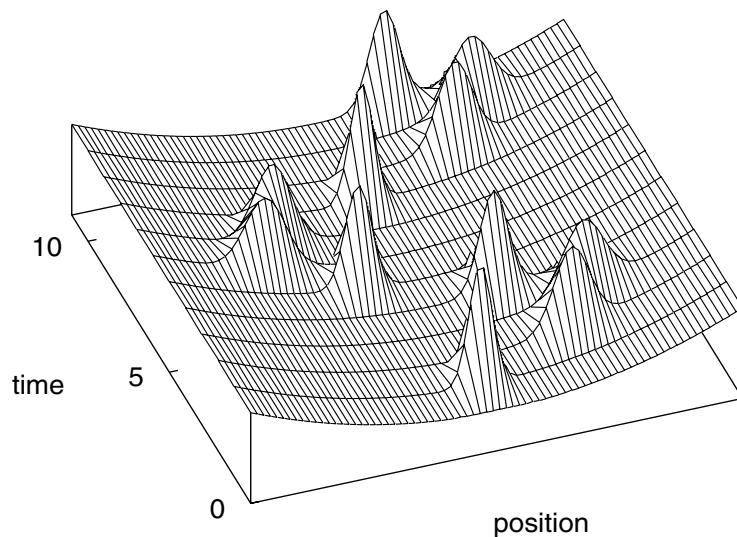
Your **problem** is to determine the wave function for all later times. The behavior of this wave packet when confined to a box is shown in Fig. 27.1, and when confined to an harmonic oscillator potential, in Fig. 27.2.

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5



**Fig. 27.2** The probability density as a function of time for an electron confined to a 1D harmonic oscillator potential well. Because the wave packet is an eigenfunction of the potential, it returns to its original form after transversing the well.

The time and space evolution of a quantum particle is described by the 1D time-dependent Schrödinger equation,

$$i \frac{\partial \psi(x, t)}{\partial t} = \tilde{H}\psi(x, t) = -\frac{1}{2m} \frac{\partial^2 \psi(x, t)}{\partial x^2} + V(x)\psi(x, t) \quad (27.2)$$

where we have set  $2m = 1$  to keep the equations simple. Because the initial wave function is complex (in order to have momentum associated with it), the wave function will be complex for all times. Accordingly, we decompose the wave function into its real and imaginary parts

$$\psi(x, t) = R(x, t) + i I(x, t) \quad (27.3)$$

$$\Rightarrow \quad \frac{\partial R(x, t)}{\partial t} = -\frac{1}{2m} \frac{\partial^2 I(x, t)}{\partial x^2} + V(x)I(x, t) \quad (27.4)$$

$$\frac{\partial I(x, t)}{\partial t} = +\frac{1}{2m} \frac{\partial^2 R(x, t)}{\partial x^2} - V(x)R(x, t) \quad (27.5)$$

where the  $V(x)$  is the potential acting on the particle.

## 27.1.1

**Finite Difference Solution**

The time-dependent Schrödinger equation can be solved with both implicit (large matrix) and explicit (leapfrog) methods. The extra challenge with the Schrödinger equation is to ensure that the integral of the probability density  $\int_{-\infty}^{+\infty} dx \rho(x, t)$  remains constant (conserved) for all time. For our project, we modify the *explicit* method described by [81, 82], which solves the probability problem by determining the real and imaginary parts of the wave function at slightly different or “staggered” times. Explicitly, the real part  $R$  is determined at times  $0, \Delta t, \dots$ , and the imaginary part  $I$  at  $\frac{1}{2}\Delta t, \frac{3}{2}\Delta t$ , and so forth. The algorithm is based on (what else) the Taylor expansions of  $R$  and  $I$ :

$$\begin{aligned} R(x, t + \frac{1}{2}\Delta t) &= R(x, t - \frac{1}{2}\Delta t) + [4\alpha + V(x)\Delta t]I(x, t) \\ &\quad - 2\alpha[I(x + \Delta x, t) + I(x - \Delta x, t)] \end{aligned} \quad (27.6)$$

where  $\alpha = \Delta t / 2(\Delta x)^2$ . In discrete form with  $R_{x=i\Delta x}^{t=n\Delta t}$ , we have

$$R_i^{n+1} = R_i^n - 2 \left\{ \alpha [I_{i+1}^n + I_{i-1}^n] - 2 [\alpha + V_i \Delta t] I_i^n \right\}, \quad (27.7)$$

$$I_i^{n+1} = I_i^n + 2 \left\{ \alpha [R_{i+1}^n + R_{i-1}^n] - 2 [\alpha + V_i \Delta t] R_i^n \right\} \quad (27.8)$$

where the superscript  $n$  indicates the time and the subscript  $i$  the position.

The probability density  $\rho$  is defined in terms of the wave function evaluated at three different times:

$$\rho(t) = \begin{cases} R^2(t) + I(t + \frac{\Delta t}{2})I(t - \frac{\Delta t}{2}), & \text{for integer } t, \\ I^2(t) + R(t + \frac{\Delta t}{2})R(t - \frac{\Delta t}{2}), & \text{for half-integer } t \end{cases} \quad (27.9)$$

Although probability is not conserved exactly with this algorithm, the error is two orders higher than that in the wave function, and this is usually quite satisfactory. If it is not, then we need to use smaller steps. While this definition of  $\rho$  may seem strange, it reduces to the usual one for  $\Delta t \rightarrow 0$ , and so can be viewed as part of the art of numerical analysis. You will investigate just how well probability is conserved. We refer the reader to [28, 82] for details on the stability of the algorithm.

## 27.1.2

**Wave Packet Implementation**

**Listing 27.1:** `Harmos.java` solves the time-dependent Schrödinger equation for a particle described by a Gaussian wave packet moving within a harmonic oscillator potential.

```
// harmos.java: Solution t-dependent Schroedinger equation for
// Gaussian wavepacket in a harmonic oscillator potential
```

```

import java.io.*;

public class harmos {
    public static void main(String[] argv)
        throws IOException, FileNotFoundException {
        PrintWriter w =
            new PrintWriter(new FileOutputStream("Harmos.dat"), true);
        double psr[][] = new double[751][2];
        double psi[][] = new double[751][2];
        double p2[] = new double[751];
        double v[] = new double[751];
        double dx=0.02, k0, dt, x, pi;
        int i, n, max = 750;

        pi = 3.14159265358979323846;
        k0 = 3.0*pi;
        dt = dx*dx/4.0;
        x = -7.5;                                // Initial conditions
        for (i=0; i<max; i++) {
            psr[i][0] = Math.exp(-0.5*(Math.pow((x/0.5),2.0)))
                *Math.cos(k0*x);           // RePsi
            psi[i][0] = Math.exp(-0.5*(Math.pow((x/0.5),2.0)))
                *Math.sin(k0*x);           // ImPsi
            v[i] = 5.0*x*x;
            x = x + dx;                           // Potential
        }
        for (n=0; n<20000; n++) {                  // Propagate in time
            for (i=1; i<max-1; i++) {             // RePsi
                psr[i][1] = psr[i][0] - dt*(psi[i+1][0] + psi[i-1][0]
                    -2.0*psi[i][0])/(dx*dx)+dt*v[i]*psi[i][0];
                p2[i] = psr[i][0]*psr[i][1]+psi[i][0]*psi[i][1];
            }
            for (i=1; i<max-1; i++) {             // ImPsi
                psi[i][1] = psi[i][0] + dt*(psr[i+1][1] + psr[i-1][1]
                    -2.0*psr[i][1])/(dx*dx)-dt*v[i]*psr[i][1];
            }
            if ((n == 0) || (n%2000 == 0)) {       // Output every 2000 steps
                for (i=0;i<max; i=i+10) w.println(""+(p2[i]+0.0015*v[i])+");
                w.println("");
            }
            for (i=0; i<max; i++) {               // New -> old
                psi[i][0] = psi[i][1];
                psr[i][0] = psr[i][1];
            }
        }
        System.out.println("data saved in Harmos.dat");
    }
}

```

On the CD you will find the program `harmon.f` that solves for the motion of the wave packet (27.1) inside harmonic oscillator potential. You should solve for the square-well potential:

$$V(x) = \begin{cases} \infty, & x < 0, \text{ or } x > 15, \\ 0, & 0 \leq x \leq 15. \end{cases}$$

1. Define arrays `R(751,2)` and `I(751,2)` for the real and imaginary parts of the wave function, and `Rho(751)` for the probability density. The first subscript refers to the  $x$  position on the grid and the second to the present and future times.
2. Use the values  $\sigma_0 = 0.5$ ,  $\Delta x = 0.02$ ,  $k_0 = 17\pi$ , and  $\Delta t = \frac{1}{2}\Delta x^2$ .
3. Use Eq. (27.1) for the initial wave packet to define `R(j,1)` for all  $j$ 's at  $t = 0$ , and `I(j,1)` at  $t = \frac{1}{2}\Delta t$ .
4. Set `Rho(1) = Rho(751) = 0.0` because the wave function must vanish at the infinitely high well walls.
5. Increment time by  $\frac{1}{2}\Delta t$ . Use (27.7) to compute `R(j,2)` in terms of `R(j,1)`, and (27.8) to compute `I(j,2)` in terms of `I(j,1)`.
6. Repeat the steps through all of space; that is, for  $i = 2-750$ .
7. Throughout all of space, replace the present wave packet (second index equal to 1) by the future wave packet (second index 2).
8. After you are sure that the program is running properly, repeat the time stepping for  $\sim 5000$  steps.

## 27.1.3

**Wave Packet Visualization and Animation**

1. Output the probability density after every 200 steps for use in animation.
2. Make a surface plot of probability versus position versus time. This should look like Fig. 27.1 or Fig. 27.2.
3. Make an animation showing the wave function as a function of time.
4. Check how well probability is conserved for early and late times by determining the integral of the probability over all of space,  $\int_{-\infty}^{+\infty} dx \rho(x)$ , and seeing by how much it changes in time (its specific value doesn't matter because that's just normalization).
5. What might be a good explanation of why collisions with the walls cause the wave packet to broaden and break up? (*Hint:* The collisions do not appear so disruptive when a Gaussian wave packet is confined within a harmonic oscillator potential well.)

## 27.2

**Wave Packets Confined to Other Wells (Exploration)**

**1D Well :** Consider a wave packet in the harmonic oscillator potential:

$$V(x) = \frac{1}{2}x^2 \quad (-\infty \leq x \leq \infty)$$

Take the initial momentum of the wave packet as  $k_0 = 3\pi$  and time and space steps as  $\Delta x = 0.02$  and  $\Delta t = \frac{1}{4}\Delta x^2$ . Note that the wave packet appears to breathe, yet returns to its initial shape!

**2D Well :** Consider now an electron moving in 2D space (Fig. 27.3). This adds another degree of freedom to the problem, which means that we must solve the 2D time-dependent Schrödinger equation:

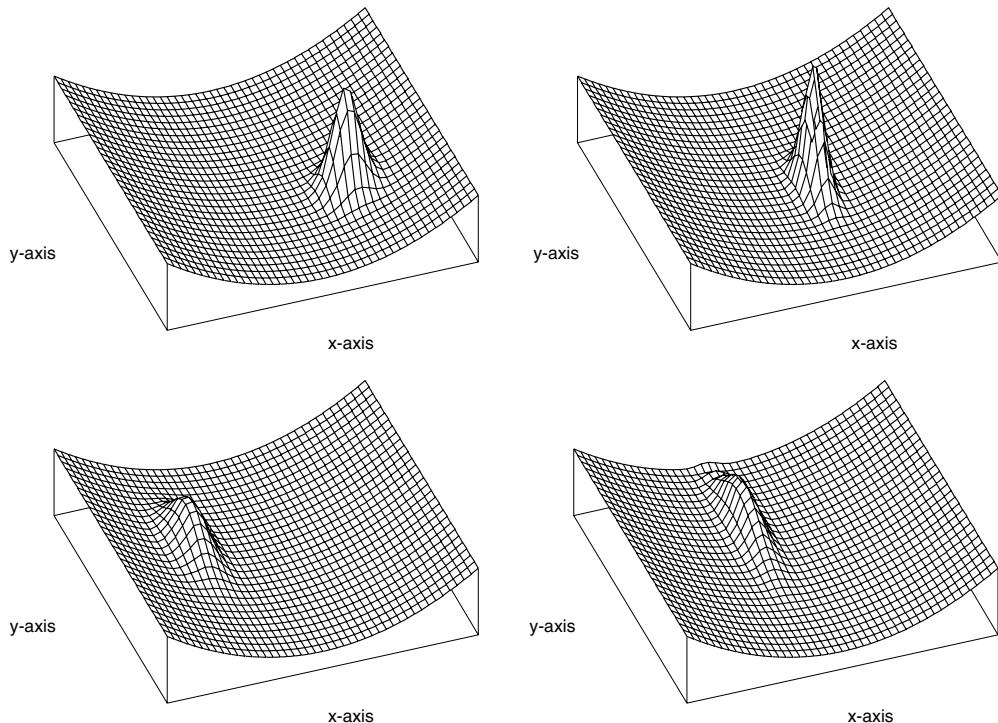
$$i \frac{\partial \psi(x, y, t)}{\partial t} = - \left( \frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} \right) + V(x, y)\psi \quad (27.10)$$

where we have chosen units in which  $2m = \hbar = 1$ . To be more specific, have the electron move in an infinitely long tube with a parabolic cross section:

$$V(x, y) = 0.9x^2 \quad -9.0 \leq x \leq 9.0 \quad 0 \leq y \leq 18.0.$$

Assume that the electron's initial localization is described by a Gaussian wave packet in two dimensions:

$$\psi(x, y, t=0) = e^{ik_{0x}x} e^{ik_{0y}y} \exp \left[ -\frac{(x-x_0)^2}{2\sigma_0^2} \right] \exp \left[ -\frac{(y-y_0)^2}{2\sigma_0^2} \right] \quad (27.11)$$



**Fig. 27.3** The probability density as a function of  $x$  and  $y$  of an electron confined to a 2D parabolic “tube.” The electron’s initial localization is described by a Gaussian wave packet in both the  $x$  and  $y$  directions. The times are 100, 300, and 500.

### 27.2.1

#### Algorithm for 2D Schrödinger Equation

One way to develop an algorithm for solving the time-dependent Schrödinger equation in two dimensions is to extend the 1D algorithm. Rather than do that, we apply quantum theory directly to obtain a more powerful algorithm. First we note that Eq. (27.10) can be integrated in a formal sense [30, p. 4] to obtain the operator solution:

$$\psi(x, y, t) = U(t)\psi(x, y, t = 0) = e^{-i\tilde{H}t}\psi(x, y, t = 0) \quad (27.12)$$

$$\tilde{H} = -\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) + V(x, y) \quad (27.13)$$

From this formal solution we deduce that a wave packet can be moved ahead by a time  $\Delta t$  with the action of the time evolution operator:

$$\psi(t + \Delta t) = U(\Delta t)\psi(t) \quad U(\Delta t) = e^{-i\tilde{H}\Delta t} \quad (27.14)$$

If the operator  $U$  was known exactly, it would provide the exact advance of the solution by one time step:

$$\psi_{i,j}^{n+1} = U(\Delta t) \psi_{i,j}^n \quad \psi_{i,j}^n \stackrel{\text{def}}{=} \psi(i\Delta x, j\Delta y, n\Delta t) \quad (27.15)$$

where the superscripts denote time and the subscripts denote the two spatial variables. Likewise, the inverse of the time evolution operator moves the solution back one time step:

$$\psi^{n-1} = U^{-1}(\Delta t) \psi^n = e^{+i\tilde{H}\Delta t} \psi^n \quad (27.16)$$

While it would be nice to have an algorithm based on a direct application of (27.15), the references show that the resulting algorithm is not stable. That being so, we base our algorithm on an indirect application [81], namely, the relation between the difference in  $\psi^{n+1}$  (27.15) and  $\psi^{n-1}$  (27.16):

$$\psi^{n+1} = \psi^{n-1} + [e^{-i\tilde{H}\Delta t} - e^{i\tilde{H}\Delta t}] \psi^n \quad (27.17)$$

where the difference in sign of the exponents is to be noted. The algorithm derives from combining the  $O(\Delta x^2)$  expression for the second derivative obtained from the Taylor expansion,

$$\frac{\partial^2 \psi}{\partial x^2} \simeq -\frac{1}{2} [\psi_{i+1,j}^n + \psi_{i-1,j}^n - 2\psi_{i,j}^n] \quad (27.18)$$

with the corresponding-order expansion of the evolution equation (27.17). When the resulting expression for the second derivative is substituted into the 2D time-dependent Schrödinger equation, there results<sup>1</sup>

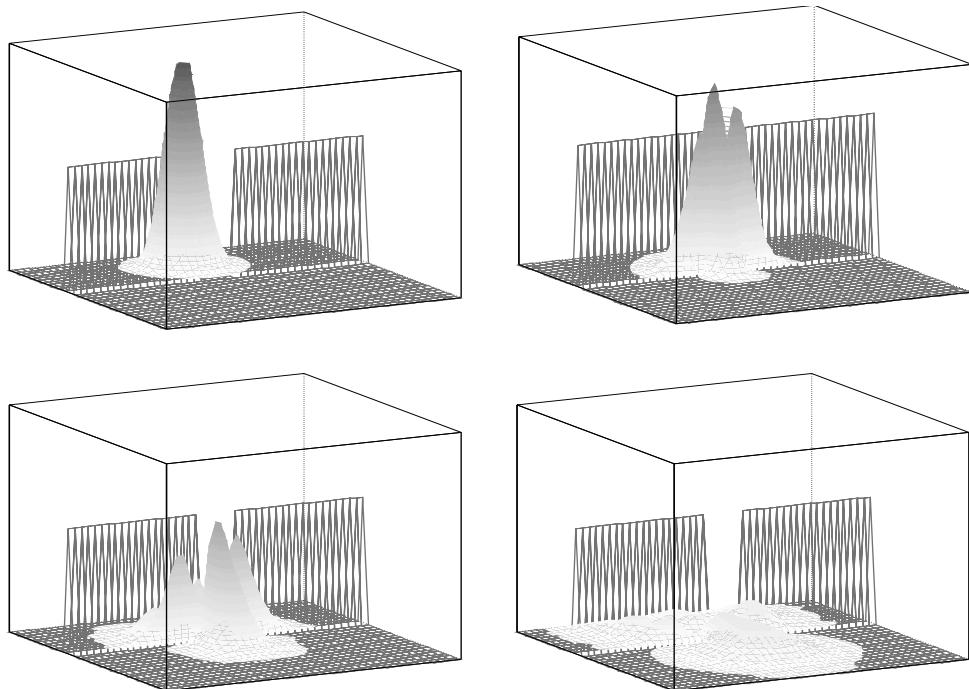
$$\psi_{i,j}^{n+1} = \psi_{i,j}^{n-1} - 2i \left[ (4\alpha + \frac{1}{2}\Delta t V_{i,j}) \psi_{i,j}^n - \alpha (\psi_{i+1,j}^n + \psi_{i-1,j}^n + \psi_{i,j+1}^n + \psi_{i,j-1}^n) \right]$$

where again  $\alpha = \Delta t / 2(\Delta x)^2$ . We convert this complex equations into coupled real equations by substituting in the real and imaginary parts of the wave function,  $\psi = R + iI$ ,

$$\begin{aligned} R_{i,j}^{n+1} &= R_{i,j}^{n-1} + 2 \left[ (4\alpha + \frac{1}{2}\Delta t V_{i,j}) I_{i,j}^n - \alpha (I_{i+1,j}^n + I_{i-1,j}^n + I_{i,j+1}^n + I_{i,j-1}^n) \right] \\ I_{i,j}^{n+1} &= I_{i,j}^{n-1} - 2 \left[ (4\alpha + \frac{1}{2}\Delta t V_{i,j}) R_{i,j}^n + \alpha (R_{i+1,j}^n + R_{i-1,j}^n + R_{i,j+1}^n + R_{i,j-1}^n) \right] \end{aligned} \quad (27.19)$$

This is the algorithm we use to integrate the 2D Schrödinger equation. To determine the probability, we use the same expression (27.9) as used in 1D.

<sup>1</sup> For reference sake, note that the constants in the equation change as the dimension of the equation change; that is, there will be different constants for the 3D equation, and therefore our constants are different from the references!



**Fig. 27.4** The probability density as a function of position and time for an electron incident upon and passing through a slit.

#### 27.2.1.1 Exploration: Bound and Diffracted 2D Packet

- Determine the motion of a 2D Gaussian wave packet within the 2D harmonic oscillator potential:

$$V(x, y) = 0.3(x^2 + y^2) \quad -9.0 \leq x \leq 9.0 \quad -9.0 \leq y \leq 9.0 \quad (27.20)$$

Center the initial wave packet at  $(x, y) = (3.0, -3)$ , and give it momentum  $(k_{0x}, k_{0y}) = (3.0, 1.5)$ .

- Young's single-slit experiment has a wave passing through a small slit, which causes the emerging wavelets to interfere with each other. In quantum mechanics, where we represent a particle by a wave packet, this means that an interference pattern should be formed when a particle passes through a small slit. Consider a Gaussian wave packet of initial width 3 incident on a slit of width 5 (Fig. 27.4). See if you can find quantum interference effects in your solution.

**28****Quantum Paths for Functional Integration**

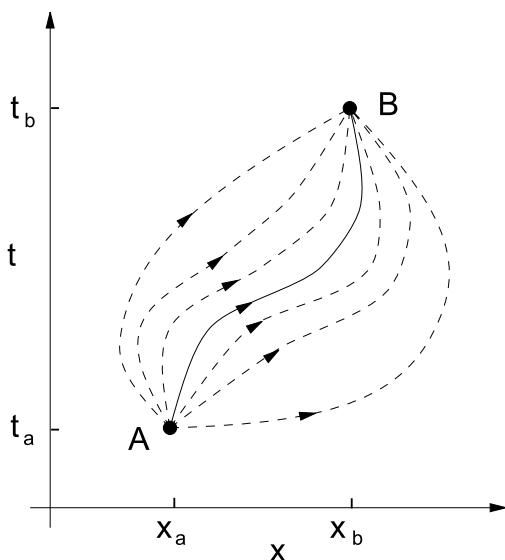
*This chapter deals with Feynman's path integral formulation of quantum mechanics [83]. It is hardest material in this book. In recent times this path integral formulation has been applied to field theory calculations (quantum chromodynamics) and, in the process, has become a major consumer of the world's high-performance computer time. The calculations we present are based on those of other authors [84–86]. Different approaches and further references can be found in an article in Computers in Physics [87].*

**Problem:** As we have seen in this book, and as is known from elementary physics, a classical particle attached to linear spring ( $F \propto x$ ), undergoes simple harmonic motion with position  $x(t) = A \sin(\omega_0 t + \phi)$ . Your **problem** is to take this classical solution or trajectory  $x(t)$ , and make a direct connection between it and the quantum wave function  $\psi(x, t)$  for a particle bound within a harmonic oscillator potential.

**28.1****Feynman's Space–Time Propagation (Theory)**

Feynman was looking for a formulation of quantum mechanics that had a more direct connection to classical mechanics than does Schrödinger theory, and which made its statistical aspects apparent from the start. He followed a suggestion by Dirac that Hamilton's principle may lead to classical mechanics occurring as a special case of quantum mechanics for vanishingly small values of  $\hbar$ . Seeing that Hamilton's principle deals with the paths of particles through space time, Feynman observed that the quantum wave function describing the propagation of a free particle from the space–time point  $a = (x_a, t_a)$  to the point  $b = (x_b, t_b)$  can be expressed as [83]

$$\psi(x_b, t_b) = \int dx_a G(x_b, t_b; x_a, t_a) \psi(x_a, t_a) \quad (28.1)$$



**Fig. 28.1** A collection of paths connecting the initial space–time point  $A$  to the final point  $B$ . The solid line is the trajectory followed by a classical particle while the dashed lines are additional paths sampled by a quantum particle. A classical particle somehow “knows” ahead of time that travel along the classical trajectory minimizes the action  $S$ .

where  $G$  is the space and time dependent *Green's function* or *propagator*

$$G(x_b, t_b; x_a, t_a) \equiv G(b, a) = \sqrt{\frac{m}{2\pi i(t_b - t_a)}} e^{i \frac{m(x_b - x_a)^2}{2(t_b - t_a)}} \quad (28.2)$$

Equation (28.1) is a form of Huygens’s wavelet principle in which each point on the wavefront  $\psi(x_a, t_a)$  emits a spherical wavelet  $G(b, a)$  that propagates forward in space and time. It states that by summation and interference with all the other wavelets, a new wavefront  $\psi(x_b, t_b)$  is created.

Feynman imagined another way of interpreting (28.1) as a form of Hamilton’s principle. It envisions the probability amplitude (wave function  $\psi$ ) for a particle to be at  $b$  as equal to the sum over all *paths* through space–time originating at time  $t_a$  and ending at  $b$  (Fig. 28.1). This view incorporates the statistical nature of quantum mechanics by having different probabilities for travel along the different paths. All paths are possible, but some are more likely than others. [When you realize that Schrödinger theory solves for wave functions and considers paths a classical concept, you appreciate how different a view is Feynman’s.] The values for the probabilities of the paths derives from *Hamilton’s principle of least action* in classical mechanics:

*The most general motion of a physical particle moving along the classical trajectory  $\bar{x}(t)$  from time  $t_a$  to  $t_b$  is along a path such that the action*

$S[\bar{x}(t)]$  is an extremum:

$$\delta S[\bar{x}(t)] = S[\bar{x}(t) + \delta x(t)] - S[\bar{x}(t)] = 0 \quad (28.3)$$

with the paths constrained to pass through the endpoints:

$$\delta(x_a) = \delta(x_b) = 0$$

This formulation of classical mechanics, which is based on the calculus of variations, is equivalent to Newton's differential equations if the action  $S$  is taken as the line integral of the Lagrangian along the path:

$$S[\bar{x}(t)] = \int_{t_a}^{t_b} dt L[x(t), \dot{x}(t)] \quad L = T[x, \dot{x}] - V[x] \quad (28.4)$$

Here  $T$  is the kinetic energy,  $V$  is the potential energy,  $\dot{x} = dx/dt$ , and a square brackets indicate a *functional*<sup>1</sup> of the function  $x(t)$  and  $\dot{x}(t)$ .

Feynman's insight begins with the observation that the classical action for a free particle ( $V = 0$ ),

$$S[b, a] = \frac{m}{2} (\dot{x})^2 (t_b - t_a) = \frac{m}{2} \frac{(x_b - x_a)^2}{t_b - t_a} \quad (28.5)$$

is related to the free-particle propagator (28.2) by

$$G(b, a) = \sqrt{\frac{m}{2\pi i(t_b - t_a)}} e^{iS[b,a]/\hbar} \quad (28.6)$$

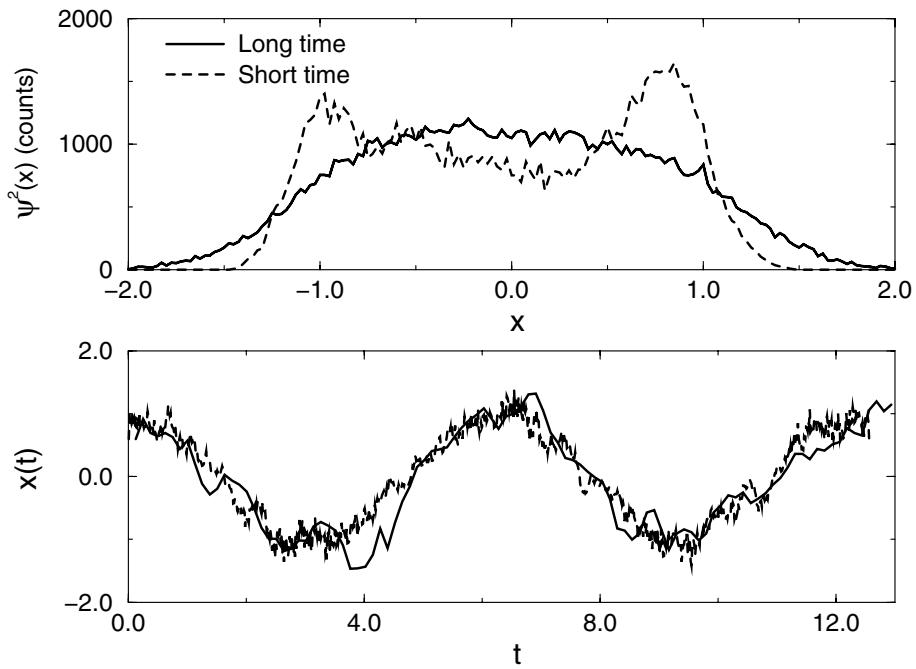
This is the much-sought a connection between quantum mechanics and Hamilton's principle. Feynman then postulated a reformulation of quantum mechanics that incorporated its statistical aspects by expressing  $G(b, a)$  as the weighted sum over all *paths* connecting  $a$  to  $b$ ,

$$G(b, a) = \sum_{\text{paths}} e^{iS[b,a]/\hbar} \quad (\text{Path integral}) \quad (28.7)$$

Here the classical action  $S$  (28.4) is evaluated along different paths (Fig. 28.1), and the exponential of the action is summed over paths. The sum (28.7) is called a *path integral* because it sums over actions  $S[b, a]$ , each of which is an integral (on the computer an integral and sum are the same anyway). The essential connection between classical and quantum mechanics is the realization

<sup>1</sup> A *functional* is a number whose value depends on the complete behavior of some function and not just on its behavior at one point. For example, the derivative  $f'(x)$

depends on the value of  $f$  at  $x$ , yet the integral  $I[f] = \int_a^b dx f(x)$  depends on the entire function and is therefore a functional of  $f$ .



**Fig. 28.2** The ground-state wave function of the harmonic oscillator as determined with a path-integral calculation. *Upper:* The dashed curve is the wave function for a short time  $t_b - t_a$  (twice the classical period) and the solid curve for a long time (20 times the classical period). The long time yields a wave function closer to the Gaussian form expected for the ground state. *Lower:* The long- and short-time trajectories in space–time used last in the solutions for the wave function. The oscillator has initial and final amplitudes of  $x = 1$ ,  $m = k = 1$ , and, consequently, a period of  $T = 2\pi$ .

that in units of  $\hbar \simeq 10^{-34}$  Js, the action is a very large number,  $S/\hbar \geq 10^{20}$ , and so even though all paths enter into the sum (28.7), the main contributions come from those paths adjacent to the classical trajectory  $\bar{x}$ . In fact, because  $S$  is an extremum for the classical trajectory, it is a constant to first order in variation of paths, and so nearby paths have phases that vary smoothly and relatively slowly. In contrast, those paths far from the classical trajectory are weighted by a rapidly oscillating  $\exp(iS/\hbar)$ , and when many are included they tend to cancel each other out. In the classical limit,  $\hbar \rightarrow 0$ , only the classical trajectory contributes and (28.7) becomes Hamilton's principle of least action! In Fig. 28.2 we show concrete examples of the trajectories used in actual path-integral calculations.

## 28.1.1

**Bound-State Wave Function (Theory)**

Although you may be thinking that you have already seen enough expressions for Green's function, there is yet another one we need for our computation. Let us assume that the Hamiltonian  $\tilde{H}$  supports a spectrum of eigenfunctions,  $\tilde{H}\psi_n = E_n\psi_n$ , labeled by the index  $n$ . Because  $\tilde{H}$  is hermitian, the solutions form a complete orthonormal set in which we may expand a general solution:

$$\psi(x, t) = \sum_{n=0}^{\infty} c_n e^{-iE_n t} \psi_n(x) \quad c_n = \int_{-\infty}^{+\infty} dx \psi_n^*(x) \psi(x, t=0) \quad (28.8)$$

where the value for the expansion coefficients  $c_n$  follows from orthonormality of  $\psi_n$ 's. If we substitute this  $c_n$  back into the wave function expansion (28.8), we obtain the identity:

$$\psi(x, t) = \int_{-\infty}^{+\infty} dx_0 \sum_n \psi_n^*(x_0) \psi_n(x) e^{-iE_n t} \psi(x_0, t=0) \quad (28.9)$$

Comparison with (28.1) yields the eigenfunction expansion for  $G$ :

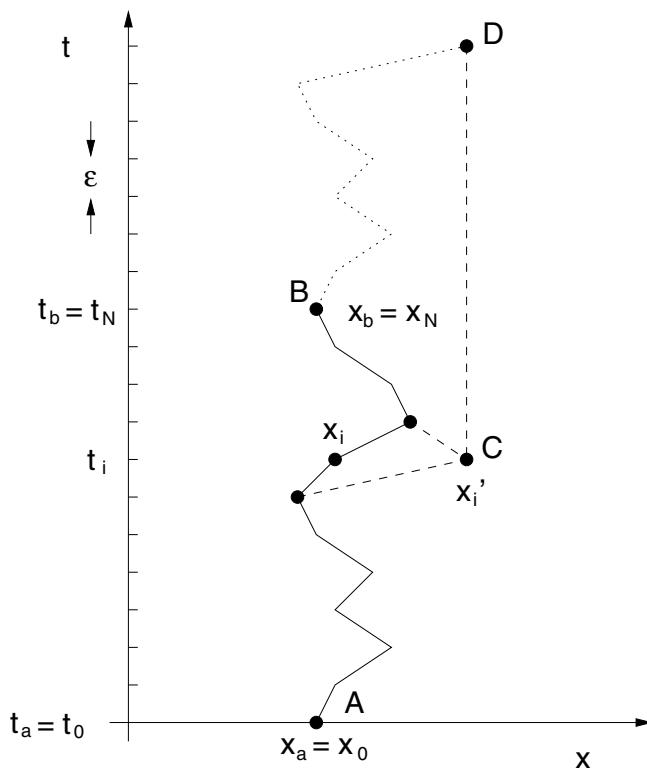
$$G(x, t; x_0, t_0 = 0) = \sum_n \psi_n^*(x_0) \psi_n(x) e^{-iE_n t} \quad (28.10)$$

We relate this to the bound-state wave function (recall our **problem** is to calculate that) by (1) requiring all paths to start and end at the space position  $x_0 = x$ , (2) by taking  $t_0 = 0$ , and (3) by making an analytic continuation of (28.10) to negative imaginary time (permissible for analytic functions):

$$\begin{aligned} G(x, -i\tau; x, 0) &= \sum_n |\psi_n(x)|^2 e^{-E_n \tau} = |\psi_0|^2 e^{-E_0 \tau} + |\psi_1|^2 e^{-E_1 \tau} + \dots \\ \Rightarrow |\psi_0(x)|^2 &= \lim_{\tau \rightarrow \infty} e^{E_0 \tau} G(x, -i\tau; x, 0) \end{aligned} \quad (28.11)$$

The limit here corresponds to long imaginary times  $\tau$ , after which the parts of  $\psi$  with higher energies decay away more quickly, leaving only the ground state  $\psi_0$ .

Equation (28.11) provides a closed-form solution for the ground-state wave function directly in terms of the propagator  $G$ . Although we will soon describe how to compute this equation, look now at Fig. 28.2 showing some results of a computation. We see in the top of the figure that if we wait for only a short imaginary time, then the wave function resembles the classical solution; namely, it has peaks near the classical turning points at the edges of the well. However, if we wait for longer imaginary times, then the wave function resembles the expected Gaussian. The bottom of the figure shows two of the variations on the classical trajectory used in the calculation. Observe that the



**Fig. 28.3** A path through the space–time lattice that starts and ends at  $x_a = x_b$ . The action is an integral over this path, while the *path integral* is a sum of integrals over all paths. The dotted path  $BD$  is a transposed replica of the path  $AC$ .

quantum paths are clearly statistical variations about the classical trajectory  $x(t) = A \sin(\omega_0 t + \phi)$ .

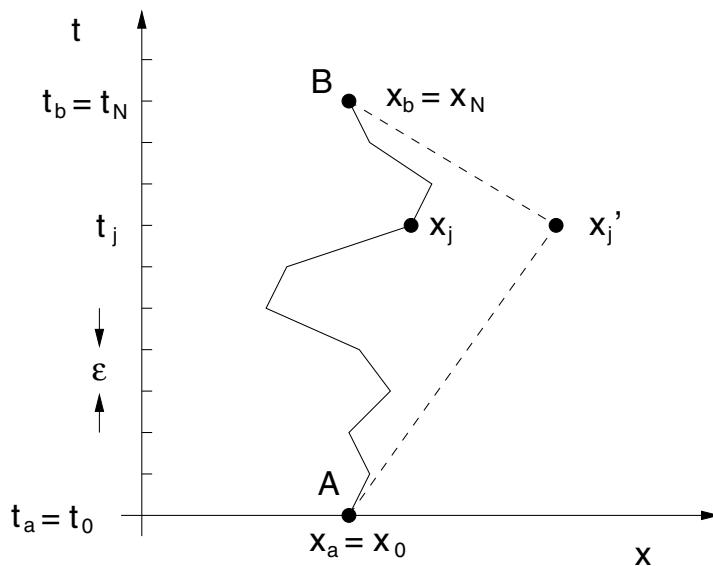
### 28.1.2

#### Lattice Path Integration (Algorithm)

Because both time and space get integrated over when evaluating a path integral, we set up a lattice of discrete points in space–time, and visualize a particle’s trajectory as a series of straight lines connecting one time to the next (Fig. 28.3). We divide the time between  $A$  and  $B$  into  $N$  equal steps of size  $\varepsilon$ , and label them with the index  $j$ :

$$\varepsilon \stackrel{\text{def}}{=} \frac{t_b - t_a}{N} \quad \Rightarrow \quad t_j = t_a + j\varepsilon \quad (j = 0, N) \quad (28.12)$$

Although more precise to use the actual positions  $x(t_j)$  of the trajectory at the times  $t_j$  to determine the  $x_j$ ’s (as in Fig. 28.3), in practice we discretize space



**Fig. 28.4** The dashed path joins the initial and final times in two equal time steps, the solid curve uses  $N$  steps each of size  $\varepsilon$ . The position of the curve at time  $t_j$  defines the position  $x_j$ .

uniformly and have the links end at the nearest regular points. Once we have a lattice, it is easy to evaluate derivatives or integrals on a link<sup>2</sup>:

$$\frac{dx_j}{dt} \simeq \frac{x_j - x_{j-1}}{t_j - t_{j-1}} = \frac{x_j - x_{j-1}}{\varepsilon} \quad (28.13)$$

$$S_j \simeq L_j \Delta t \simeq \frac{1}{2}m \frac{(x_j - x_{j-1})^2}{\varepsilon} - V(x_j)\varepsilon \quad (28.14)$$

where we have assumed that the Lagrangian is constant over each link.

Path integration on a lattice is based on the *composition theorem* for propagators:

$$G(b, a) = \int dx_j G(x_b, t_b; x_j, t_j) G(x_j, t_j; x_a, t_a) \quad (t_a < t_j < t_b). \quad (28.15)$$

<sup>2</sup> Even though Euler's rule has a large error, it is often used in lattice calculations because of its simplicity. However, if the

Lagrangian contains second derivatives, you should use the more precise central difference method to avoid singularities.

For a free particle, this yields

$$\begin{aligned} G(b, a) &= \sqrt{\frac{m}{2\pi i(t_b - t_j)}} \sqrt{\frac{m}{2\pi i(t_j - t_a)}} \int dx_j e^{i(S[b,j] + S[j,a])} \\ &= \sqrt{\frac{m}{2\pi i(t_b - t_a)}} \int dx_j e^{iS[b,a]} \end{aligned} \quad (28.16)$$

where we have added the actions since line integrals combine as  $S[b, j] + S[j, a] = S[b, a]$ . For the  $N$ -linked path of Fig. 28.3, Eq. (28.15) becomes

$$G(b, a) = \int dx_1 \cdots dx_{N-1} e^{iS[b,a]} \quad S[b, a] = \sum_{j=1}^N S_j \quad (28.17)$$

where  $S_j$  is the value of the action for link  $j$ . At this point the integral over the *single* path shown in Fig. 28.3 has become an  $N$ -term sum that becomes an infinite sum as the time step  $\varepsilon$  approaches zero.

To summarize, Feynman's path-integral postulate (28.7) means that we sum over all paths connecting  $a$  to  $b$  to obtain the Green's function  $G(b, a)$ . This means that we must sum not only over the links in one path but *also* over all different paths in order to produce the variation in paths that is required by Hamilton's principle. The sum is constrained such that paths must pass through  $a$  and  $b$  and cannot double back on themselves (causality requires that particles move only forward in time). This is the essence of *path integration*. Because we are integrating over functions as well as along paths, the technique is also known as *functional integration*.

The propagator (28.7) is the sum over all paths connecting  $a$  to  $b$ , with each path weighted by the exponential of the action along that path, explicitly:

$$G(x, t; x_0, t_0) = \int dx_1 dx_2 \cdots dx_{N-1} e^{iS[x,x_0]} \quad (28.18)$$

$$S[x, x_0] = \sum_{j=1}^{N-1} S[x_{j+1}, x_j] \simeq \sum_{j=1}^{N-1} L(x_j, \dot{x}_j) \varepsilon \quad (28.19)$$

where  $L(x_j, \dot{x}_j)$  is the average value of the Lagrangian on link  $j$  corresponding to time  $t = j\varepsilon$ . To keep the computation simple, we assume that the potential  $V(x)$  is independent of velocity and does not depend on other  $x$  values (local potential). Next we observe that in the expression (28.11) for the ground-state wave function,  $G$  is evaluated with a negative imaginary time. Accordingly,

we evaluate the Lagrangian with  $t = -i\tau$ :

$$L(x, \dot{x}) = T - V(x) = +\frac{1}{2}m \left( \frac{dx}{dt} \right)^2 - V(x) \quad (28.20)$$

$$\Rightarrow L\left(x, \frac{dx}{-id\tau}\right) = -\frac{1}{2}m \left( \frac{dx}{d\tau} \right)^2 - V(x) \quad (28.21)$$

We see that the reversal of the sign of the kinetic energy in  $L$  means that  $L$  now equals the negative of the Hamiltonian evaluated at a real positive time  $t = \tau$ :

$$H\left(x, \frac{dx}{d\tau}\right) = \frac{1}{2}m \left( \frac{dx}{d\tau} \right)^2 + V(x) = E \quad (28.22)$$

$$\Rightarrow L\left(x, \frac{dx}{-id\tau}\right) = -H\left(x, \frac{dx}{d\tau}\right) \quad (28.23)$$

In this way we rewrite the  $t$ -path integral of  $L$  as a  $\tau$ -path integral of  $H$ , and so express the action and Green's function in terms of the Hamiltonian:

$$S[j+1, j] = \int_{t_j}^{t_{j+1}} L(x, t) dt = -i \int_{\tau_j}^{\tau_{j+1}} H(x, \tau) d\tau \quad (28.24)$$

$$\Rightarrow G(x, -i\tau; x_0, 0) = \int dx_1 \cdots dx_{N-1} e^{-\int_0^\tau H(\tau') d\tau'} \quad (28.25)$$

where the line integral of  $H$  is over an entire trajectory. Next we express the path integral in terms of an average energy of the particle on each link  $E_j = T_j + V_j$ , and then sum over links<sup>3</sup> to obtain the summed energy  $\mathcal{E}$ :

$$\int H(\tau) d\tau \simeq \sum_j \varepsilon E_j = \varepsilon \mathcal{E}(\{x_j\}) \quad (28.26)$$

$$\mathcal{E}(\{x_j\}) \stackrel{\text{def}}{=} \sum_{j=1}^N \left[ \frac{m}{2} \left( \frac{x_j - x_{j-1}}{\varepsilon} \right)^2 + V\left(\frac{x_j + x_{j-1}}{2}\right) \right] \quad (28.27)$$

In (28.27) we have approximated each path link as a *straight line*, used Euler's derivative rule for the velocity, and evaluated the potential at the midpoint of each link. We now substitute this expression for  $G$  into our solution (28.11)

<sup>3</sup> In some cases, such as for an infinite square well, this can cause problems if the trial link cause the energy to be infinite. In that

case, one can modify the algorithm to use the potential at the beginning of a link.

for the ground-state wave function, which requires that the initial and final points in space be the same:

$$\lim_{\tau \rightarrow \infty} \frac{G(x, -i\tau, x_0 = x, 0)}{\int dx G(x, -i\tau, x_0 = x, 0)} = \frac{\int dx_1 \cdots dx_{N-1} \exp \left[ -\int_0^\tau H dt' \right]}{\int dx dx_1 \cdots dx_{N-1} \exp \left[ -\int_0^\tau H dt' \right]}$$

$$\Rightarrow |\psi_0(x)|^2 = \frac{1}{Z} \lim_{\tau \rightarrow \infty} \int dx_1 \cdots dx_{N-1} e^{-\varepsilon \mathcal{E}} \quad (28.28)$$

$$Z = \lim_{\tau \rightarrow \infty} \int dx dx_1 \cdots dx_{N-1} e^{-\varepsilon \mathcal{E}} \quad (28.29)$$

The similarity of these expressions to thermodynamics, even with a partition function  $Z$ , is no accident; by making the time parameter of quantum mechanics imaginary, we have converted the time-dependent Schrödinger equation into the heat-diffusion equation:

$$i \frac{\partial \psi}{\partial(-i\tau)} = \frac{-\nabla^2}{2m} \psi \quad \rightarrow \quad \frac{\partial \psi}{\partial \tau} = \frac{\nabla^2}{2m} \psi. \quad (28.30)$$

It is not a surprise then that the sum over paths in Green's function has each path weighted by the Boltzmann factor  $\mathcal{P} = e^{-\varepsilon \mathcal{E}}$  that is usually associating with thermodynamics. We make the connection complete by identifying the temperature with the inverse time step:

$$\mathcal{P} = e^{-\varepsilon \mathcal{E}} = e^{-\mathcal{E}/k_B T}, \quad \Rightarrow \quad k_B T = \frac{1}{\varepsilon} \equiv \frac{\hbar}{e}. \quad (28.31)$$

Consequently, the  $\varepsilon \rightarrow 0$  limit, which makes time continuous, is a "high-temperature" limit. The  $\tau \rightarrow \infty$  limit, which is required to project out the ground-state wave function, means that we must integrate over a path that is long in imaginary time, that is, long compared to a typical time  $\hbar / \Delta E$ . Just as our simulation of the Ising model in Chapter 12 required us to wait a long time while the system equilibrated, so the present simulation requires us to wait around a long time so that all but the ground state wave function has decayed away. Alas, the solution to our **problem**.

To summarize, we have expressed Green's function as an evaluation of the path integral (28.28), which requires integration of the Hamiltonian along paths and a summation over all paths. We evaluate this path integral as the sum over all trajectories in our space-time lattice. The links on each path, and correspondingly each trial path, occur with a probability based on its action. We use the Metropolis algorithm to fluctuate the links as if they are in thermal equilibrium and obeying a Boltzmann distribution of energy. This is similar to our work with the Ising model in Chapter 12, however, now, rather than reject or accept a flip in spin based on the change in energy, we reject or accept

a change in a link based on the change in energy. The more iterations we let the algorithm run for, the more the determined wave function equilibrates to the ground state wave function.

In general, Monte Carlo Green's function techniques work best if we start off with a good guess at the correct answer and have the algorithm calculate variations to our guess. For the present problem this means that if we start off with a path in space-time close to the classical trajectory, the algorithm may be expected to do a good job at simulating the quantum fluctuations about the classical trajectory. However, it does not appear to be good at finding the classical trajectory from arbitrary locations in space time. We suspect that the latter arises from  $\delta S/\hbar$  being so large, that  $\exp(\delta S/\hbar)$  fluctuates wildly (essentially averaging out to zero) and so loses its sensitivity.

### 28.1.2.1 A Time Saving Trick

As we have formulated the computation, we pick a value of  $x$  and perform a rather lengthy computation of line integrals over all space and time to obtain  $|\psi_0(x)|^2$  at one  $x$ . To obtain the wave function at another  $x$ , the entire simulation would have to be repeated from scratch. Rather than go through all that trouble again and again, we will compute the entire  $x$  dependence of the wave function in one fell swoop. The trick is to insert a delta function into the probability integral (28.28), thereby fixing the initial position to be  $x_0$ , and then to integrate over all values for  $x_0$ :

$$|\psi_0(x)|^2 = \int dx_1 \cdots dx_N e^{-\varepsilon \mathcal{E}(x, x_1, \dots)} = \int dx_0 \cdots dx_N \delta(x - x_0) e^{-\varepsilon \mathcal{E}(x, x_1, \dots)}. \quad (28.32)$$

This equation expresses the wave function as an average of a delta function over all paths, a procedure that might appear totally inappropriate for numerical computation because there is tremendous error in representing a singular function on a finite-word-length computer. Yet when we simulate the sum over all paths with (28.32), there will always be some  $x$  value for which the integral is nonzero, and we need to only accumulate the solution for various (discrete)  $x$  values to determine  $|\psi_0(x)|^2$  for all  $x$ .

To understand how this works in practise, consider path  $AB$  in Figs. 28.3 and 28.4 for which we have just calculated the summed energy. We next form a new path by having one point on the chain jump to point  $C$  (which changes two links). If we replicate section  $AC$  and use it as the extension  $AD$  to form the top path, we see that the path  $CBD$  has the same summed energy (action) as path  $ACB$  and in this way can be used to determine  $|\psi(x'_j)|^2$ . That being the case, once the system is equilibrated, we determine new values of the wave function at new locations  $x'_j$  by flipping links to new values and calculating

new actions. The more frequently some  $x_j$  is accepted, the greater is the wave function at that point.

**Listing 28.1:** QMC.java solves for the ground state probability distribution via a Feynman path integration using the Metropolis algorithm to simulate variations about the classical trajectory.

```
// QMC.java : Quantum MonteCarlo Feynman path integration

import java.io.*;                                     // Location of PrintWriter
import java.util.*;                                    // Location of Random
import java.lang.*;                                   // Location of Math

public class QMC {

    public static void main(String[] argv)
        throws IOException, FileNotFoundException {

        PrintWriter q = new PrintWriter(                  // File output
            new FileOutputStream("QMC.DAT"), true);
        int N = 100, M = 101, Trials = 25000, seedTrials = 200;
        double path[] = new double[N], xscale = 10.;
        long prop[] = new long[M], seed = 10199435;           // Begin Trials
        for (int count = 0; count < seedTrials*10; count += 10) {
            Random randnum = new Random(seed + count);
            double change = 0., newE = 0., oldE = 0.;           // Initial path
            for (int i=0; i < N; i++) path[i] = 0.;             // Find E of path
            oldE = energy(path);                                // Pick random element
            for (int i=0; i < Trials; i++) {
                int element = randnum.nextInt(N);
                change = 1.8*(0.5 - randnum.nextDouble());       // Change path
                path[element] += change;                         // Find new E
                newE = energy(path);                            // Metropolis algorithm
                if (newE > oldE && Math.exp(-newE + oldE))           // Reject
                    <= randnum.nextDouble() ) path[element]-=change;   // Add probabilities
                for (int j=0; j < N; j++) {
                    element = (int) Math.round((M-1)*(path[j]/xscale + .5));
                    if (element < M && element>=0) prop[element]++;
                }
                oldE = newE;                                         // t loop
            }                                                       // Seed loop
        }                                                       // M loop
        for (int i=0; i < M; i++) q.println(xscale*(i-(M-1)/2)
            + " " + (double)prop[i]/((double)Trials*(double)seedTrials));
        System.out.println(" ");
        System.out.println("QMC Program Complete.");
        System.out.println("Data stored in QMC.DAT");
        System.out.println(" ");
    }

    public static double energy(double path[]) {
```

```

int i = 0;
double sum = 0. ;

for ( i=0; i < path.length -2; i++ )
    { sum += (path[i+1] - path[i])*(path[i+1] - path[i]) ; }
sum += path[i+1]*path[i+1];
return sum;
} } // End class

```

### 28.1.3

#### Lattice Implementation

The program `QMC.java` in List 28.1 evaluates the integral (28.7) by finding the average of the integrand  $\delta(x_0 - x)$  with paths distributed according to the weighting function  $\exp[-\varepsilon\mathcal{E}(x_0, x_1, \dots, x_N)]$ . The physics enters via (28.34), the calculation of the summed energy  $\mathcal{E}(x_0, x_1, \dots, x_N)$ . We evaluate the action integral for the harmonic oscillator potential

$$V(x) = \frac{1}{2}x^2 \quad (28.33)$$

and for a particle of mass  $m = 1$ . A convenient set of natural units is to measure lengths in  $\sqrt{1/m\omega} \equiv \sqrt{\hbar/m\omega} = 1$ , and times in  $1/\omega = 1$ . Correspondingly, the oscillator has a period  $T = 2\pi$ . Figure 28.2 shows results from an application of the Metropolis algorithm. In this computation we started off with an initial path close to the classical trajectory and then examined one-half million variations about this path. All paths are constrained to begin and end at  $x = 1$  (which turns out to be somewhat less than the amplitude of the classical oscillation).

When the time difference  $t_b - t_a$  equals a short time like  $2T$ , the system has not had enough time to equilibrate to its ground state and, as we see in the top of Fig. 28.2, the wave function looks like the probability distribution of an excited state (nearly classical with the probability highest for the particle to be near its turning points where its velocity vanishes). However, when the time difference  $t_b - t_a$  equals the longer time  $20T$ , the system has enough time to decay to its ground state and the wave function looks like the expected Gaussian distribution. In either case, we see in the bottom part of Fig. 28.2 that the trajectory through space-time fluctuates about the classical trajectory. This fluctuation is a consequence of the Metropolis algorithm occasionally going uphill in its search; if you modify the program so that searches go only downhill, the space-time trajectory would be a very smooth trigonometric function (the classical trajectory), but the wave function, which is a measure of the fluctuations about the classical trajectory, would vanish! The explicit steps of the calculation are

1. Construct a time grid of  $N$  time steps of length  $\varepsilon$  (Fig. 28.3). Start at  $t = 0$  and extend to time  $\tau = N\varepsilon$  [this means  $N$  time intervals and  $(N + 1)$  lattice points in time]. Note that time always increases monotonically along a path.
2. Construct a space grid of  $M$  points separated by steps of size  $\delta$ . Use a range of  $x$  values several time larger than the characteristic size or range of the potential being used, and start with  $M \simeq N$ .
3. When calculating the wave function, any  $x$  or  $t$  value falling between lattice points should be assigned to the closest lattice point.
4. Associate a position  $x_j$  with each time  $\tau_j$ , subject to the boundary conditions that the initial and final positions always remain the same,  $x_N = x_0 = x$ .
5. Choose an *arbitrary path* of straight-line links connecting the lattice points. For the most realistic simulation it may be best to start with something close to the classical trajectory, as otherwise the simple numerical procedures may not converge. Note that the  $x$  values for the links of the path may have values that increase, decrease, or remain unchanged (in contrast to time, which always increases).
6. Evaluate the energy  $\mathcal{E}$  by summing the kinetic and potential energies for each link of the path starting at  $j = 0$ :

$$\mathcal{E}(x_0, x_1, \dots, x_N) \simeq \sum_{j=1}^N \left[ \frac{m}{2} \left( \frac{x_j - x_{j-1}}{\varepsilon} \right)^2 + V \left( \frac{x_j + x_{j-1}}{2} \right) \right] \quad (28.34)$$

7. Begin the first of a sequence of repetitive steps in which a random position  $x_j$  associated with time  $t_j$  is changed to the position  $x'_j$  (point C in Fig. 28.4). This changes *two* links in the path.
8. For the coordinate that gets changed, weigh the change with the Boltzmann factor (28.31) by using the Metropolis algorithm.
9. For each lattice point, establish a running sum to represent the value of the wave function squared at that point.
10. After each single-link change (or decision not to change), increase the running sum for the new  $x$  value by 1. After a sufficiently long running time, the sum divided by the number of steps is the simulated value for  $|\psi(x_j)|^2$  at each lattice point  $x_j$ .

11. Repeat the entire link-changing simulation using a different seed for the Metropolis algorithm. The average wave function from a number of intermediate-length runs should be better than that from one very long run.

#### 28.1.4

##### Assessment and Exploration

1. Examine some of the actual space–time paths used in the simulation. Compare those paths to the classical trajectory.
2. For a more continuous picture of the wave function, make the  $x$  lattice spacing smaller; for a more precise value of the wave function at any particular lattice site, sample more points (run longer) and use a smaller time step  $\varepsilon$ .
3. Because there are no sign changes in a ground-state wave function, you can ignore the phase and assume  $\psi(x) = \sqrt{\psi^2(x)}$ , and then estimate the energy via

$$E = \langle \psi | H | \psi \rangle = \frac{\omega}{2} \int_{-\infty}^{+\infty} \psi^*(x) \left( -\frac{d^2}{dx^2} + x^2 \right) \psi(x) dx \quad (28.35)$$

where the space derivative is evaluated numerically.

4. Explore the effect of making  $\hbar$  larger, and thus permitting greater fluctuations around the classical trajectory. Do this by decreasing the value of the exponent in the Boltzmann factor. Determine if this makes the calculation more or less robust in its ability to find the classical trajectory.
5. Test the wave function computation for the gravitational potential

$$V(x) = mg|x|, \quad x(t) = x_0 + v_0 t + \frac{1}{2} g t^2 \quad (28.36)$$

You may want to set the initial positions to be close to the classical trajectory to ensure convergence.

## 29

**Quantum Bound States via Integral Equations**

*The power and accessibility of high-speed computers has changed the view as to which theories and equations in physics are soluble. In Chaps. 15 and 19 we saw how even nonlinear differential equations can be easily solved to give new insight into the physical world. In this chapter we examine how integral equations can be solved for both bound and scattering quantum states. This chapter extends our treatment of the eigenvalue problem, solved as a coordinate ( $r$ ) space differential equation in Chap. 15, to the equivalent problem solved as a homogeneous integral equation in momentum ( $p$ ) space. Chap. 30 examines quantum scattering in momentum space, a more advanced problem than bound states. After these chapters we hope the reader views both integral and differential equations as soluble.*

**Problem:** A projectile particle interacts with the particles in a medium through which the projectile passes (Fig. 29.1). The multiple-particle nature of the interaction leads to a nonlocal potential in which the potential at  $\mathbf{r}$  depends on the wave function at all  $\mathbf{r}'$  values. This changes the interaction term in the Schrödinger equation to [30]:

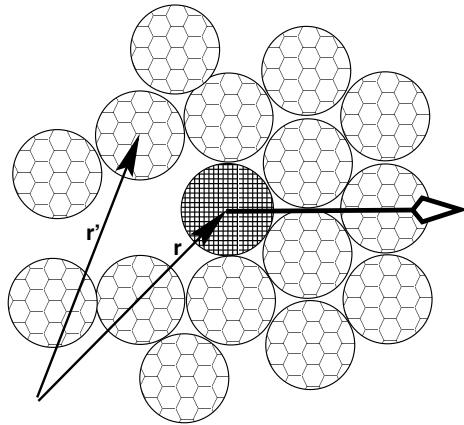
$$V(r)\psi(r) \rightarrow \int dr' V(r, r')\psi(r') \quad (29.1)$$

The integration in (29) leads to a Schrödinger equation that is a combined integral and differential ("integrodifferential") equation:

$$-\frac{1}{2\mu} \frac{d^2\psi(r)}{dr^2} + \int dr' V(r, r')\psi(r') = E\psi(r) \quad (29.2)$$

Your **problem** is to figure out how to find the bound-state energies  $E_n$  and wave functions  $\psi_n$  for the integral equation in (29.2).<sup>1</sup>

<sup>1</sup> We use "natural" units in which Planck's constant  $\hbar \equiv 1$ , and so there is no difference between momenta and wave vectors.



**Fig. 29.1** A particle (dark) moving to the right through a medium with multiple particles present. The nonlocality of the potential felt by the dark particle at  $\mathbf{r}$  arises from the particle interactions at all  $\mathbf{r}'$  with the other particles.

### 29.1 *k*-Space Schrödinger Equation (Theory)

One way of dealing with Eq. (29.2) is by going to momentum space and the partial-wave basis, in which it becomes the integral equation [30]:

$$\frac{k^2}{2\mu}\psi_n(k) + \frac{2}{\pi} \int_0^\infty dp p^2 V(k, p)\psi_n(p) = E_n\psi_n(k) \quad (29.3)$$

Here  $V(k, p)$  is the momentum-space representation (Fourier transform) of the potential,  $\psi_n(k)$  is the momentum-space wave function (the probability amplitude for finding the particle with momentum  $k$ ) and the subscript  $n$  distinguishes solutions of different energies  $E_n$ . To be more specific,  $\psi_n(k)$  is converted to  $\psi_n(r)$  by a Bessel transform,

$$\psi_n(r) = \int_0^\infty dk \psi_n(k) j_l(kr) k^2 \quad (29.4)$$

and the momentum-space potential  $V(k', k)$  is obtained from the  $V(r', r)$  by a double Bessel transform:

$$V(k, p) = (2\pi)^3 \int_0^\infty dr r^2 j_l(kr) V(r) j_l(pr) \quad (\text{local } V). \quad (29.5)$$

The  $j_l(kr)$ 's are spherical Bessel functions,

$$j_0(z) = \frac{\sin z}{z} \quad j_1(z) = \frac{\sin z}{z^2} - \frac{\cos z}{z} \quad \dots \quad (29.6)$$

for which, you may recall, we developed a technique for computing in Chap. 3. For the  $l = 0$  case considered in this chapter, the potential matrix

element is

$$V(k, p) = \frac{(2\pi)^3}{kp} \int_0^\infty dr r^2 \sin(kr) V(r) \sin(pr) \quad (29.7)$$

Equation (29.3) is an integral equation for  $\psi_n(k)$ . It is more than an integral expression for  $\psi_n(k)$  because the integral in it cannot be evaluated until  $\psi_n(p)$  is known for all  $p$ 's; that is, until the equation is solved! Nevertheless, we shall see that we can transform this problem into a matrix eigenvalue problem. The matrix eigenvalue problem is easy to solve on the computer using a mathematical subroutine library.<sup>2</sup>

Those with a mathematical bent may care to observe that the standard formulation of the bound-state problem imposes the boundary condition that the wave function decays exponentially as  $r \rightarrow \infty$ . While the formulation given in this section does not explicitly impose that condition, it assumes that the wave packet is normalizable. The two conditions are equivalent because only an exponentially decaying wave function will be normalizable.

### 29.1.1

#### Integral to Linear Equations (Method)

One technique for solving integral equations is to transform them to linear equations that can be solved as matrix equations. This is also done in our discussion of quantum scattering in Section 30.1.3. We approximate the integral over the potential as a weighted sum over  $N$  integration points (usually Gauss quadrature<sup>3</sup>),  $p = k_j$ ,  $j = 1, N$ :

$$\int_0^\infty dp p^2 V(k, p) \psi_n(p) \simeq \sum_{j=1}^N w_j k_j^2 V(k, k_j) \psi_n(k_j). \quad (29.8)$$

This converts the integral equation (29.3) into the algebraic equation

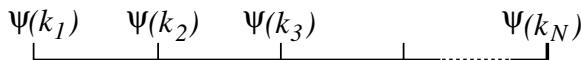
$$\psi_n(k) \frac{k^2}{2\mu} \psi_n(k) + \frac{2}{\pi} \sum_{j=1}^N w_j k_j^2 V(k, k_j) \psi_n(k_j) = E_n \quad (29.9)$$

For a given value of the label  $n$ , (29.9) contains the  $N$  unknowns  $\psi_n(k_j)$ , the single unknown  $E_n$ , and the unknown function  $\psi_n(k)$ . We eliminate the unknown function  $\psi_n(k)$  by evaluating the equation on a grid (Fig. 29.2) composed of the same  $N$   $k_i$  values used to approximate the integral. This leads to the set of  $N$  coupled linear equations in  $N + 1$  unknowns:

$$\frac{k_i^2}{2\mu} \psi_n(k_i) + \frac{2}{\pi} \sum_{j=1}^N w_j k_j^2 V(k_i, k_j) \psi_n(k_j) = E_n \psi_n(k_i) \quad i = 1, N \quad (29.10)$$

<sup>2</sup> The use of libraries is described in Chap. 8

<sup>3</sup> See Chap. 5 for a discussion of numerical integration.



**Fig. 29.2** The grid in momentum space on which the integral equation for the wave function is solved.

As a case in point, if  $N = 2$  we would have the two simultaneous linear equations

$$\begin{aligned}\frac{k_1^2}{2\mu}\psi_n(k_1) + \frac{2}{\pi}w_1k_1^2 V(k_1, k_1)\psi_n(k_1) + w_2k_2^2 V(k_1, k_2) &= E_n\psi_n(k_1) \\ \frac{k_2^2}{2\mu}\psi_n(k_2) + \frac{2}{\pi}w_1k_1^2 V(k_2, k_1)\psi_n(k_1) + w_2k_2^2 V(k_2, k_2)\psi_n(k_2) &= E_n\psi_n(k_2)\end{aligned}$$

We write our coupled dynamical equations in the matrix form as

$$[H][\psi_n] = E_n[\psi_n] \quad (29.11)$$

or as explicit matrices

$$\left( \begin{array}{cccc} \frac{k_1^2}{2\mu} + \frac{2}{\pi}V(k_1, k_1)k_1^2w_1 & \frac{2}{\pi}V(k_1, k_2)k_2^2w_2 & \cdots & \frac{2}{\pi}V(k_1, k_N)k_N^2w_N \\ \frac{2}{\pi}V(k_2, k_1)k_1^2w_1 & \frac{2}{\pi}V(k_2, k_2)k_2^2w_2 + \frac{k_2^2}{2\mu} & \cdots & \\ \ddots & & & \\ \cdots & & \cdots & \frac{k_N^2}{2\mu} + \frac{2}{\pi}V(k_N, k_N)k_N^2w_N \end{array} \right) \times \begin{pmatrix} \psi_n(k_1) \\ \psi_n(k_2) \\ \ddots \\ \psi_n(k_N) \end{pmatrix} = E_n \begin{pmatrix} \psi_n(k_1) \\ \psi_n(k_2) \\ \ddots \\ \psi_n(k_N) \end{pmatrix} \quad (29.12)$$

Equation (29.11) is the matrix representation of the Schrödinger equation (29.3). The wave function  $\psi_n(k)$  on the grid is the  $N \times 1$  vector

$$[\psi_n(k_i)] = \begin{pmatrix} \psi_n(k_1) \\ \psi_n(k_2) \\ \ddots \\ \psi_n(k_N) \end{pmatrix} \quad (29.13)$$

where the subscript  $n$  on  $\psi$  is an energy index.

The acute reader may be questioning the possibility of solving  $N$  equations for  $N + 1$  unknowns. That reader is wise; only sometimes, and only for certain

values of  $E_n$  (eigenvalues) will the computer be able to find solutions. To see how this arises, we try to apply the matrix inversion technique (which we will use successfully for scattering in Unit II of this chapter). We rewrite (29.11) as

$$[H - E_n I][\psi_n] = [0] \quad (29.14)$$

We now multiply both sides of this equation by the inverse of  $[H - E_n I]$  to obtain the formal solution

$$[\psi_n] = [H - E_n I]^{-1}[0] \quad (29.15)$$

This equation tells us that one of two things is happening. If the inverse exists, then we have the *trivial* solution  $\psi_n \equiv 0$ , which is not very interesting. Alternatively, if nontrivial solutions are to exist, then our assumption that the inverse exists must be incorrect. Yet we know from the theory of linear equations that the inverse fails to exist when the determinant vanishes:

$$\det[H - E_n I] = 0 \quad (\text{bound-state condition}) \quad (29.16)$$

Equation (29.16) is the additional equation needed to find unique solutions to the eigenvalue problem. There is, as the case may be, no guarantee that solutions of (29.16) can always be found. When they are found, they correspond to the *eigenvalues* of (29.11) and are the bound-state energies of our physical system.

### 29.1.2

#### Delta-Shell Potential (Model)

To keep things simple, and to have an analytic answer to compare with, we consider the local, delta-shell potential:

$$V(r) = \frac{\lambda}{2\mu} \delta(r - b) \quad (29.17)$$

This would be a good model for an interaction that occurs when two particles are predominantly a fixed distance  $b$  apart. Because (29.17) is a local potential, we use (29.5) to determine its momentum-space representation:

$$V(k', k) = \int_0^\infty r^2 j_l(k'r') \frac{\lambda}{2\mu} \delta(r - b) j_l(kr) dr = \frac{\lambda b^2}{2\mu} j_l(k'b) j_l(kb) \quad (29.18)$$

where the subscript  $l$  indicates the angular momentum state for which we are solving the problem. (*Beware:* The potential  $V(k', k)$  oscillates too much to be well behaved. To obtain stable answers, you need to look at the functional dependence of the integrand, and distribute your integration points on that account.)

If the energy is parameterized in terms of a wave vector  $\kappa$  by  $E = -\kappa^2/2\mu$ , then for this potential there is, at most, one bound state for each value of the angular momentum  $l$ , and it satisfies the transcendental equation [8]

$$1 - \frac{\lambda}{i\kappa} (ikb)^2 j_l(ikb) [n_l(ikb) - ij_l(ikb)] = 0 \quad (29.19)$$

For  $l = 0$ , this takes the simple form

$$e^{-2\kappa b} - 1 = \frac{2\kappa}{\lambda} \quad (l = 0) \quad (29.20)$$

Note, bound states occur only for attractive potentials, and even then, only if the attraction is strong enough. For the present case this means that we must have  $\lambda < 0$ .

**Exercise:** Pick some values of  $b$  and  $\lambda$ , and use them to verify with a hand calculation that (29.20) can be solved for  $\kappa$ .

### 29.1.3

#### Binding Energies Implementation

In List 29.1 we present our solution of the integral equation for bound states of the delta-shell potential using the JAMA matrix library and the `gauss` method for Gaussian quadrature points and weights. An actual computation may follow two paths. The first would call subroutines to evaluate the determinant of the  $[H - E_n I]$  matrix in (29.16) and then to *search* for those values of energy for which the computed determinant vanishes. This provides  $E_n$ , but not wave functions. The other approach would call an eigenproblem solver that may give some or all eigenvalues and eigenfunctions. In both cases, the solution is obtained iteratively, and you may be required to guess starting values for both the eigenvalues and eigenvectors.

1. Write your own, or modify the code on the CD, so that you can solve the integral equation (29.11) for the delta-shell potential (29.18). You can do this either by evaluating the determinant of  $[H - E_n I]$  and then finding the  $E$  for which the determinant vanishes, *or* by finding the eigenvalues and eigenvectors for this  $H$ .
2. Set the scale by setting  $2\mu = 1$  and  $b = 10$ .
3. Set up the potential and Hamiltonian matrices  $V(i, j)$  and  $H(i, j)$  for Gaussian quadrature integration with at least  $N = 16$  grid points.
4. Adjust the value and sign of  $\lambda$  to find the  $l = 0$  bound state. A good approach is to start with a large negative value for  $\lambda$ , and then make it less negative. You should find the eigenvalue moves up in energy.

5. Try increasing the number of grid points in steps of 8, for example; 16, 24, 32, 64, . . . , and see how the energy changes.
6. *Note:* Your eigenenergy solver may return several eigenenergies. Only the true bound state will be at negative energy and will be stable as the number of grid points change.
7. Extract the best value for the bound-state energy and estimate its precision by seeing how it changes with the number of grid points.
8. Check your solution by comparing the RHS and LHS in the matrix multiplication  $[H][\psi] = E[\psi]$ .
9. Verify that, regardless of the potential's strength, there is only a single bound state.

#### 29.1.4

##### Wave Function (Exploration)

1. Determine the momentum–space wave function  $\psi_n(k)$ . Does it fall off at  $k \rightarrow \infty$ ? Does it oscillate? Is it well behaved at the origin?
2. Determine the coordinate–space wave function via the Bessel transform

$$\psi_0(r) = \int_0^\infty dk \psi_0(k) j_0(kr) k^2 \quad (29.21)$$

Does  $\psi_0(r)$  fall off like you would expect for a bound state? Does it oscillate? Is it well behaved at the origin?

3. Compare the  $r$  dependence of this  $\psi_0(r)$  to the analytic wave function:

$$\psi_0(r) \propto \begin{cases} e^{-kr} - e^{kr} & \text{for } r < b \\ e^{-kr} & \text{for } r > b \end{cases} \quad (29.22)$$

4. Deduce the energy of the  $l = 1$  bound state.

**Listing 29.1:** `Bound.java` solves the Lippmann–Schwinger integral equation for bound states within a delta-shell potential. The integral equations are converted to matrix equations using Gaussian grid points, and they are solved with JAMA.

```
// Bound.java: Bound states in p space for delta shell potential
// uses JAMA and includes Gaussian integration

import Jama.*;

public class Bound {
    static double min = 0., max = 200., u = 0.5, b = 10.; // Class vars
```

```

public static void main(String[] args) {

    System.out.println("M, lambda, eigenvalue");
    for ( int M = 16; M <= 128; M += 8) {
        for ( int lambda = -1024; lambda < 0; lambda /= 2) {
            double A[][] = new double[M][M], // Hamiltonian
                  WR[] = new double[M], VR, // RE eigenvalues, potential
                  k[] = new double[M], w[] = new double[M]; // Pts & wts
                                                 // Call gauss integration
            gauss(M, min, max, k, w); // Set Hamiltonian
            for ( int i=0; i < M; i++ )
                for ( int j=0; j < M; j++ ) {
                    VR=lambda/2*u*Math.sin(k[i]*b)/k[i]*Math.sin(k[j]*b)/k[j];
                    A[i][j] = 2/Math.PI*VR*k[j]*k[j]*w[j];
                    if ( i == j ) A[i][j] += k[i]*k[i]/2/u;
                }
            // Eigenvalue decomposition
            EigenvalueDecomposition E
            = new EigenvalueDecomposition(new Matrix(A));
            WR = E.getRealEigenvalues(); // RE eigenvalues
            // Matrix V = E.getV(); // Eigenvectors
            for ( int j=0; j < M; j++ ) if (WR[j] < 0) {
                System.out.println(M + " " + lambda + " " + WR[j]);
                break;
            } } } }

// Method gauss: pts & wts for Gauss quadrature, uniform [a, b]
private static void
gauss(int npts, double a, double b, double[] x, double[] w) {
    int m = (npts + 1)/2;
    double t, t1, pp = 0, p1, p2, p3, eps = 3.e-10;
                                                 // eps = accuracy TO CHANGE
    for ( int i=1; i <= m; i++ ) {
        t = Math.cos(Math.PI*(i-0.25)/(npts + 0.5)); t1 = 1;
        while((Math.abs(t-t1))>=eps) {
            p1 = 1.; p2 = 0.;
            for ( int j=1; j <= npts; j++ )
                {p3 = p2; p2 = p1; p1 = ((2*j-1)*t*p2-(j-1)*p3)/j;}
            pp = npts*(t*p1-p2)/(t*t-1);
            t1 = t; t = t1 - p1/pp;
        }
        x[i-1] = -t; x[npts-i] = t;
        w[i-1] = 2./((1-t*t)*pp*pp); w[npts-i] = w[i-1];
    }
    for ( int i=0; i < npts ; i++ ) {
        x[i] = x[i]*(b-a)/2. + (b + a)/2. ;
        w[i] = w[i]*(b-a)/2. ;
    } } }

```

**30****Quantum Scattering via Integral Equations**

*In this chapter we develop techniques to solve the singular, inhomogeneous integral equation appropriate for quantum scattering, a different and more advanced problem than that in Chap. 29. After these two chapters we hope the reader views both integral and differential equations as soluble.*

**Problem:** The problem is essentially the same problem of a particle interacting with a nonlocal potential discussed in Chap. 29 (Fig. 29.1), only now we need to deduce the scattering (Fig. 30.1) that occurs when a particle passes through a dense medium.

**30.1****Lippmann–Schwinger Equation (Theory)**

Because scattering experiments measure scattering amplitudes, it is convenient to convert the Schrödinger equation into an equation dealing with amplitudes rather than wave functions. An integral form of the Schrödinger equation dealing with the amplitude  $R$  (reaction matrix) is the *Lippmann–Schwinger equation*:<sup>1</sup>

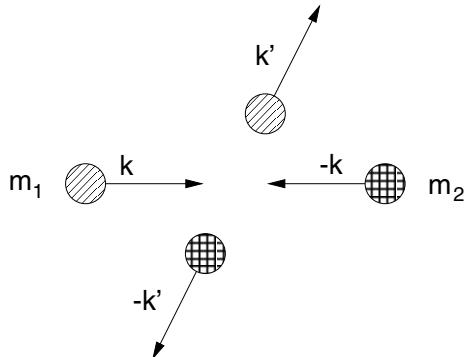
$$R(k', k) = V(k', k) + \frac{2}{\pi} \mathcal{P} \int_0^\infty dp \frac{p^2 V(k', p) R(p, k)}{(k_0^2 - p^2)/2\mu} \quad (30.1)$$

Note that Eq. (30.1) requires more than just an integral to evaluate. It is an integral equation in which  $R(p, k)$  is integrated over, yet since  $R(p, k)$  is unknown, the integral cannot be evaluated until after the equation is solved!

The symbol  $\mathcal{P}$  in (30.1) indicates the Cauchy principal-value prescription for avoiding the singularity arising from the zero of the denominator (we discuss how to do that in Section 30.1.1). This equation describes the scattering of two particles with reduced mass and center-of-mass energy (Fig. 30.1):

$$\mu = \frac{m_1 m_2}{m_1 + m_2} \quad E = \frac{k_0^2}{2\mu} \quad (30.2)$$

<sup>1</sup> To keep the presentation simple, our equations are given in the partial-wave basis but without the  $l$  subscripts to indicate it.



**Fig. 30.1** The scattering of mass  $m_1$  and  $m_2$  in their center-of-momentum system.

and initial and final center-of-mass momenta  $k$  and  $k'$ . The diagonal matrix element  $R(k_0, k_0)$  is the experimental scattering amplitude needed to solve your **problem**.

### 30.1.1

#### Singular Integrals (Mathematics)

A *singular* integral

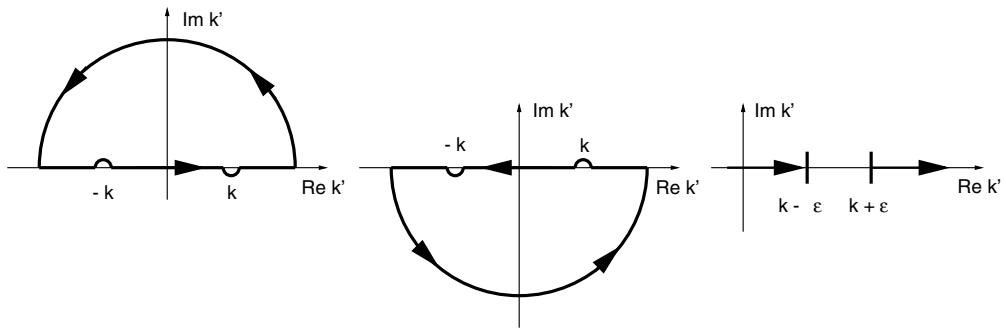
$$\mathcal{G} = \int_a^b g(k) dk \quad (30.3)$$

is one in which the integrand  $g(k)$  is singular at a point  $k_0$  within the interval  $[a, b]$ , and yet the integral  $\mathcal{G}$  is finite. (If the integral itself was infinite, we could not compute it.) Unfortunately, computers are notoriously bad at dealing with infinite numbers, and if an integration point gets too near to the singularity, severe subtractive cancellation or overflow occurs. Consequently, we apply some results from complex analysis before evaluating singular integrals numerically.<sup>2</sup>

In Fig. 30.2 we show three ways in which the singularity of an integrand can be avoided. The paths in A and B move the singularity slightly off the real  $k$  axis by giving the singularity a small imaginary part  $\pm ie$ . The Cauchy principal-value prescription  $\mathcal{P}$  (Fig. 30.2 right) is seen to “pinch” both sides of the singularity at  $k_0$ , but not to pass through it:

$$\mathcal{P} \int_{-\infty}^{+\infty} f(k) dk = \lim_{\epsilon \rightarrow 0} \left[ \int_{-\infty}^{k_0 - \epsilon} f(k) dk + \int_{k_0 + \epsilon}^{+\infty} f(k) dk \right] \quad (30.4)$$

<sup>2</sup> Ref. [88] describes a different approach using *Maple* and *Mathematica*.



**Fig. 30.2** Three different paths in the complex  $k'$  plane used to evaluate line integrals when there are singularities. Here the singularities are at  $k$  and  $-k$ , and the integration variable is  $k'$ .

The preceding three prescriptions are related by

$$\int_{-\infty}^{+\infty} \frac{f(k)dk}{k - k_0 \pm i\epsilon} = \mathcal{P} \int_{-\infty}^{+\infty} \frac{f(k)dk'}{k - k_0} \mp i\pi f(k_0) \quad (30.5)$$

which follows from Cauchy's residue theorem and some contour distortions.

### 30.1.2

#### Numerical Principal Values

A numerical principal value limit (30.4) is awkward because computers have limited precision. A better prescription for computers follows from the calculus relation

$$\mathcal{P} \int_{-\infty}^{+\infty} \frac{dk}{k - k_0} = 0 \quad (30.6)$$

This equation means that the curve of  $1/(k - k_0)$  as a function of  $k$  has equal and opposite areas on both sides of the singular point  $k_0$ . If we break the integral up into one over positive  $k$  and one over negative  $k$ , a change of variable  $k \rightarrow -k$  permits us to express (30.6) as

$$\mathcal{P} \int_0^{+\infty} \frac{dk}{k^2 - k_0^2} = 0 \quad (30.7)$$

We observe that the principal-value exclusion of the singular point's contribution is equivalent to a simple subtraction of the zero integral (30.7):

$$\mathcal{P} \int_0^{+\infty} \frac{f(k)dk}{k^2 - k_0^2} = \int_0^{+\infty} \frac{[f(k) - f(k_0)]dk}{k^2 - k_0^2} \quad (30.8)$$

We notice that there is no  $\mathcal{P}$  on the RHS of (30.8) because the integrand is no longer singular at  $k = k_0$  (it is proportional to the  $df/dk$ ) and can therefore be

$$\underbrace{R(k_1) \quad R(k_2) \quad R(k_3)}_{\dots} \quad R(k_N)$$

**Fig. 30.3** The grid in momentum space on which the integral equation for the  $R$  is solved.

evaluated numerically as can any other integral! The integral (30.8) is called the *Hilbert transform* of  $f$  and also arises in inverse problems.

### 30.1.3

#### Reducing Integral to Matrix Equations (Method)

Now that we know how to handle singular integrals, we return to our problem of a singular integral equation. We want to reduce the integral equation into a set of linear equations that are then solved with matrix operations. We need to solve the integral equation (30.1) with the potential (30.22). The momentum<sup>3</sup>  $k_0$  is related to the energy  $E$  and the reduced mass  $\mu$  by (30.2). The experimental observable that results from a solution of (30.1) is the amplitude  $R(k_0, k_0)$ , or equivalently, the scattering phase shift  $\delta_l$ :

$$R(k_0, k_0) = -\frac{\tan \delta_l}{\rho} \quad \rho = 2\mu k_0 \quad (30.9)$$

The procedure for the computer solution of (30.1) uses (30.8) to rewrite the principal-value prescription as a definite integral [89]:

$$R(k', k) = V(k', k) + \frac{2}{\pi} \int_0^\infty dp \frac{p^2 V(k', p) R(p, k) - k_0^2 V(k', k_0) R(k_0, k)}{(k_0^2 - p^2)/2\mu} \quad (30.10)$$

We convert this integral equation into linear equations by approximating the integral as a sum over  $N$  integration points (usually Gauss quadrature)  $\{k_j; j = 1, N\}$  with weights  $w_j$ :

$$\begin{aligned} R(k, k_0) &\simeq V(k, k_0) + \frac{2}{\pi} \sum_{j=1}^N \frac{k_j^2 V(k, k_j) R(k_j, k_0) w_j}{(k_0^2 - k_j^2)/2\mu} \\ &\quad - \frac{2}{\pi} k_0^2 V(k, k_0) R(k_0, k_0) \sum_{m=1}^N \frac{w_m}{(k_0^2 - k_m^2)/2\mu} \end{aligned} \quad (30.11)$$

We note that the last term in (30.11) implements the principal-value prescription and cancels the singular behavior of the previous term.

Equation (30.11) contains the  $N + 1$  unknowns  $R(k_j, k_0)$  for  $j = 1, N$ , and  $R(k_0, k_0)$ . We turn it into  $N + 1$  simultaneous equations by evaluating it for

<sup>3</sup> We are formulating this problem with “natural” units in which Planck’s constant  $\hbar \equiv 1$ . This means that there is no difference between momentum and wave vectors.

$N + 1$   $k$  values on a grid consisting of the observable momentum  $k_0$  and the integration points (Fig. 30.3):

$$k = k_i = \begin{cases} k_j, & j = 1, N \quad (\text{quadrature points}), \\ k_0, & i = 0 \quad (\text{observable point}). \end{cases} \quad (30.12)$$

There are now  $N + 1$  linear equations for  $N + 1$  unknowns  $R_i \stackrel{\text{def}}{=} R(k_i, k_0)$ :

$$R_i = V_i + \frac{2}{\pi} \sum_{j=1}^N \frac{k_j^2 V_{ij} R_j w_j}{(k_0^2 - k_j^2)/2\mu} - \frac{2}{\pi} k_0^2 V_{ii} R_0 \sum_{m=1}^N \frac{w_m}{(k_0^2 - k_m^2)/2\mu} \quad (30.13)$$

We express these equations in the matrix form  $\mathbf{Ax} = \mathbf{b}$  by combining the denominators and weights into a single denominator vector  $D$ :

$$D_i = \begin{cases} +\frac{2}{\pi} \frac{w_i k_i^2}{(k_0^2 - k_i^2)/2\mu} & \text{for } i = 1, N, \\ -\frac{2}{\pi} \sum_{j=1}^N \frac{w_j k_0^2}{(k_0^2 - k_j^2)/2\mu} & \text{for } i = N + 1 \end{cases} \quad (30.14)$$

The linear equations (30.13) now assume that the matrix form

$$R - DVR = [1 - DV] R = V \quad (30.15)$$

where  $R$  and  $V$  are *column vectors* of length  $N_1 \equiv N + 1$ :

$$[R] = \begin{pmatrix} R_{1,N_1} \\ R_{2,N_1} \\ \ddots \\ R_{N_1,N_1} \end{pmatrix} \quad [V] = \begin{pmatrix} V_{1,N_1} \\ V_{2,N_1} \\ \ddots \\ V_{N_1,N_1} \end{pmatrix} \quad (30.16)$$

We call the matrix  $[1 - Dv]$  in (30.15) the wave matrix  $F$ , and write the integral equation as the matrix equation:

$$[F][R] = [V] \quad F_{ij} = \delta_{ij} - D_j V_{ij} \quad (30.17)$$

With  $R$  the unknown vector, (30.17) is in the standard form  $AX = B$ , which can be solved by the mathematical subroutine libraries discussed in Chap. 8.

### 30.1.4

#### Solution via Inversion, Elimination

An elegant (but alas not efficient) solution to (30.17) is by matrix inversion:

$$[R] = [F]^{-1}[V] \quad (30.18)$$

Because the inversion of even complex matrices is a standard routine in mathematical libraries, (30.18) is a *direct solution* for the  $R$  matrix. A more efficient

approach is to find an  $[R]$  that solves  $[F][R] = [V]$  without computing the inverse. This is accomplished by Gaussian *elimination*.

### 30.1.5

#### Solving $i\epsilon$ Integral Equations $\odot$

The integral equation most commonly encountered in quantum mechanics corresponds to outgoing wave boundary conditions. This means that the singularity is handled by giving the energy  $k_0^2/2\mu$  a small positive imaginary part  $i\epsilon$ . This procedure leads to the Lippmann–Schwinger equation for the  $T$  matrix:

$$T(k', k) = V(k', k) + \frac{2}{\pi} \int_0^\infty dp \frac{p^2 V(k', p) T(p, k)}{(k_0^2 - p^2 + i\epsilon)/2\mu} \quad (30.19)$$

Solving this equation is essentially the same as solving (30.1) for the  $R(k', k)$  matrix. We use the identity (30.5) and decompose the  $i\epsilon$  integral into a principal-value part and an on-shell term:

$$T(k', k) = V(k', k) + \frac{2}{\pi} \mathcal{P} \int_0^\infty dp \frac{p^2 V(k', p) T(p, k)}{(k_0^2 - p^2)/2\mu} - 2i\mu k_0 V(k', k_0) T(k_0, k).$$

Now the last term is incorporated into the numerical analysis by adding an imaginary term to the  $D$  matrix (30.14):

$$D_{N+1} = -\frac{2}{\pi} \sum_{j=1}^N \frac{w_i k_0^2}{(k_0^2 - k_j^2)/2\mu} - 2\mu i k_0 \quad (30.20)$$

The solution proceeds as before, only now with complex matrices arising from the new definition of  $D$ . The resulting on-shell  $T$  matrix element is related to the same experimental phase shift as before, only now through the complex expression

$$T(k_0, k_0) = -\frac{e^{i\delta_l} \sin \delta_l}{\rho} \quad \rho = 2\mu k_0 \quad (30.21)$$

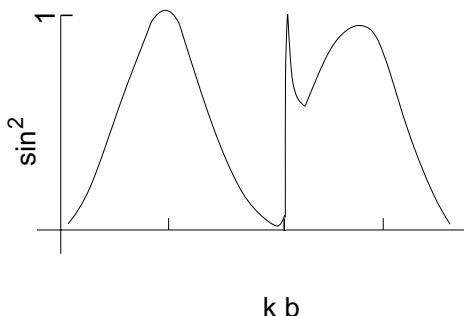
### 30.1.6

#### Delta-Shell Potential Implementation

In Section 29.1.2 we discussed the *delta-shell potential* and gave its momentum-space matrix element for  $l = 0$  partial waves:

$$V(r) = \frac{\lambda}{2\mu} \delta(r - b) \quad (30.22)$$

$$V(k', k) = \frac{\lambda}{2\mu k' k} \sin(k' b) \sin(k b) \quad (30.23)$$



**Fig. 30.4** The energy dependence of that part of the scattering cross section arising from the  $l = 0$  phase shift.

This is one of the few potentials for which the Lippmann–Schwinger integral equation (30.1) has an analytic solution [8]:

$$e^{i\delta} \sin \delta = \frac{-\lambda k_0 b^2 \sin(k_0 b)}{1 + i\lambda k_0 b^2 \sin(k_0 b) [\cos(k_0 b) + i \sin(k_0 b)]} \quad (30.24)$$

With these equations we can calculate the  $l = 0$  phase shift and compare it to that obtained from your numerical solution of the integral Schrödinger equation. In Fig. 30.4 we give a plot of  $\sin^2 \delta_0$  versus  $kb$ . This is proportional to the scattering cross section arising from the  $l = 0$  phase shift. It is seen to reach its maximum value at energies corresponding to resonances. Your numerical results should be similar to this, although it may be difficult to reproduce the very sharp energy dependence.

1. Set up the matrices  $V(i,j)$ ,  $V(i)$ ,  $D(j)$ , and  $F(i,j)$  according to (30.16)–(30.23). Use Gaussian quadrature points with at least  $N = 16$  for your grid.
2. Employ a matrix inversion routine you have obtained from a library to calculate  $F^{-1}$ .
3. Calculate the vector  $R$  by matrix multiplication  $R = F^{-1}V$ .
4. Deduce the  $S$ -wave phase shift  $\delta$  from your  $R$  vector:

$$R(k_0, k_0) = R_{N1,N1} = -\frac{\tan \delta_l}{\rho} \quad \rho = 2\mu k_0 \quad (30.25)$$

5. Estimate the precision of your solution by increasing the number of grid point in steps of 4. If your phase shift changes in the second or third decimal place, you probably have that much precision.

6. Plot  $\sin^2 \delta_0$  versus energy  $E = k_0^2/2\mu$  starting at zero energy and ending at energies where the phase shift is again small. Your results should be similar to those in Fig. 30.4 (calculated from the analytic result). Note that a *resonance* occurs when  $\delta_l$  increases rapidly through  $\pi/2$ ; that is, when  $\sin^2 \delta_0 = 1$ .
7. Check your answer against the analytic results (30.24).

### 30.1.7

#### Scattering Wave Function (Exploration)

1. The  $F^{-1}$  matrix that occurred in our solution to the integral equation,

$$R = F^{-1}V = (1 - VG)^{-1}V \quad (30.26)$$

is actually quite useful. In scattering theory it is known as the *wave matrix* because it is used in the expansion of the wave function:

$$u_l(r) = N_0 \sum_{i=1}^N j_l(k_i r) F(k_i, k_0)^{-1} \quad (30.27)$$

Here  $N_0$  is a normalization constant and standing-wave boundary conditions are built into  $u_l$  if the  $R$  matrix is used to calculate  $F$ . Plot this wave function and compare it to a free wave.

2. Solve for the part of the scattering cross section arising from the  $l = 1$  phase shift for  $0 \leq kb \leq 2\pi$ .

**Listing 30.1:** `Scatt.java` solves the Lippmann–Schwinger integral equation for scattering from a delta-shell potential. The singular integral equations are regularized by a subtraction, converted to matrix equations using Gaussian grid points, and then solved with JAMA.

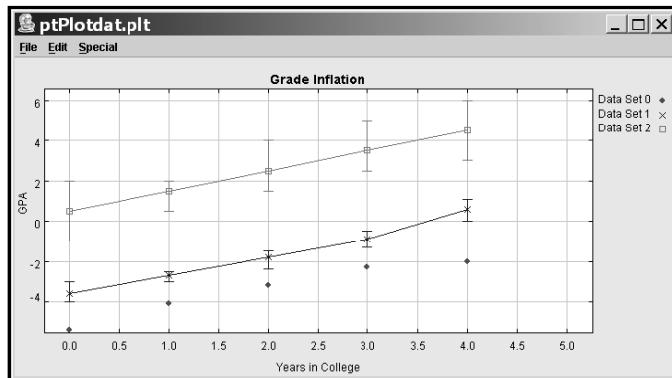
```
// Scatt.java: Soln of Lippmann–Schwinger in p space for scattering

import Jama.*;
import java.io.*;
import java.util.*;

public class Scatt {
    public static void main(String[] argv) throws IOException,
                                                FileNotFoundException {
        PrintWriter q = new PrintWriter(
            new FileOutputStream("sin2.dat"), true);
        int n, i, j, m, Row, Column, M = 300;
        double pot, lambda, scale, ko, Temp, shift, shiftan, sin2, k2;
        double pi = 3.1415926535897932384626, b = 10., RN1, potlast=0.0;
        double[][] F = new double[M][M]; double[] k = new double[M];
        double[] w = new double[M]; double[] D = new double[M];
        double[] r = new double[M]; double[] V = new double[M];
        double[][] P = new double[M][M]; double[][] L = new double[M][M];
        double[][] U = new double[M][M];
        n = 26; scale = n/2; pot = 0.; // Set up Gauss points
        shiftan = 0.; lambda = 1.5; // Set up D matrix
        Gauss.gauss(n, 2, 0., scale, k, w);
        ko = 0.02;
        for (m=1;m<901;m++) {
            k[n] = ko;
            for (i=0; i <= n-1; i++) {
                D[i] = 2/pi*w[i]*k[i]*k[i]/(k[i]*k[i]-ko*ko);
            }
            D[n] = 0.; // Set up F matrix and V vector
            for (j=0; j <= n-1; j++) D[n] = D[n] + w[j]*ko*ko/(k[j]*k[j]-ko*ko);
            D[n] = D[n]*(-2./pi);
            for (i=0; i <= n; i++) {
                for (j=0; j <= n; j++) {
                    pot = -b*b * lambda * Math.sin(b*k[i])
                        * Math.sin(b*k[j])/(k[i]*b*k[j]*b);
                    F[i][j] = pot*D[j];
                    if (i==j) F[i][j] = F[i][j] + 1.;
                }
                V[i] = pot;
            }
            // Change arrays into matrices
            Matrix Fmat = new Matrix(F, n+1, n+1);
            Matrix Vvec = new Matrix(n+1, 1);
            Matrix Finv = Fmat.inverse();
            for (i=0; i <= n; i++) Vvec.set(i, 0, V[i]);
            Matrix R = Finv.times(Vvec); // Invert matrix
            RN1 = R.get(n, 0); // Get last value of R
            // Define phase shift
            shift = Math.atan(-RN1*ko);
            sin2 = Math.sin(shift)*Math.sin(shift);
            q.println(ko*b + " " + sin2);
            ko=ko+0.2*3.141592/1000.0;
        }
        System.out.println("Output in sin2.dat");
    }
}
```

**A****PtPlot: 2D Graphs within Java**

*PtPlot* is an excellent plotting package that lets you plot directly from your Java programs. PtPlot is free, written in Java (and thus runs under Unix, Linux, Mac OS, and MS Windows), is easy to use, and is actually part of Ptolemy, an entire computing environment that is supported by the University of California. Figure A.1 is an example of a PtPlot graph. Because PtPlot is not built into Java, your Java program needs to import the PtPlot package and work with its classes. We suggest that you download the most recent version over the Web.



**Fig. A.1** Sample output from PtPlot in which three data sets are placed on one plot. Observe the error bars on two of the sets.

The program `EasyPtPlot.java` in Listing A.1 is an example of how to construct a simple graph of  $\cos(x)$  versus  $x$  with PtPlot. On line 2 we see the statement `import ptolemy.plot.*;` that imports the PtPlot classes. (In order for this to work for you, you may have to modify your CLASSPATH environmental variable.) PtPlot represents your plot as a *Plot object*, which we name `myPlot` and create on line 7. We then add various features, step by step, to `myPlot` to make it just the plot we want. As is standard with objects in Java, we first give the name of the object and then modify it with “dot modifiers.” Rather than tell PtPlot what ranges for  $x$  and  $y$  to plot, we let PtPlot set the  $x$  and  $y$  ranges based on the data it is given. By having `true` as the fourth ar-

gument in `myPlot.addPoint(0, x, y, true)`, we are telling PtPlot to connect the plotted points. For the plot to appear on the screen, line 16 creates a `PlotApplication` with the plot as input.

**Listing A.1:** `EasyPtPlot.java` plots a function using the package PtPlot. Note that the `Application` object must be created to see the plot on your screen.

```
// EasyPtPlot.java: Simple PtPlot application

import ptolemy.plot.*;

public class EasyPtPlot {
    public static final double Xmin = -5., Xmax = 5.; // Graph domain
    public static final int Npoint = 500;

    public static void main(String[] args) {

        Plot plotObj = new Plot(); // Create Plot object
        plotObj.setTitle("f(x) vs x");
        plotObj.setXLabel("x");
        plotObj.setYLabel("f(x)");
        // plotObj.setSize(400, 300);
        // plotObj.setXRange(Xmin, Xmax);
        // plotObj.addPoint(int Set, double x, double y, boolean connect)
        double xStep = (Xmax - Xmin) / Npoint; // Plotting loop
        for ( double x = Xmin; x <= Xmax; x += xStep) {
            double y = Math.sin(x)*Math.sin(x);
            plotObj.addPoint(0, x, y, true);
        }
        PlotApplication app = new PlotApplication(plotObj); // Display
    }
}
```

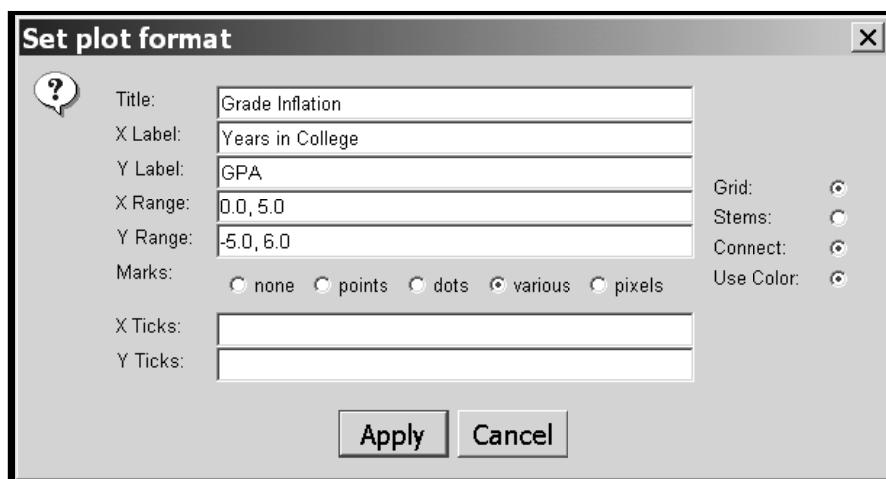
We encourage you to make your plot more informative by including further options in the commands, or by using the pull-down menus in the PtPlot window displaying your plot. The options are found in the description of the methods on the PtPlot Website [90], and include:

### Calling PtPlot from Your Program

<code>Plot myPlot = new Plot();</code>	Name and create plot object <code>myPlot</code>
<code>PlotApplication app = new PlotApplication(myPlot);</code>	Display
<code>myPlot.setTitle("f(x) vs x");</code>	Add title to plot
<code>myPlot.setXLabel("x");</code>	Label <i>x</i> axis
<code>myPlot.setYLabel("f(x)");</code>	Label <i>y</i> axis
<code>myPlot.addPoint(0, x, y, true);</code>	Add $(x, y)$ to set 0, connect points
<code>myPlot.addPoint(1, x, y, false);</code>	Add $(x, y)$ to set 1, no connect points
<code>myPlot.addLegend(0, "Set 0");</code>	Label data set 0 in legend
<code>myPlot.addPointWithErrorBars(0, x, y, yLo, yHi, true);</code>	Plot, $(x, y - YLo), (x, y + yHi)$ + error bars
<code>myPlot.clear(0);</code>	Remove all points from data set 0
<code>myPlot.clear(false);</code>	Remove data from all sets
<code>myPlot.clear(true);</code>	Remove all points, default options
<code>myPlot.setSize(500, 400);</code>	Set plot size in pixels (optional)
<code>myPlot.setXRange(-10., 10.);</code>	Set an <i>x</i> range (default fit to data)
<code>myPlot.setYRange(-8., 8.);</code>	Set a <i>y</i> range (default fit to data)
<code>myPlot.setXLog(true);</code>	Use log scale for <i>x</i> axis
<code>myPlot.setYLog(true);</code>	Use log scale for <i>y</i> axis
<code>myPlot.setGrid(false);</code>	Turn off the grid
<code>myPlot.setColor(false);</code>	Color in black and white
<code>myPlot.setButtons(true);</code>	Display zoom-to-fit button on plot
<code>myPlot.fillPlot();</code>	Adjust <i>x</i> , <i>y</i> ranges to fit data
<code>myPlot.setImpulses(true, 0);</code>	Lines from points to <i>x</i> axis, set 0
<code>myPlot.setMarksStyle("none", 0);</code>	none, points, <i>etc.</i>
<code>myPlot.setBars(true);</code>	Display data as bar charts
<code>String s = myPlot.getTitle();</code>	Extract title (or other properties)

Once you have a PtPlot application on your screen, explore some of the ways to modify your plot from the application window:

1. Examine the Edit pull-down menu (underlined letters are shortcuts). Select Edit and pull it down.
2. From the Edit pull-down menu select Format. You should get a window (Fig. A.2) that lets you control various options in your graph.
3. Experiment with the Format menu; change the graph so only points are plotted and so that your name is in the title.
4. Select a central portion of your plot and **zoom in** on it by drawing a box (with mouse button depressed) starting from the upper left corner and then moving down before you release the mouse button. You **zoom out**



**Fig. A.2** The Format submenu located under the Edit menu in a PtPlot application. This submenu controls the plot's basic features.

by drawing a box from the lower right corner and moving up. You may also resize your graph by selecting Special/Reset Axes or by resetting the  $x$  and  $y$  ranges. And of course, you always have the option of starting over by closing the Java window and running the `java` command again.

5. Scrutinize the File menu and its options for printing your graphs, as well as exporting them to files in postscript (.ps) and other formats.

It is also possible to have PtPlot make graphs by reading data from a file in which the  $x$  and  $y$  values are separated by spaces, tabs, or commas. There is even the option of including PtPlot formatting commands in the file with data. The program `TwoPlotExample.java` on the CD and its data file `data.plt` show how to place two plots side by side, and how to read in a data file containing error bars and various symbols for the points. In the simplest form, a *PtPlot Data Format* is just a text file with a single  $x, y$  point per line. To illustrate, Fig. A.1 was produced from the data file `PtPlotdat.plt`:

#### Sample PtPlot Data file `PtPlotdat.plt`

```
# This is a comment: Sample data for PtPlot TitleText: Grade
Inflation XRange: 0,5 YRange: -5, 6 Grid: on XLabel: Years in
College YLabel: GPA Marks: various NumSets:3 Color: on DataSet: Data
Set 0 Lines:off 0,-5.4 1,-4.1 2,-3.2 3,-2.3 4,-2 DataSet: Data Set
1 Lines:on 0,-3.6, -4,-3 1,-2.7, -3, -2.5 2,-1.8, -2.4,-1.5 3,-0.9,
-1.3, -0.5 4, 0.6, 0,1.1 DataSet: Data Set 2 0,0.5, -1,2 1, 1.5,
0.5, 2 2, 2.5, 1.5, 4 3, 3.5, 2.5, 5 4, 4.5, 3, 6
```

To plot up your data files directly from the command line, enter

```
> java ptolemy.plot.PlotApplication dataFile // Plot data in dataFile
```

This causes the standard PtPlot window to open and display your data. If this does not work, then your `CLASSPATH` variable may not be defined properly, or PtPlot may not be installed.

Reading in your data from the PtPlot window itself is an alternative. Either use an already open window, or issue Java's run command:

```
> java ptolemy.plot.PlotApplication // Open PtPlot window
```

To look at your data from the PtPlot window, choose `File → Open → FileName`. By default, PtPlot will look for files with suffixes `.plt` or `.xml`. However, you may enter any name you want, or pull down the `Filter` menu and select `*` to see all of your files. The same holds for the `File → SaveAs` option. In addition, you may Export your plot as an Encapsulated PostScript (`.eps`) file, a format useful for inserting in printed documents. (You may also use drawing programs to edit the output from PtPlot or to convert it into other formats.)

As with any good plot, you should label your axes, add a title, and add what is needed to be informative and clear. To do this, incorporate PtPlot commands with your data, or work in the PtPlot window with the pull-down menus under `Edit` and `Special`. The options are essentially the same as the ones you would call from your program:

<b>TitleText: f(x) vs. x</b>	Add title to plot
<b>XLabel: x</b>	Label <i>x</i> axis
<b>YLabel: y</b>	Label <i>y</i> axis
<b>XRange: 0, 12</b>	Set <i>x</i> range (default: fit to data)
<b>YRange: -3, 62</b>	Set <i>y</i> range (default: fit to data)
<b>Marks: none</b>	(Default) No marks at points, lines connects points
<b>Marks: points</b>	or: dots, various, pixels
<b>Lines: on/off</b>	Do not connect points with lines; default: <code>on</code>
<b>Impulses: on/off</b>	Lines down from points to <i>x</i> axis; default: <code>off</code>
<b>Bars: on/off</b>	Bar graph (turn off lines) default: <code>off</code>
<b>Bars: width (, offset)</b>	Bar graph; bars of <code>width</code> and (optional) <code>offset</code>
<b>DataSet: string</b>	Specify data set to plot; <i>string</i> appears in legend
<b>x, y</b>	Specify a data point; comma, space, tab separators
<b>move: x, y</b>	Do not connect this point to previous
<b>x, y, yLo, yHi</b>	Plot $(x, y - y_{Lo}), (x, y + y_{Hi})$ with error bars

If commands appear before `DataSet` directives, then the command will apply to all data sets. If commands appear after `DataSet` directives, then it will apply to that data set only.

## B

### Glossary

<b>absolute value</b>	Value of a quantity expressed as a positive number, e.g., $ f(x) $ .
<b>accuracy</b>	The degree of exactness provided by a description or theory. <i>Accuracy</i> usually refers to an absolute quality, while <i>precision</i> usually refers to the number of digits used to represent a number.
<b>address</b>	The numerical designation of a location in memory. An identifier, such as a label, that points to an address in memory or a data source.
<b>algorithm</b>	A set of rules for solving a problem in a finite number of steps. Usually independent of the software or hardware.
<b>allocate</b>	To assign a resource for use, often memory.
<b>alphanumeric</b>	The combination of alphabetic letters, numerical digits, and special characters, such as %, \$, and /.
<b>analog</b>	The mapping of a continuous physical observable to numbers. As an instance, a car's speed to its speedometer.
<b>animation</b>	A process in which motion is simulated by presenting a series of slightly different pictures (frames) in succession.
<b>append</b>	To add on, especially to the end of an object or word.
<b>application</b>	A self-contained executable program containing tasks to be performed by a computer, usually for a practical purpose.
<b>architecture</b>	The overall design of a computer in terms of its major components: memory, processor, I/O, and communication.
<b>archive</b>	To copy programs and data to an auxiliary medium or file system for long-term and compact storage.

<b>argument</b>	A parameter passed from one program part to another, or to a command.
<b>arithmetic unit</b>	Part of the central processing unit that performs arithmetic.
<b>array</b>	( <b>matrix</b> ) A group of numbers stored together in rows and columns that may be referenced by one or more subscripts. Each number in an array is an array element.
<b>assignment statement</b>	Command that sets a value to a variable or symbol.
<b>B</b>	Abbreviation for byte (8 bits).
<b>b</b>	Abbreviation for bit.
<b>background</b>	(1) A technique of having a programming run at low priority (“in background”) while a higher priority program runs “in foreground.” (2) The part of video display not containing windows.
<b>base</b>	The radix of a number system. (10 is the radix of the decimal system.)
<b>basic machine language</b>	Instructions telling the hardware to do basic operations such as store or add binary numbers.
<b>batch</b>	The running of programs without user interaction; often in background.
<b>baud</b>	Number of signal elements per unit time, often 1 bit per second.
<b>binary</b>	Related to the number system with base 2.
<b>BIOS</b>	Basic Input/Output System.
<b>bit</b>	Contraction of “binary digit;” digits 0 or 1 in binary representation.
<b>Boolean algebra</b>	A branch of symbolic logic dealing with logical relations as opposed to numerical values.
<b>boot</b>	To “bootstrap;” to start a computer by loading the operating system.
<b>branch</b>	To pick a path within a program based on the value of variables.

<b>bug</b>	A mistake in a computer program or operating system; a malfunction.
<b>bus</b>	A communication channel (bunch of wires) used for transmitting information quickly among computer parts.
<b>byte</b>	Eight bits of storage. Java uses two bytes to store a single character in extended unicode.
<b>byte code</b>	Compiled code read by all computer systems, but still needing to be interpreted (or recompiled). Contained in class file.
<b>cache</b>	Small, very fast memory used as temporary storage between very fast CPU registers and main memory, or disk and RAM.
<b>calling sequence</b>	Data and setup needed to call a method or subprogram.
<b>CPU</b>	<b>(Central Processing Unit)</b> Part of a computer that accepts and acts on instructions; where calculations are done and communications controlled.
<b>checkpoint</b>	A statement within a program that stops normal execution and provides output to assist in debugging.
<b>checksum</b>	Summation of digits or bits used to check integrity of data.
<b>child</b>	Object created by parent object.
<b>class</b>	(1) Group of objects or methods having a common characteristic. (2) Collection of data types and associated methods. (3) An instance of an object. (4) Byte code version of a Java program.
<b>clock</b>	Electronics that generate periodic signals to control execution.
<b>code</b>	A program or the writing of a program (often compiled).
<b>column</b>	The vertical line of numbers in an array.
<b>column-major order</b>	Method used by Fortran to store matrices in which the leftmost subscript attains its maximum value before subscript to the right is incremented. (Java and C use row-major order.)
<b>command</b>	A computer instruction. A control signal.

<b>command key</b>	A keyboard key, or combination of keys, that performs a pre-defined function.
<b>compilation</b>	Translation of a program written in a high-level language into (more) basic language.
<b>compiler</b>	A program that translates source code from a high-level computer language to more basic machine language.
<b>concatenate</b>	To join together two or more strings head to tail.
<b>concurrent processing</b>	Same as parallel processing; simultaneous execution of several related instructions.
<b>conditional statement</b>	Statement executed only under certain conditions.
<b>control character</b>	A character that modifies or controls the running of a program (e.g., <i>control + C</i> ).
<b>control statement</b>	A statement within a program that transfers control to another section of the program.
<b>copy</b>	To transfer data <i>without</i> removing the original.
<b>CPU</b>	See central processing unit.
<b>crash</b>	The abnormal termination of a program or a piece of hardware.
<b>cycle time</b>	( <b>clock speed</b> ) Time for CPU to execute a simple instruction.
<b>data</b>	Information stored in numerical form; plural of datum.
<b>data dependence</b>	Two statements addressing identical storage locations.
<b>dependence</b>	Relation among program statements in which the results depend on the order in which the statements are executed.
<b>data type</b>	Definitions that permits proper interpretation of character string.
<b>debug</b>	To detect, locate, and remove mistakes in software or hardware.
<b>default</b>	The assumption made when no specific directive is given.
<b>delete</b>	To remove and leave no record.
<b>DFT</b>	Discrete Fourier transform.

<b>digital</b>	Representation of quantities in discrete form; contrast analog.
<b>dimension of array</b>	Number of elements that may be referenced by an array index. <i>Logical dimension</i> is the largest value actually used by the program.
<b>directory</b>	A collection of files given their own name.
<b>disc, disk</b>	A circular magnetic medium used for storage.
<b>discrete</b>	Related to distinct elements.
<b>double precision</b>	Use of two memory words to store a number.
<b>download</b>	To transfer data <i>from</i> a remote computer <i>to</i> a local computer.
<b>DRAM</b>	Dynamic RAM, needing periodic refreshment. See SRAM
<b>driver</b>	A set of instructions needed to transmit data to/from external device.
<b>dump</b>	Data resulting from listing all information in memory.
<b>dynamic RAM</b>	Computer memory needing frequent refreshment.
<b>E</b>	A symbol for exponent. To illustrate, $1.97\text{E}2 = 1.97 \times 10^2$ .
<b>element</b>	An item of data within an array; a component of a language.
<b>enable</b>	To make a computer part operative.
<b>ethernet</b>	A high-speed local area network (LAN) composed of specific cable technology and communication protocols.
<b>executable program</b>	A set of instructions that can be loaded into the computer's memory and executed.
<b>executable statement</b>	A statement that causes some computational action, such as assigning a value to a variable.
<b>fetch</b>	To locate and retrieve information from storage.
<b>FFT</b>	Fast Fourier transform.
<b>flash memory</b>	Memory that does not require power to retain contents.
<b>floating point</b>	Finite storage of numbers in scientific notation.
<b>FLOP</b>	Floating Point Operation.

<b>foreground</b>	Running high-priority programs before lower priority.
<b>Fortran</b>	Acronym for <b>formula translation</b> .
<b>fragmentation</b>	File storage in many small, dispersed pieces.
<b>G</b>	Abbreviation for giga.
<b>garbage</b>	Meaningless numbers, usually the result of error or improper definition. Obsolete data in memory waiting to be removed (“collected”).
<b>giga</b>	Prefix indicating one billion, $10^9$ , of something (USA).
<b>GUI</b>	Graphical user interface; a window environment.
<b>hard disk</b>	A circular, spinning, storage device using magnetic memory.
<b>hardware</b>	Physical components of a computer system.
<b>hashing</b>	A transformation that converts keystrokes to data values.
<b>heuristic</b>	A trial-and-error approach to problem solving.
<b>hexadecimal</b>	Base 16; {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}.
<b>hidden line surface</b>	Part of a graphics object normally hidden from view.
<b>high-level language</b>	Programming language similar to normal language.
<b>host computer</b>	A central or remote computer serving other computers.
<b>HPC</b>	High-performance computing.
<b>icon</b>	Small on-screen symbol that activates an application.
<b>increment</b>	The amount added to a variable, especially an array index.
<b>index</b>	The symbol used to locate a variable in an array, the subscript.
<b>infinite loop</b>	The endless repeating of a set of instructions.
<b>input</b>	Introduction of data from an external device into main storage.
<b>instructions</b>	Commands to the hardware to do basic things.
<b>instruction stack</b>	Ordered group of instructions currently in use.
<b>interpolation</b>	Finding values between known values.

<b>interpreter</b>	A language translator that sequentially converts each line of source code into machine code and immediately executes each line.
<b>interrupt</b>	A command that stops execution of a program when an abnormal condition is encountered.
<b>iterate</b>	To repeat a series of steps automatically.
<b>jump</b>	A departure from the linear processing of code; branch, transfer.
<b>just-in-time</b>	(Compiler) A program that recompiles Java class file into more efficient machine code.
<b>K</b>	Abbreviation for KILO, one thousand, $10^3$ .
<b>kernel</b>	The inner or central part of a large program or of an operating system that does not get modified (much) when run on different computers.
<b>kill</b>	To delete or stop a process.
<b>LAN</b>	Local area network.
<b>LAPACK</b>	Linear algebra package (subroutine library).
<b>language</b>	Rules, representations, and conventions used to communicate information.
<b>LHS</b>	Left-hand side.
<b>library (lib)</b>	A collection of programs or methods usually on a related topic.
<b>linking</b>	Connecting separate pieces of code to form an executable program.
<b>literal</b>	A symbol that defines itself, such as the letter <i>A</i> .
<b>load</b>	To read information into the computer's memory.
<b>load module</b>	A program that is loaded into memory and run immediately.
<b>log in (on)</b>	To sign onto the computer, to begin a session.
<b>loop</b>	Set of instructions executed repeatedly as long as some condition is met.
<b>low-level language</b>	Machine-related commands not for humans.

<b>machine language</b>	Commands understood by computer hardware.
<b>machine precision</b>	The maximum positive number that, when added to the number stored as 1, does not change it.
<b>macro</b>	A single, higher level statement resulting in several lower level ones.
<b>main method</b>	Section of application program where execution begins.
<b>main storage</b>	The fast, electronic memory; physical memory.
<b>mantissa</b>	Significant digits in a floating-point number; e.g., 1.2 in 1.2E3.
<b>mega, M</b>	A prefix denoting a million, or $1,048,576 = 2^{20}$ .
<b>method</b>	A subroutine used to calculate a function or manipulate data.
<b>MIMD</b>	Multiple instruction, multiple data computer.
<b>modular programming</b>	The technique of writing program with many, reusable, and independent parts.
<b>mod</b>	<b>(modulo)</b> Function that yields remainder after division of numbers.
<b>multiprocessors</b>	Computers with more than one processor.
<b>multitasking</b>	The system by which several jobs reside in a computer's memory simultaneously; may run in parallel or sequentially.
<b>NAN</b>	Not a number, a computer error message.
<b>nesting</b>	Embedding a group of statements within another group.
<b>object</b>	A software component with multiple parts or properties.
<b>OOP</b>	<b>(object-oriented programming)</b> A modular programming style focused on classes of data objects and associated methods to interact with the objects.
<b>object program</b>	<b>(code)</b> A program in basic machine language produced by compiling a high-level language.
<b>octal</b>	Base 8; easy to convert to or from binary.
<b>ODE</b>	Ordinary differential equation.

<b>1D</b>	One-dimensional.
<b>operating system (OS)</b>	The program that controls the computer and runs applications, processes I/O, and shells.
<b>OOP</b>	Object-oriented programming.
<b>optimization</b>	The modification of a program to make it run more quickly.
<b>overflow</b>	Result of trying to store too large a number.
<b>package</b>	A collection of related programs or classes.
<b>page</b>	A segment of memory that gets read as a single block.
<b>parallel processing</b>	Simultaneous or independent processing in different CPUs.
<b>parallelization</b>	Rewriting an existing program to run in parallel.
<b>partition</b>	The section of memory assigned to a program during its execution.
<b>PC</b>	Personal computer.
<b>PDE</b>	Partial differential equation.
<b>physical memory</b>	The fast, electronic memory of a computer; main memory; contrast to <i>virtual memory</i> .
<b>physical record</b>	The physical unit of data for input or output that may contain a number of logical records.
<b>pipeline units</b>	Assembly-line approach to central processing; CPU simultaneously gathers, stores, and processes data.
<b>pixel</b>	A picture element, a dot on the screen. See also voxel.
<b>.pdf</b>	(Portable Document Format) A document format developed by Adobe that is of high quality and readable by extended browser.
<b>.ps</b>	(PostScript) A language developed by Adobe for printing high quality text and graphics.
<b>precision</b>	The degree of exactness with which a quantity is presented. High-precision numbers are not necessarily <i>accurate</i> .
<b>program</b>	A set of instructions that a computer interprets and executes.

<b>protocol</b>	A set of rules or conventions.
<b>pseudocode</b>	A mixture of normal language and coding that provides a symbolic guide to a program.
<b>queue</b>	An ordered group of items waiting to be acted upon in turn.
<b>radix</b>	The base number in a number system that gets raised to powers.
<b>RAM</b>	Random access (central) memory that is reached directly.
<b>random access</b>	Reading or writing memory independent of storage order.
<b>record</b>	A collection of data items treated as a unit.
<b>recursion</b>	Repetition producing new values from previous ones.
<b>registers</b>	Very high-speed memory used by the central processing unit.
<b>reserved words</b>	Words that cannot be used in an application program.
<b>RHS</b>	Right-hand side.
<b>RISC</b>	A CPU design with for a Reduced Instruction Set Computer.
<b>row-major order</b>	The method used by Java to store matrices in which the rightmost subscript varies most rapidly and attains its maximum value before the left subscript is incremented.
<b>run</b>	To execute a program.
<b>scalar</b>	A data value or number, to illustrate, $\pi$ .
<b>serial processing (scalar)</b>	Calculations in which numbers are processed in sequence. Contrast to vector and parallel processing.
<b>shell</b>	A command line interpreter; the part of the operating system in which the user enters commands.
<b>SIMD</b>	Single instruction, multiple data computer.
<b>simulation</b>	The modeling of a real system by a computer program.
<b>single precision</b>	The use of one computer word to store a variable.
<b>SISD</b>	Single instruction, single data computer.
<b>software</b>	Programs or instructions.

<b>source code</b>	Program in high-level language needing compilation to run.
<b>SRAM</b>	Static RAM. Memory that retains its contents as long as power is applied. Contrast DRAM.
<b>stochastic</b>	A process in which there is an element of chance.
<b>stride</b>	Number of array elements stepped through as an operation repeats.
<b>string</b>	A connected sequence of characters treated as a single object.
<b>structure</b>	The organization or arrangement of a program or a computer.
<b>subprogram</b>	Part of program invoked by another program unit; <i>subroutine</i> .
<b>supercomputer</b>	The class of fastest and most powerful computers available.
<b>superscalar</b>	A latter generation RISC computer.
<b>syntax</b>	The rules governing the structure of a language.
<b>TCP/IP</b>	Transmission control protocol/internet protocol.
<b>telnet</b>	Protocols for computer-computer communications.
<b>tera, T</b>	$10^{12}$ , or $2^{30} = 1,073,741,824$ .
<b>top-down</b>	Designing a program from the most general view of the problem, down to the specific subroutines.
<b>unary</b>	An operation that uses only one operand; monadic.
<b>underflow</b>	Result of trying to store too small a number.
<b>unit</b>	A device having a special function.
<b>upload</b>	Data transfer <i>from</i> local <i>to</i> remote computer; opposite of download.
<b>URL</b>	Universal resource locator, web address.
<b>utility programs</b>	Programs to enhance other programs or do chores.
<b>vector</b>	A group of $N$ numbers in memory arranged in 1D order.
<b>vector processing</b>	Calculations in which an entire vector of numbers is processed with one operation.

**virtual memory** Memory on the slow, hard disk and not in fast RAM.

**visualization** Conversion of numbers into 2D and 3D pictures or graphs.

**volume** A physical unit of a storage medium, such as a disk.

**word** A unit of main storage, usually 1, 2, 4, 6, or 8 bytes.

**word length** The amount of memory used to store a computer word.

**WWW** World wide web.

## C

### Fortran 95 Codes

(Alphabetic order modified somewhat to avoid awkward continuations.)

**Listing C.1: decay.f95**

```

!     decay.f90: Spontaneous radioactive decay simulation
!
Program decay

Implicit none
Real*8 :: r, ranDom, lambda
Integer :: i, j, h, nleft, nloop, start, seed

! Set params (decay rate, initial no of atoms, seed), plant seed
lambda = 0.01
start = 1000
seed = 11168
h = 1
nloop = start
nleft = start
open(6, File = 'decay.dat')           ! open output 'file'
                                         ! loop over times and over atoms
Do j = 1, 10000
  Do i = 1, nleft
    r = ranDom(seed)
    If (r <= lambda) then
      nloop = nloop -1
    Endif
  End Do
                                         ! atom loop Ends
  nleft = nloop
  Write (6, *) h, ' ', real(nleft)/start
  h = h + 1
  If (nleft == 0) goto 30
End Do
30 close(6)
Stop 'data saved in decay.dat'
End Program decay

```

**Listing C.2: bessel.f95**

```

! bessel.f95: Computation spherical Bessel functions by recurrence

Program bessel

Implicit none
Real*8 :: step, x, xmin, xmax, up, Down, t1, t2
Integer :: order, start

xmin = 0.25
xmax = 40.0
step = 0.1
order = 10
start = 50
open(6, File = 'bessel.dat', Status = 'Unknown') ! open output file
Do x = xmin, xmax, step
    t1 = Down(x, order, start)
    t2 = up(x, order)
    write (6, 50) x, t1, t2
End Do
Close(6)
50 Format (f15.10, f15.10, f15.10)
Stop 'data saved in bessel.dat'
End Program bessel

Function Down(x, order, start) ! calculate using Downward recursion
Implicit none
Integer :: k, order, start
Real*8 :: Down, scale, x, j(100)
                           ! the arbitrary start
j(start + 1) = 1
j(start) = 1
Do k = start, 2, -1
    j(k - 1) = ((2*k - 1.0)/x)*j(k) - j(k + 1)
End Do
                           ! scale so that j(1) = sin(x)/x
scale = (sin(x)/x)/j(1)
Down = j(order + 1)*scale
Return
End

Function up(x, order) ! calculate using upward recursion
Implicit none
Integer :: k, order
Real*8 :: up, x, one, two, thr
one = sin(x)/x
two = (sin(x) - x*cos(x))/(x*x)
Do k = 1, (order - 1)
    thr = ((2*k + 1.0)/x)*two - one
    one = two
    two = thr
End Do
up = thr
Return
End

```

**Listing C.3: diff.f95**

```

! diff.f90: Forward, central and extrapolated differentiation

Program diff

Implicit none
Real*8 :: f, h, result(3), x, xmin, xmax, xstep

open(6, File = 'diff.dat', Status = 'Unknown')
h      = 1.e-5
xmin  = 0.0
xmax  = 7.0
xstep = 0.01
Do x = xmin, xmax, xstep
    result(1) = (f(x+h) - f(x))/h
    result(2) = (f(x+h/2) - f(x-h/2))/h
    result(3) = (8*(f(x+h/4)-f(x-h/4)) - (f(x+h/2)-f(x-h/2)))/(3*h)
    write (6, 20) x, result(1), result(2), result(3)
End Do
20 Format(F5.3, TR4, F10.8, TR4, F10.8, TR4, F10.8)
close(6)
Stop 'data saved in diff.dat'
End Program diff
                                         ! function to integrate

Function f(x)
Implicit none
Real*8 f, x
f = cos(x)
Return
End

```

**Listing C.4: eqheat.f95, label**

```

!   eqheat.f90: Solution of heat equation using with finite diffs

Program heat

Implicit none
Double precision :: cons, ro, sph, thk, u(101, 2)
Integer :: i, k, max

open(9, FILE = 'eqheat.dat', Status = 'Unknown')
sph = 0.113                                ! specific heat iron
thk = 0.12                                 ! thermal conductivity iron
ro = 7.8                                    ! density for iron
cons = thk/(sph*ro)
max = 30000                                  ! number of iterations
                                                ! t = 0, all points at 100 C

Do i = 1, 100
  u(i, 1) = 100.0
End Do
Do i = 1, 2                                  ! Endpoints always zero
  u(1, i) = 0.0
  u(101, i) = 0.0
End Do
                                                ! loop over time
Do k = 1, max
  ! loop over space
  Do i = 2, 100
    u(i, 2) = u(i, 1) + cons*(u(i+ 1, 1) + u(i- 1, 1) - 2*u(i, 1))
  End Do
  If ( (Mod(k, 1000) == 0) .or. (k == 1) ) then ! every 1000 steps
    Do i = 1, 101, 2
      write(9, 22) u(i, 2)
    End Do
    Write (9, 22)
  EndIf
  ! new values -> old
  Do i = 2, 100
    u(i, 1) = u(i, 2)
  End Do
End Do
22 format (f10.6)
close(9)
Stop 'data saved in eqheat.dat (for gnuplot)'
End Program heat

```

**Listing C.5: eqstring.f95**

```

!      eqstring.f90: Solution of wave equation using time stepping

Program eqstring

Implicit none
Real*8 :: x(101, 3)
Integer :: i, j, k, max

max = 100
open(9, FILE = 'eqstring.dat', Status = 'Unknown')
Do i = 1, 101
  Do j = 1, 3
    x(i, j) = 0.0
  End Do
End Do                                ! initialize

Do i = 1, 80
  x(i, 1) = 0.00125*i
End Do
Do i = 81, 101
  x(i, 1) = 0.1 - 0.005*(i- 81)
End Do                                    ! first time step

Do i = 2, 100
  x(i, 2) = x(i, 1) + 0.5*(x(i+1, 1) + x(i-1, 1) - 2.*x(i, 1))
End Do                                     ! other time steps

Do k = 1, max
  Do i = 2, 100
    x(i,3) = 2.*x(i,2) - x(i,1) + (x(i+1,2) + x(i-1,2) - 2.*x(i,2))
  End Do
  Do i = 1, 101
    x(i, 1) = x(i, 2)                      ! new -> old
    x(i, 2) = x(i, 3)
  End Do
  If (modulo(k, 10) == 0) then            ! output data every 10 steps
    Do i = 1, 101
      write(9, 11) x(i, 3)
    End Do
    write(9, *) ' '
  Endif
End Do
11 format (e12.6)
close(9)
Stop 'data saved in eqstring.dat (for gnuplot)'
End Program eqstring

```

**Listing C.6: exp-bad.f95**

```

!   exp-bad.f90: calculating exp(- x) as a finite sum, bad algorithm

Program expbad

Implicit none
      ! min = accuracy, x step, max in x, up numer, down denomin.
Real*8 :: down, min, max, step, sum, up, x
Integer :: i, j

min = 1E - 10
max = 10.
step = 0.1
open(6, File = 'exp-bad.dat', Status = 'Unknown')           ! summation
Do x = 0, max, step
  sum = 1
  i = 0
  down = 1
  up = 1
      ! while loop may never stop
  Do while((sum == 0).or.(abs((up/down)/sum) > min))
    i = i + 1
    down = 1
    up = 1
    Do j = 1, i
      up = - up*x
      down = down*j
    End Do
    sum = sum + up/down
  End Do
  write (6, *) x, sum
End Do
close(6)
Stop 'data saved in exp-bad.dat'
End Program expbad

```

**Listing C.7: exp-good.f95**

```

!   exp-good.f90: calculate e^- x as a finite sum, good algorithm

Program expgood

Implicit none
Real*8 :: element, min, max, step, sum, x
Integer :: n

min = 1E - 10
max = 10.
step = 0.1
open(6, File = 'exp-good.dat', Status = 'Unknown')           ! summation
Do x = 0, max, step
  sum = 1
  element = 1

```

```

n = 0
      ! while loop may never stop
Do while ((abs(element/sum) > min) .or. (sum .eq. 0))
  n = n + 1
  element = element*(-x)/n
  sum     = sum + element
End Do
  write (6, *) x, sum
End Do
close(6)
Stop 'data saved in exp-good.dat'
End Program expgood

```

Listing C.8: fit.f95

```

! fit.f95: Least square fit

Program fit

Implicit none
Integer :: i
Real*8 :: s, sx, sy, sxx, sxy, delta, inter, slope
Real*8 :: x(12), y(12), d(12)

Data y /328, 187, 821, 78, 88, 6, 5, 82, 2, 0.1, 84, 1/ ! y values
Do i = 1, 12                                         ! values x
  x(i) = i*10 - 5
End Do
                                         ! input delta y
Do i = 1, 12
  d(i) = 1.
End Do
s = 0.0;   sx = 0.;    sy = 0.
sxx = 0.
sxy = 0.
                                         ! calculate sums
Do i = 1, 12
  s = s +           1 / (d(i)*d(i))
  sx = sx +         x(i) / (d(i)*d(i))
  sy = sy +         y(i) / (d(i)*d(i))
  sxx = sxx + x(i)*x(i) / (d(i)*d(i))
  sxy = sxy + x(i)*y(i) / (d(i)*d(i))
End Do
                                         ! calculate coefficients
delta = s*sxx - sx*sx
slope = (s*sxy - sx*sy) / delta
inter = (sxx*sy - sx*sxy) / delta
write(*, *) 'intercept = ', inter
write(*, *) 'slope = ', slope
write(*, *) 'correlation = ', -sx/sqrt(sxx*s)
Stop 'fit'
End Program fit

```

**Listing C.9: gauss.f95**

```

! gauss.f90: Points and weights for Gaussian quadrature
! rescales the gauss - legendre grid points and weights
!
! npts      number of points
! job      = 0  rescalling uniformly between (a, b)
!           1  for integral (0, b) with 50% points inside (0, ab/(a + b))
!           2  for integral (a, inf) with 50% inside (a, b + 2a)
! x, w      output grid points and weights.

subroutine gauss(npts, job, a, b, x, w)

    Integer :: npts, job, m, i, j
    Real*8 :: x(npts), w(npts), a, b, xi
    Real*8 :: t, t1, pp, p1, p2, p3, aj
    Real*8 :: eps, pi, zero, two, one, half, quarter
    parameter (pi = 3.14159265358979323846264338328, eps = 3.0E - 14)
    parameter (zero = 0.d0, one = 1.d0, two = 2.d0)
    parameter (half = 0.5d0, quarter = 0.25d0)

    m = (npts + 1)/2
    Do i = 1, m
        t = cos(pi*(i - quarter)/(npts + half))
    10 continue
        p1 = one
        p2 = zero
        aj = zero
        Do j = 1, npts
            p3 = p2
            p2 = p1
            aj = aj + one
            p1 = ((two*aj - one)*t*p2 - (aj - one)*p3)/aj
        End Do
        pp = npts*(t*p1 - p2)/(t*t - one)
        t1 = t
        t = t1 - p1/pp
        If (abs(t - t1) > eps ) goto 10
        x(i) = -t
        x(npts + 1 - i) = t
        w(i) = two/((one - t*t)*pp*pp)
        w(npts + 1 - i) = w(i)
    End Do
                                ! rescale grid points
    select case(job)
                                ! scale to (a, b) uniformly
        case (0)
            Do i = 1, npts
                x(i) = x(i)*(b - a)/two + (b + a)/two
                w(i) = w(i)*(b - a)/two
            End Do
                                ! scale to (0, b) with 50% points inside (0, ab/(a + b))
        case(1)
            Do i = 1, npts
                xi = x(i)
                x(i) = a*b*(one + xi)/(b + a - (b - a)*xi)
                w(i) = w(i)*two*a*b*b/((b + a - (b-a)*xi)*(b + a - (b-a)*xi))
            End Do
    End select

```

```

    End Do
        ! scale to (a, inf) with 50% inside (a, b + 2a)
case(2)
    Do i = 1, npts
        xi = x(i)
        x(i) = (b*xi + b + a + a)/(one - xi)
        w(i) = w(i)*two*(a + b)/((one - xi)*(one - xi))
    End Do
End select
Return
End

```

**Listing C.10: int10d.f95**

```

! int - 10d.f90: Ten dimensional integration using Monte - Carlo

Program int10d

Implicit none
Integer :: m = 16, k
Real*8 :: s, integ(16)                                ! number of trials

s = 0.
Do k = 1, m
    call montecarlo(integ, k);
    s = s + integ(k)
End Do
write(*, *) s/m
End Program int10d

subroutine montecarlo(integ, k)
Implicit none
Integer :: i, j, k, max = 65536
Real*8 :: x, y, sum, ranDom, integ(16)
x = 0.
y = 0.
sum = 0.
Do i = 1, max
    x = 0                                         ! reset x
                                                ! sum 10 x values
    Do j = 1, 10
        x = x + ranDom()
    End Do
                                                ! square and sum up
    y = y + x*x
    sum = sum + y/i;
End Do
integ(k) = sum/max
write(*, *) k, integ(k)
End

```

**Listing C.11: harmos.f95**

```

!   harmos.f90: Solves t dependent Schrödinger eqtn for Gaussian wavepacket
!               in harmonic oscillator potential well

Program harmos

Implicit None
Real*8 :: psr(750, 2), psi(750, 2), v(750), p2(750)
Real*8 :: pi, dx, k0, dt, x
Complex :: exc, zi
Integer :: max, i, j, n

Open(9, FILE = 'harmos.dat', Status = 'Unknown')
pi = 3.1415926535897932385E0
zi = cmplx(0., 1.)
dx = 0.02
k0 = 3 * pi
dt = dx*dx/4.                                ! initial momentum
max = 750
Do i = 1, max
  Do j = 1, 2
    psi(i, j) = 0.
    psr(i, j) = 0.
  End Do
End Do                                         ! initial conditions
x = - 7.5
Do i = 1, max
  exc = exp(zi*k0*x)
  psr(i, 1) = real(exc*exp(- 0.5*(x/0.5)**2)) ! real wave Function
  psi(i, 1) = aimag(exc*exp(- 0.5*(x/0.5)**2)) ! imag wave Function
  v(i) = 5.*x*x                                     ! potential
  x = x + dx
End Do                                         ! propagate solution in time
Do n = 1, 20000
  Do i = 2, max - 1 ! real part psr and the probability p2
    psr(i, 2) = psr(i, 1) - dt*(psi(i+ 1, 1) + psi(i- 1, 1) &
                               - 2.*psi(i, 1))/(dx*dx) + dt*v(i)*psi(i, 1)
    p2(i) = psr(i, 1)*psr(i, 2) + psi(i, 1)*psi(i, 1)
  End Do
  Do i = 2, max - 1 ! imag part
    psi(i, 2) = psi(i, 1) + dt*(psr(i+ 1, 2) + psr(i- 1, 2) &
                               - 2.*psr(i, 2))/(dx*dx) - dt*v(i)*psr(i, 2)
  End Do
  If((n == 1).or.(modulo(n, 2000) == 0)) Then
    Do i = 2, max - 1, 10
      Write(9, 11) p2(i) + 0.0015*v(i)
    End Do
    Write(9, *) ' '
  EndIf
  Do i = 1, max ! new -> old
    psi(i, 1) = psi(i, 2)
    psr(i, 1) = psr(i, 2)
  End Do

```

```

End Do
11 Format(E12.6)
Close(9)
Stop 'data saved in harmos.dat (for gnuplot)'
End

```

**Listing C.12: lagrange.f95**

```

!      lagrange.f: Langrange interpolation of cross table

Program lagrange

Implicit none
Real*8 :: inter, x, xin(9), yin(9)
Integer :: i, e

e = 9
open(6, File = 'lagrange.dat', Status = 'Unknown')           ! Input data
data xin /0, 85, 580, 758, 800, 1285, 850, 795, 82/
data yin /18.6, 16, 85, 83.5, 58.8, 19.9, 10.8, 88.25, 4.7 /
                                                ! Calculate f(x)
Do i = 0, 1000
  x = i*0.2
  write (6, *) x, inter(xin, yin, e, x)
End Do
Close(6)
Stop 'data saved in lagrange.dat'
End Program lagrange                                         ! Evaluate interpolation function(x)

Function inter(xin, yin, e, x)
Implicit none
Integer :: i, j, e
Real*8 :: inter, lambda(9), xin(9), yin(9), x
inter = 0
Do i = 1, e
  lambda(i) = 1
  Do j = 1, e
    If (i.neqv.j) then
      lambda(i) = lambda(i) * ((x - xin(j))/(xin(i) - xin(j)))
    Endif
  End Do
  inter = inter + (yin(i) * lambda(i))
End Do
Return
End

```

**Listing C.13: integ.f95**

```

! integrate.f90: Integrate exp(-x) using trap , Simp and Gauss rules
!                               Need to add in Gauss.f95

Program integrate

Implicit none
Real*8 :: trapez, simpson, quad, r1, r2, r3           ! declarations
Real*8 :: theo, vmin, vmax
Integer :: i

theo = 0.632120558829      ! theoretical result, integration range
vmin = 0.
vmax = 1.
open(6, File = 'integ.dat', Status = 'Unknown')
      ! calculate integral using both methods for steps = 3..501
Do i = 3, 501, 2
    r1 = trapez(i, vmin, vmax)
    r1 = abs(r1 - theo)
    r2 = simpson(i, vmin, vmax)
    r2 = abs(r2 - theo)
    r3 = quad(i, vmin, vmax)
    r3 = abs(r3 - theo)
    write(6, *) i, r1, r2, r3
End Do
close(6)
Stop 'data saved in integ.dat'
End Program integrate                                ! Function we want to integrate

Function f(x)
Implicit none
Real*8 :: f, x
f = exp( - x)
Return
End

Function trapez(i, min, max)! trapezoid rule
Implicit none
Integer :: i, n
Real*8 :: f, interval, min, max, trapez, x
trapez = 0
interval = ((max - min) / (i - 1))
Do n = 2, (i - 1)                                     ! sum midpoints
    x = interval * (n - 1)
    trapez = trapez + f(x)*interval
End Do
trapez = trapez + 0.5*(f(min) + f(max))*interval ! add Endpoints
Return
End                                                 ! Simpson rule

Function simpson(i, min, max)
Implicit none
Integer :: i, n
Real*8 :: f, interval, min, max, simpson, x
simpson = 0
interval = ((max - min) / (i - 1))

```

```

Do n = 2, (i-1), 2                                ! loop for odd points
  x = interval * (n - 1)
  simpson = simpson + 4*f(x)
End Do
Do n = 3, (i-1), 2                                ! loop for even points
  x = interval * (n - 1)
  simpson = simpson + 2*f(x)
End Do
simpson = simpson + f(min) + f(max)            ! add the Endpoints
simpson = simpson*interval/3
Return
End

Function quad(i, min, max)                         ! uses Gauss points
  Implicit none
  Real*8 :: w(1000), x(1000)
  Real*8 :: f, min, max, quad
  Integer :: i, job, n
  quad = 0
  job = 0
  call gauss(i, job, min, max, x, w)
  Do n = 1, i
    quad = quad + f(x(n))*w(n)
  End Do
  Return
End

```

Listing C.14: limit.f95

```

!      limit.f90: determines the machine precision
!
Program limit                                     ! determines the machine precision

Implicit none
Integer :: I, N
Real*8 :: eps, one

N = 60                                         ! number of iterations N
eps = 1.                                         ! set initial values
one = 1.0

Do I = 1, N
  eps = eps / 2
  one = 1 + eps
  write (*, *) I, one, eps
End Do
Stop 'limit'
End Program limit

```

**Listing C.15: LaplaceSOR.f95**

```

! LaplaceSOR.f90: Solve Laplace eq with finite differences c SOR

Program LaplaceSOR

Implicit none
Integer :: max = 40, i, j, iter
Real*8 :: tol, omega, r, p(40, 40)

Open(6, FILE = 'laplaceR.dat', Status = 'Unknown')      ! Data file
omega = 1.8                                              ! SOR parameter
                                                       ! clear the array

Do i = 1, max
  Do j = 1, max
    p(i, j) = 0
  End Do
End Do
                                                       ! p[i][0] = 100 V

Do i = 1, max
  p(i, 1) = + 100.0
End Do
tol = 1.0                                              ! tolerance
iter =1                                                 ! iterations

Do while ( (tol > 0.000001).and. (iter <= 140) )
  tol = 0.0
  Do i = 2, (max - 1)
    Do j = 2, (max - 1)
      r = omega * ( p(i, j + 1) + p(i, j - 1) + p(i+ 1, j) + &
                     p(i- 1, j) - 4. * p(i, j) ) / 4.0
      p(i, j) = p(i, j) + r
      If ( abs(r) > tol ) then
        tol = abs(r)
      Endif
    End Do
    iter = iter + 1
  End Do
End Do
                                                       ! write data gnuplot 3D format

Do i = 1, max
  Do j = 1, max
    write(6, *) p(i, j)
  End Do
  write(6, *) ''
End Do
close(6)
Stop 'data stored in laplaceR.dat (for gnuplot)'
End Program LaplaceSOR

```

**Listing C.16: Newton\_cd.f95**

```

! Newton_cd.f90: Newton-Raphson root finder, central diff derivative
!
Program Newton_cd

Implicit none
Integer :: it, imax = 10      ! Maximum number of iterations permitted
Real*8 :: x, dx = 1e - 2, eps = 1e - 6, f1, df, F

                           ! x guess, must be close to root
x = 2.
Do   it = 0, imax
    f1   = F(x)                      ! Compute Function value
    write(*, *) it, x, f1
    df = ( F(x + dx/2) - F(x - dx/2) )/dx
    dx   = - f1/df
    x =   x + dx                     ! New guess
    If ( abs(F(x)) <= eps ) then    ! Check for convergence
        write(*, *) eps
        Stop
    Endif
End Do
End Program Newton_cd

Function F(x)
Implicit none
Real*8 :: x, F
F = 2*cos(x) - x
End

```

**Listing C.17: Newton\_fd.f95**

```

! Newton_fd.f90:Newton-Raphson root finder, forward diff derivative
!
Program NewtonRHL_fd

Implicit none
Integer :: it, imax = 10          ! Max number iterations
Real*8 :: x, dx = 1e - 2, eps = 1e - 6, df, F

                           ! Guess must be close
x = 2.
Do   it = 1, imax
    df = ( F(x + dx) - F(x) )/dx
    dx   = - F(x)/df
    x =   x + dx                  ! New guess
    write(*, *) it, x, F(x)
    If ( abs(F(x)) <= eps ) then ! Check for convergence
        write(*, *) eps
        Stop
    Endif

```

```

    End Do
End                                         ! Find zero of this function
function F(x)
  Implicit none
  Real*8 :: x, F
  F = 2*cos(x) - x
End

```

**Listing C.18:** overflow.f95

```

!      overflow.f90: determine overflow and underflow limits

Program overflow

Implicit none
Integer :: I, N
Real*8 :: under, over
N = 1024                      ! number of iterations , may need bigger
under = 1.                         ! set initial values
over = 1.
Do I = 1, N          ! calc underflow and overflow, output to screen
  under = under / 2
  over = over * 2
  write (*, *) I, over, under
End Do
Stop 'overflow'
End Program overflow

```

**Listing C.19:** pond.f95

```

!      pond.f90: pi via Monte-Carlo integration (throwing stones)

Program pond

Implicit none
Real*8 :: area, x, y, ranDom
Integer :: i, max, pi

max = 2000                      ! open file , set initial value, seed generator
Open(6, File = 'pond.dat', Status = 'Unknown')
pi = 0                            ! execute
Do i = 1, max
  x = ranDom()*2 - 1
  y = ranDom()*2 - 1
  If ((x*x + y*y) <= 1) then
    pi = pi + 1
  Endif
  area = 4. * pi/Real(i)
  write(6, *) i, area
End Do
close(6)
Stop 'data saved in pond.dat'
End Program pond

```

**Listing C.20: qmc.f95**

```

!     qmc.f90: Feynman path integral for ground state wave Function

Program qmc

Implicit none
Integer :: i, j, max, element, prop(100)
Real*8 :: change, ranDom, energy, newE, oldE, out, path(100)

max = 250000
open(9, FILE = 'qmc.dat', Status = 'Unknown')           ! initial path and probability
Do j = 1, 100
    path(j) = 0.0
    prop(j) = 0
End Do
                                         ! find energy of initial path
oldE = energy(path, 100)                         ! pick random element, change by random
Do i = 1, max
    element = ranDom()*100 + 1
    change = ((ranDom() - 0.5)*2)
    path(element) = path(element) + change
    newE = energy(path, 100)                         ! find new energy
                                                ! Metropolis algorithm

If ((newE > oldE) .AND. (exp(-newE + oldE) < ranDom())) then
    path(element) = path(element) - change
EndIf
                                         ! add up probabilities
Do j = 1, 100
    element = path(j)*10 + 50
    prop(element) = prop(element) + 1
End Do
oldE = newE
End Do
                                         ! write output data to file
Do j = 1, 100
    out = prop(j)
    write(9, *) j - 50, out/max
End Do
close(9)
Stop 'data saved in qmc.dat'
End Program qmc
                                         ! Function calculates energy of the system
Function energy(array, max)
Implicit none
Integer :: i, max
Real*8 :: energy, array(max)
energy = 0
Do i = 1, (max - 1)
    energy = energy + (array(i+ 1) - array(i))**2 + array(i)**2
End Do
Return
End

```

**Listing C.21: rk4.f95**

```

!      rk4.f90: 4th order rk solution for harmonic oscillator

Program oscillator

Implicit none
! n: number of equations, min/max in x, dist: length of x - steps
! y(1): initial position, y(2):initial velocity
Real*8 :: dist, min1, max1, x, y(5)
Integer :: n

n = 2
min1 = 0.0;  max1 = 10.0
dist = 0.1
y(1) = 1.0;  y(2) = 0.
open(6, File = 'rk4.dat', Status = 'Unknown')
                                         ! Do n steps rk algorithm
Do x = min1, max1, dist
    call rk4(x, dist, y, n)
    write (6, *) x, y(1)
End Do
close(6)
Stop 'data saved in rk4.dat'
End Program oscillator                                ! End of main Program

subroutine rk4(x, xstep, y, n)                      ! rk4 subroutine
Implicit none
Real*8 :: deriv, h, x, xstep, y(5)
Real*8, dimension(5) :: k1, k2, k3, k4, t1, t2, t3
Integer :: i, n
h = xstep/2.0
Do i = 1, n
    k1(i) = xstep * deriv(x, y, i)
    t1(i) = y(i) + 0.5*k1(i)
End Do
Do i = 1, n
    k2(i) = xstep * deriv(x + h, t1, i)
    t2(i) = y(i) + 0.5*k2(i)
End Do
Do i = 1, n
    k3(i) = xstep * deriv(x + h, t2, i)
    t3(i) = y(i) + k3(i)
End Do
Do i = 1, n
    k4(i) = xstep * deriv(x + xstep, t3, i)
    y(i) = y(i) + (k1(i) + (2.*(k2(i) + k3(i))) + k4(i))/6.0
End Do
Return
End                                         ! Function Returns derivatives

Function deriv(x, temp, i)
Implicit none
Real*8 :: deriv, x, temp(2)
Integer :: i
If (i == 1) deriv = temp(2)
If (i == 2) deriv = - temp(1)
Return
End

```

**Listing C.22: rk45.f95**

```

!   rk45.f90: ODE solver via variable step size rk, Tol = error

Program Rk45

Implicit none
Real *8 :: h, t, s, hmin, hmax, Tol = 2*1E - 7, Tmin = 0., &
           Tmax = 10.
Real *8, dimension(2) :: y, FReturn, ydumb, k1, k2, k3, k4, &
                         k5, k6, err
Integer :: i, Ntimes = 10

Open(6, FILE = 'rk45.dat', Status = 'Unknown')          ! initialize
y(1) = 3.0 ; y(2) = - 5.0
h = (Tmax - Tmin) / Ntimes                           ! tentative number of steps
hmin = h/64                                         ! minimum and maximum step size
hmax = h*64
t = Tmin
                                         ! output to file

Do while (t < Tmax)
    write(*, *) t, y(1), y(2)
    write(6, *) t, y(1)
    If ( (t + h) > Tmax ) then
        h = Tmax - t ! the last step
    EndIf
                                         ! evaluate both RHSs and Return in F
    call f(t, y, FReturn)
    Do i = 1, 2
        k1(i) = h*FReturn(i)
        ydumb(i) = y(i) + k1(i)/4
    End Do
    call f(t + h/4, ydumb, FReturn)
    Do i = 1, 2
        k2(i) = h*FReturn(i)
        ydumb(i) = y(i) + 3*k1(i)/32 + 9*k2(i)/32
    End Do
    call f(t + 3*h/8, ydumb, FReturn)
    Do i = 1, 2
        k3(i) = h*FReturn(i)
        ydumb(i) = y(i) + 1932*k1(i)/2197 - 7200*k2(i)/2197. &
                    + 7296*k3(i)/2197
    End Do
    call f(t + 12*h/13, ydumb, FReturn)
    Do i = 1, 2
        k4(i) = h*FReturn(i)
        ydumb(i) = y(i) + 439*k1(i)/216 - 8*k2(i) &
                    + 3680*k3(i)/513 - 845*k4(i)/4104
    End Do
    call f(t + h, ydumb, FReturn)
    Do i = 1, 2
        k5(i) = h*FReturn(i)
        ydumb(i) = y(i) - 8*k1(i)/27 + 2*k2(i) - 3544*k3(i)/2565 &
                    + 1859*k4(i)/4104 - 11*k5(i)/40
    End Do
    call f(t + h/2, ydumb, FReturn)

```

```

Do i = 1, 2
    k6(i) = h*FReturn(i)
    err(i) = abs( k1(i)/360 - 128*k3(i)/4275 - 2197*k4(i)/75240 &
                  + k5(i)/50. + 2*k6(i)/55 )
End Do
If ((err(1) < Tol).or.(err(2) < Tol).or.(h <= 2*hmin)) then
    ! accept approximation
    Do i = 1, 2
        y(i) = y(i) + 25*k1(i)/216. + 1408*k3(i)/2565. &
                  + 2197*k4(i)/4104. - k5(i)/5.
    End Do
    t = t + h
Endif
If ((err(1) == 0).or.(err(2) == 0)) then
    s = 0! trap division by 0
else
    s = 0.84*Tol*h/err(1)**0.25
endif
If ( (s < 0.75).and. (h > 2*hmin) )then
    h = h/2.! reduce step
else If ( (s > 1.5).and. (2* h < hmax) )then
    h = h*2. ! increase step
endif
                                         ! End loop
End Do
close(6)
Stop'Data stored in rk45.dat'
End Program Rk45
                                         ! PLACE YOUR FUNCTION HERE
subroutine f(t,      y,  FReturn)
    Implicit none; Real *8 t, y(2), FReturn(2)
    FReturn(1) = y(2)! RHS of first equation
    FReturn(2) = - 100*y(1) - 2*y(2) + 10*sin(3*t)! RHS of 2nd equation
    Return
End

```

Listing C.23: random.f95

```

!     ranDom.f90: simple random number generator, not for serious work

Program random

Implicit none
Integer :: i, number, old, seed, x, y

! set parameters (seed for generator, number of generated numbers)
seed = 11
number = 1000
! open output file, seed number generator
open(6, FILE = 'ranDom.dat', Status = 'Unknown')
old = seed
                                         ! execution
Do i = 1, number
    x = modulo((57*old + 1), 256)

```

```

y = modulo((57*x + 1), 256)
write (6, *) x, y
old = y
End Do
close(6)
Stop 'data saved in ranDom.dat'
End Program random

```

**Listing C.24: scatt.f95**

```

!      scatt.f90: scattering phase shift in p space from delta shell
!      potential, LU decomposition with partial pivoting.
!      uses gauss.f,    LUfactor, LUSolve (included)

Program scatt

Integer :: n, Size, i, j, Row, Column
Double Precision :: b, Pot
Parameter (Size = 300, pi = 3.1415926535897932384626, b = 10.0)
Double Precision :: lambda, scale, ko, Temp
Double Precision :: F(Size, Size), k(Size), w(Size), D(Size), r(Size)
Double Precision :: V(Size), L(Size, Size), U(Size, Size), P(Size, Size)
Integer :: PivotInfo(Size)

          ! Enter potential strength lambda
Write(*, *) 'enter lambda'
Read(*, *) lambda
Write(*, *) 'enter scaling factor'
Read(*, *) scale
Write(*, *) 'enter ko'
Read(*, *) ko
Write(*, *) 'enter grid size'
Read(*, *) n
          ! Set up Gaussian integration points and weights
          ! on interval [0, inf] with the mid - point at 'scale'
          ! Set last element in k array to ko
call gauss(n, 2, 0d0, scale, k, w)
          ! Set up D matrix
Do i = 1, n
  D(i) = 2.0d0/pi*w(i)*k(i)*k(i)/(k(i)*k(i) - ko*ko)
End Do
D(n + 1) = 0.0
Do j = 1, n
  D(n + 1) = D(n + 1) + w(j)*ko*ko/(k(j)*k(j) - ko*ko)
End Do
D(n + 1) = D(n + 1)*(- 2.0d0/pi)
          ! Set up F matrix and V vector
Do i = 1, n
  Do j = 1, n
    Pot = - b*b*lambda*SIN(b*k(i))*SIN(b*k(j))
    Pot = Pot/(k(i)*b*k(j)*b)
    F(i, j) = Pot*D(j)
    If (i == j) then
      F(i, j) = F(i, j) + 1.0d0
    End If
  End Do
End Do

```

```

        Endif
    End Do
    V(i) = Pot
End Do
!      LU factorization. Put LU factors of F in corresponding matrix
!              (not efficient but easy ). Store partial pivoting info
!
call LUfactor(F, n, Size, L, U, PivotInfo)
!      Pivot and solve
!      Set P to identity matrix
Do Row = 1, n + 1
Do Column = 1, n + 1
    P(Row, Column) = 0
    If (Row .EQ. Column) P(Row, Column) = 1
End Do
End Do
! Interchange rows to get true P matrix
Do Row = 1, n
Do Column = 1, n
    Temp = P(Row, Column)
    P(Row, Column) = P(PivotInfo(Row), Column)
    P(PivotInfo(Row), Column) = Temp
End Do
End Do
call LUSolve(V, L, U, n, Size, PivotInfo, r)
! output results
write(*, *) ko*ko, DATAN( - r(n)*ko)
End Program scatt

!      LU factorization , partial pivoting of A in Ax = b
subroutine LUfactor(A, n, Size, L, U, PivotInfo)
Integer :: n, Column, CurrentPivotRow, CurrentRow, SwapCol, Row
Integer :: ElimCol, Size
Double Precision :: A(Size, Size), L(Size, Size), U(Size, Size)
Integer :: PivotInfo(Size)
Double Precision :: CurrentPivotValue, Swap
Do Column = 1, n - 1
    CurrentPivotRow = Column
    CurrentPivotValue = A(CurrentPivotRow, Column)
    ! Determine row for largest pivot
    Do CurrentRow = Column + 1, n
        If ( DABS(A(CurrentRow, Column)) .GT. CurrentPivotValue ) Then
            CurrentPivotValue = DABS(A(CurrentRow, Column))
            CurrentPivotRow = CurrentRow
        Endif
    End Do
    PivotInfo(Column) = CurrentPivotRow
    ! Swap rows so largest value at pivot
    Do SwapCol = Column, n
        Swap = A(Column, SwapCol)
        A(Column, SwapCol) = A(PivotInfo(Column), SwapCol)
        A(PivotInfo(Column), SwapCol) = Swap
    End Do
!
! Gauss Elimin , upper triangular A, unpivoted lower triangular L
!
Do Row = Column + 1, n

```

```

L(Row, Column) = A(Row, Column)/A(Column, Column)
Do ElimCol = Column + 1, n
    A(Row, ElimCol) = A(Row, ElimCol) &
        - L(Row, Column)*A(Column, ElimCol)
End Do
End Do
End Do
                                ! Ensure bottom right not pivoted to 0
PivotInfo(n) = n
Do Row = 2, n - 1
                                ! Now pivot the L
    DO Column = 1, Row - 1
        Swap = L(Row, Column)
        L(Row, Column) = L(PivotInfo(Row), Column)
        L(PivotInfo(Row), Column) = Swap
    End Do
End Do
                                ! Clean up L and U
Do Column = 1, n
    Do Row = 1, Column
        U(Row, Column) = A(Row, Column)
        L(Row, Column) = 0
        IF (Row .EQ. Column) L(Row, Column) = 1
    End Do
    Do Row = Column + 1, n
        U(Row, Column) = 0
    End Do
End Do
Return
End

! Part of an LU decomposition + partial pivoting to solve Ax = b
Subroutine LUSolve(b, L, U, n, Size, PivotInfo, x)
    Integer :: n, Size, Row, Column
    Double Precision :: b(Size), x(Size)
    Integer :: PivotInfo(Size)
    Double Precision :: L(Size, Size), U(Size, Size)
    Double Precision :: Temp
    Do Row = 1, n
        ! Interchange rows of b for pivoting
        Temp = b(Row)
        b(Row) = b(PivotInfo(Row))
        b(PivotInfo(Row)) = Temp
    End Do
    ! Solve Ly = b, where y = Ux, by forward elimination
    Do Row = 2, n
        DO Column = 1, Row - 1
            b(Row) = b(Row) - L(Row, Column)*b(Column)
        End Do
        b(Row) = b(Row)/L(Row, Row)
    End Do
    ! Solve Ux = y by back substitution
    x(n) = b(n)/U(n, n)
    Do Row = n - 1, 1, - 1
        x(Row) = b(Row)
        Do Column = Row + 1, n
            x(Row) = x(Row) - U(Row, Column)*x(Column)
        End Do
    End Do

```

```

      x(Row) = x(Row)/U(Row, Row)
End Do
Return
End

```

**Listing C.25: slit.f95**

```

!      slit.f90: Solves time - dependent Schroedinger equation for a
!      two - dimensional Gaussian wavepacket entering a slit

Program slit

Implicit none
Real*8 :: psr(91, 91, 2), psi(91, 91, 2), v(91, 91), p2(91, 91)
Real*8 :: a1, a2, dt, dx, k0x, k0y, x0, y0, x, y
Integer i, j, k, max, n, time
Complex exc, zi

      ! input positive int proportional to time for plot
write(*, *)'Enter a positive Integer from 1(initial time)'
write(*, *)'to 800 to get wave packet position at that time'
read(*, *)time
write(*, *)'processing data for time', time
open(9, FILE = 'slit.dat', Status = 'Unknown')
      ! initialize constants and wave packet
zi = cmplx(0.0, 1.0)
dx = 0.2
dt = 0.0025/(dx*dx)
      ! initial momentum, position
k0x = 0.0;   k0y = 2.5
x0 = 0.0;   y0 = - 7.0
max = 90
      ! clear the arrays
Do i = 1, 91
  Do j = 1, 91
    Do k = 1, 2
      psi(i, j, k) = 0.0
      psr(i, j, k) = 0.0
    End Do
  End Do
End Do
      ! initial wave function
y = - 9.0
Do j = 1, max + 1
  x = - 9.0d0
  Do i = 1, max + 1
    exc = exp(zi*(k0x*x + k0y*y))
    a1 = exp(- 0.5*((x - x0)**2 + (y - y0)**2))
    psr(i, j, 1) = real(a1*exc)
    psi(i, j, 1) = aimag(a1*exc)
    x = x + dx
  End Do
  y = y + dx
End Do
      ! set potential slit width: 50 - 40 = 10 units
Do j = 1, max + 1

```

```

Do i = 1, max + 1
  If ((j == 35).and.((i < 40).or.(i > 51)))then
    v(i, j) = 0.5
  else
    v(i, j) = 0.0
  Endif
End Do
End Do
Do n = 1, time
  ! propagate psi through time
  Do j = 2, max
    ! compute real wave packet and probability
    Do i = 2, max
      a2 = v(i, j)*psi(i, j, 1) + 2.0*d0*dt*psi(i, j, 1)
      a1 = psi(i+1,j,1) + psi(i-1,j,1) +psi(i,j+1,1) + psi(i,j-1,1)
      psr(i, j, 2) = psr(i, j, 1) - dt*a1 + 2.0*a2
      If (n == time) then
        p2(i, j) = psr(i, j, 1)*psr(i, j, 1) + psi(i, j, 1)*psi(i, j, 1)
      Endif
    End Do
    psr(1, j, 2) = psr(2, j, 2)! at x edges derivative =0
    psr(max + 1, j, 2) = psr(max, j, 2)
  End Do
  ! imaginary part of psi
  Do j = 2, max
    Do i = 2, max
      a2 = v(i, j)*psr(i, j, 2) + 2.0*dt*psr(i, j, 2)
      a1 = psr(i+1,j,2) + psr(i-1,j,2) +psr(i,j-1,2) + psr(i,j+1,2)
      psi(i, j, 2) = psi(i, j, 1) + dt*a1 - 2.0*a2
    End Do
    psi(1, j, 2) = psi(2, j, 2)! at x edges derivative =0
    psi(max + 1, j, 2) = psi(max, j, 2)
  End Do
  ! new -> old
  Do j = 1, max + 1
    Do i = 1, max + 1
      psi(i, j, 1) = psi(i, j, 2)
      psr(i, j, 1) = psr(i, j, 2)
    End Do
  End Do
  ! write probabilities & potential scaled by 0.025 (to fit)
  Do j = 2, max, 3
    Do i = 2, max, 2
      write(9, 11)p2(i, j) + v(i, j)
    End Do
    write(9, *) ''
  End Do
  11 format (E12.6)
  close(9)
  Stop 'data saved in slit.dat'
End

```

**Listing C.26: soliton.f95**

```

! soliton.f90: Solves the KdV Equation via finite differences
!
Program soliton

Implicit None
Real*8 :: ds, dt, max, mu, eps, u(131, 3)
parameter(ds = 0.4, dt = 0.1, max = 2000, mu = 0.1, eps = 0.2)
! delta t, delta x, time steps, mu and eps from KdV equation
Real*8 :: a1, a2, a3, fac, time
Integer :: i, j, k

open (9, FILE = 'soliton.dat', Status = 'Unknown')
! Initial condition
Do i = 1, 131
  u(i, 1) = 0.5*(1. - tanh(0.2*ds*(i- 1) - 5.))
End Do
! Endpoints
u(1, 2) = 1.
u(1, 3) = 1.
u(131, 2) = 0.
u(131, 3) = 0.
fac = mu*dt/(ds**3.)
time = dt
! first step
Do i = 2, 130
  a1 = eps*dt*(u(i+ 1, 1) + u(i, 1) + u(i- 1, 1))/(ds*6.d0)
  If ((i > 2).and.(i <= 129)) then
    a2 = u(i+ 2, 1) + 2.*u(i- 1, 1) - 2.*u(i+ 1, 1) - u(i- 2, 1)
  Endif
  If ((i == 2).or.(i == 130)) then
    a2 = u(i- 1, 1) - u(i+ 1, 1)
  Endif
  a3 = u(i+ 1, 1) - u(i- 1, 1)
  u(i, 2) = u(i, 1) - a1*a3 - fac*a2/3.d0
End Do
! all other times
Do j = 1, max
  Do i = 2, 130
    a1 = eps*dt*(u(i+ 1, 2) + u(i, 2) + u(i- 1, 2))/(3.d0*ds)
    If ((i > 2).and.(i <= 129)) then
      a2 = u(i+2,2) + 2.d0*u(i-1,2) - 2.d0*u(i+1,2) - u(i-2,2)
    Endif
    If ((i == 2).or.(i == 130)) then
      a2 = u(i- 1, 2) - u(i+ 1, 2)
    Endif
    a3 = u(i+ 1, 2) - u(i- 1, 2)
    u(i, 3) = u(i, 1) - a1*a3 - 2.d0*fac*a2/3.d0
    u(1, 3) = 1.d0
  End Do
  ! new -> old
  Do k = 1, 131
    u(k, 1) = u(k, 2)
    u(k, 2) = u(k, 3)
  End Do
  ! output every 200 time steps
End

```

```

If (modulo(j, 200) == 0) then
  Do k = 1, 131
    write(9, 22)u(k, 3)
  End Do
  write(9, 22)
EndIf
time = time + dt
End Do
22 format(f10.6)
close(9)
Stop 'data saved in soliton.dat (for gnuplot)'
End Program soliton

```

Listing C.27: Spline.f95

```

! spline.f90: Cubic Spline fit, based on " Numerical Recipes in C "
Program spline
Implicit none

! input array x[n], y[n] represents tabulation Function y(x)
! with x0 < x1 ... < x(n - 1). n = # of tabulated points
! output yout for given xout (here xout via loop at End)
! yp1 and ypn: 1st derivatives at Endpoints, evaluated internally
! y2[n] is array of second derivatives
! (setting yp1 or ypn > 0.99e30 produces natural spline)

Real*8 :: xout, yout, h, b, a, Nfit, p, qn, sig, un, yp1, ypn, x(9)
REAL*8 :: y(9), y2(9), u(9)
Integer :: klo, khi, k, n, i
                           ! Save data, input data
open(9, FILE = 'Spline.dat', Status = 'Unknown')
open(10, FILE = 'Input.dat', Status = 'Unknown')
                           ! enter your own data here!
data x / 0., 1.2, 2.5, 3.7, 5., 6.2, 7.5, 8.7, 9.9/
data y / 0., 0.93, 0.6, - 0.53, - 0.96, - 0.08, 0.94, 0.66, - 0.46 /
n = 9

Do i = 1, n
  write(10, *) x(i), y(i)
End Do
Nfit = 3000;
                           ! enter the desired number of points to fit
yp1 = (y(2) - y(1))/(x(2) - x(1)) - (y(3) - y(2))/(x(3) - x(2)) &
      + (y(3) - y(1))/(x(3) - x(1))                                ! 1st deriv
ypn = (y(n-1) - y(n-2))/(x(n-1) - x(n-2)) - (y(n-2) &
      - y(n-3))/(x(n-2)-x(n-3)) + (y(n-1)-y(n-3))/(x(n-1)-x(n-3))
If (yp1 > 0.99e30) then
  y2(1) = 0.0
  u(1) = 0.0
else
  y2(1) = ( - 0.5)
  u(1) = (3.0/(x(2) - x(1)))*((y(2) - y(1))/(x(2) - x(1)) - yp1)
Endif
                           ! decomposition loop; y2, u are temps

```

```

Do i = 2, n - 1
    sig = (x(i) - x(i- 1))/(x(i+ 1) - x(i- 1));
    p = sig*y2(i- 1) + 2.0
    y2(i) = (sig - 1.0)/p
    u(i) = (y(i+1)-y(i))/(x(i+1)-x(i)) - (y(i)-y(i-1))/(x(i)-x(i-1))
    u(i) = (6.0*u(i)/(x(i+1) - x(i-1)) - sig*u(i-1))/p;
End Do
                                ! test for natural
                                ! else evaluate second derivative
If (ypn > 0.99e30) then
    qn = 0.0
    un = 0.
    else
        qn = 0.5
        un = (3/(x(n-1) - x(n-2)))*(ypn - (y(n-1)-y(n-2)) &
                                         /(x(n-1) - x(n-2)))
        y2(n - 1) = (un - qn*u(n - 2))/(qn*y2(n - 2) + 1.0)
Endif
                                ! back substitution
Do k = n - 2, 1, - 1
    y2(k) = y2(k)*y2(k + 1) + u(k)
End Do
                                ! splint (initialization) Ends

                                ! Parameters determined, Begin *spline* fit
                                ! loop over xout values
Do i = 1, Nfit
    xout = x(1) + (x(n) - x(1))*(i)/(Nfit)
    klo = 0
    khi = n - 1
                                ! Bisection algor for place in table
                                ! klo, khi bracket xout
Do while (khi - klo > 1)
    k = (khi + klo)/2.0
    If (x(k) > xout) then
        khi = k
    else
        klo = k
    Endif
End Do
h = x(khi) - x(klo)
If (x(k) > xout) then
    khi = k
else
    klo = k
Endif
h = x(khi) - x(klo)
a = (x(khi) - xout)/h
b = (xout - x(klo))/h
yout = (a*y(klo)+b*y(khi) &
        + ((a*a*a-a)*y2(klo)+(b*b*b-b)*y2(khi))*h*h/6)
                                ! write data in gnuplot 2D format
    write (9, *) xout, yout
End Do
Stop 'data stored in Spline.dat'
End Program spline

```

**Listing C.28: sqwell.f95**

```

! sqwell.f90: Solves the t-dependent Schroedinger equation for a
! Gaussian wavepacket in a infinite square well potential

Program sqwell

Implicit None
Real*8 :: psr(751, 2), psi(751, 2), p2(751)
Real*8 :: dx, k0, dt, x, pi
Integer :: i, j, n, max
Complex exc, zi

Common /values/dx, dt

open(9, FILE = 'sqwell.dat', Status = 'Unknown')
max      = 750
pi       = 3.14159265358979323846
zi       = CMPLX(0.0, 1.0)
dx       = 0.02
k0       = 17.0*pi
dt       = dx*dx

! clear the arrays

Do i = 1, 751
  Do j = 1, 2
    psr(i, j) = 0.0
    psi(i, j) = 0.0
    p2(i) = 0.0
  End Do
End Do
! initial conditions

x      = 0.0
Do i = 1, max + 1
  exc      = exp(zi*k0*x)
  psr(i, 1) = real(exc*exp(-0.5*(2.0*(x - 5.0))**2)) ! real part
  psi(i, 1) = aimag(exc*exp(-0.5*(2.0*(x - 5.0))**2)) ! imag
  x        = x + dx
End Do
! propagate solution through time

Do n = 1, 6000
  Do i = 2, max! real part & prob
    psr(i, 2) = psr(i, 1) - dt*(psi(i+ 1, 1) + psi(i- 1, 1)-
                                - 2.0*psi(i, 1))/(2.0*dx*dx)
    p2(i)     = psr(i, 1)*psr(i, 2) + psi(i, 1)*psi(i, 1)
  End Do
  Do i = 2, max
    ! imaginary part
    psi(i, 2) = psi(i, 1) + dt*(psr(i+ 1, 2) + psr(i- 1, 2)-
                                - 2.0*psr(i, 2))/(2.0*dx*dx)
  End Do
! selective printout

If (Mod(n, 300) == 0) then
  Do i = 1, max + 1, 15
    write(9, 11) p2(i)
  End Do
  write(9, *) ' '
Endif
! new soltn -> old

```

```

      Do i = 1, max + 1
        psi(i, 1) = psi(i, 2)
        psr(i, 1) = psr(i, 2)
      End Do
    End Do
  11 format (E12.6)
  close(9)
  Stop 'data saved in sqwell.dat'
End

```

**Listing C.29: tune.f95**

```

!      tune.f90: matrix algebra program to be tuned for performance

Program tune

parameter (ldim = 2050)
Implicit Double precision (a - h, o - z)
dimension ham(ldim, ldim), coef(ldim), sigma(ldim)
                           ! set up H and starting vector
Do i = 1, ldim
  Do j = 1, ldim
    If ( abs(j - i) > 10) then
      ham(j, i) = 0.
    else
      ham(j, i) = 0.3**Abs(j - i)
    EndIf
  End Do
  ham(i, i) = i
  coef(i) = 0.
End Do
coef(1) = 1.                                ! start iterating
err = 1.
iter = 0
20  If (iter < 15 .and. err > 1.e-6) then
    iter = iter + 1                          ! compute current energy & normalize
    ener = 0.
    ovlp = 0.
    Do i = 1, ldim
      ovlp = ovlp + coef(i)*coef(i)
      sigma(i) = 0.
      Do j = 1, ldim
        sigma(i) = sigma(i) + coef(j)*ham(j, i)
      End Do
      ener = ener + coef(i)*sigma(i)
    End Do
    ener = ener/ovlp
    Do I = 1, ldim
      coef(i) = coef(i)/Sqrt(ovlp)
      sigma(i) = sigma(i)/Sqrt(ovlp)
    End Do
                           ! compute update and error norm
    err = 0.
    Do i = 1, ldim

```

```

If ( i == 1) goto 23
step = (sigma(i) - ener*coef(i))/(ener - ham(i, i))
coef(i) = coef(i) + step
err = err + step**2
23 End Do
err = sqrt(err)
write(*, '(1x, i2, 7f10.5)' ) iter, ener, err, coef(1)
goto 20
Endif
Stop
End Program tune

```

Listing C.30: twodsol.f95

```

! twodsol.f90: Solves the sine - Gordon equation for a 2D soliton

Program twodsol

Implicit none
Double precision :: u(201, 201, 3)
Integer :: nint

Open(9, FILE = 'twodsol.dat', Status = 'UNKNOWN')
write(*, *)' Enter an Integer from 1 to 100'
write(*, *)' this number is proportional to time'
write(*, *)' time = 0 is for the Integer = 1'
read(*, *)nint
write(*, *)'working with input = ', nint
call initial(u)                                ! initialize
                                                ! output for t proportional to nint
call solution(u, nint)
Stop
End Program twodsol                           ! initialize constants and soliton

Subroutine initial(u)
Implicit none
Integer :: i, j, k
Double precision :: u(201, 201, 3), dx, dy, dt, xx, yy, dts, time

Common /values/ dx, dy, dt, time, dts
Do i = 1, 201! clear arrays
  Do j = 1, 201
    Do k = 1, 3
      u(i, j, k) = 0.0
    End Do
  End Do
End Do
dx = 14.0/200. ! initial condition
dy = dx
dt = dx/sqrt(2.0)
dts = (dt/dx)**2
yy = - 7.0
time = 0.0
Do i = 1, 201
  xx = - 7.0
  Do j = 1, 201

```

```

        u(i, j, 1) = 4.0*Datan(3. - sqrt(xx*xx + yy*yy))
        xx = xx + dx
    End Do
    yy = yy + dy
End Do
Return
End
! solve SGE, initial conditions in initial
Subroutine solution(u, nint)
Implicit none
Double precision :: u(201, 201, 3), dx, dy, dt, time, a2, zz, dts, a1
Integer :: l, m, mm, k, j, i, nint
Common/values/ dx, dy, dt, time, dts
time = time + dt
    ! 2nd iteration uses d phi/dt(t=0) = 0 (G(x, y, 0) = 0)
    ! d U/dx = 0 at -x0, x0, -y0 and y0
Do l = 2, 200
    Do m = 2, 200
        a2 = u(m+1, 1, 1) + u(m-1, 1, 1) + u(m, 1+1, 1) + u(m, 1-1, 1)
        u(m, 1, 2) = 0.5*(dts*a2 - dt*dt*DSIN(0.25*a2))
    End Do
End Do
! the borders in 2nd iteration
Do mm = 2, 200
    u(mm, 1, 2) = u(mm, 2, 2)
    u(mm, 201, 2) = u(mm, 200, 2)
    u(1, mm, 2) = u(2, mm, 2)
    u(201, mm, 2) = u(200, mm, 2)
End Do
! the still undefined terms
u(1, 1, 2) = u(2, 1, 2)
u(201, 1, 2) = u(200, 1, 2)
u(1, 201, 2) = u(2, 201, 2)
u(201, 201, 2) = u(200, 201, 2)
! 3rd and following iterations use your input, loop up to nint
Do k = 1, nint
    Do l = 2, 200
        Do m = 2, 200
            a1 = u(m+1, 1, 2) + u(m-1, 1, 2) + u(m, 1+1, 2) + u(m, 1-1, 2)
            u(m, 1, 3) = -u(m, 1, 1) + dts*a1 - dt*dt*DSIN(0.25*a1)
            u(m, 1, 3) = u(m, 2, 3)
            u(m, 201, 3) = u(m, 200, 3)
        End Do
    End Do
    Do mm = 2, 200
        u(mm, 1, 3) = u(mm, 2, 3)
        u(mm, 201, 3) = u(mm, 200, 3)
        u(1, mm, 3) = u(2, mm, 3)
        u(201, mm, 3) = u(200, mm, 3)
    End Do
    u(1, 1, 3) = u(2, 1, 3)
    u(201, 1, 3) = u(200, 1, 3)
    u(1, 201, 3) = u(2, 201, 3)
    u(201, 201, 3) = u(200, 201, 3)
! new -> old
Do l = 1, 201
    Do m = 1, 201

```

```

        u(1, m, 1) = u(1, m, 2)
        u(1, m, 2) = u(1, m, 3)
    End Do
End Do
        ! Output solution at time proportional to nint
If (k == nint) then
    Do i = 1, 201, 5
        Do j = 1, 201, 5
            zz = DSIN(u(i, j, 3)/2.0)
            write(9, *)zz
        End Do
            ! need blank lines to separate spatial rows for 3-D
            write(9, *)' '
        End Do
    Endif
    time = time + dt
End Do
Return
End

```

**Listing C.31: walk.f95**

```

! walk.f90 :RanDom walk simulation

Program walk

Implicit none
Real*8 :: ranDom, root2, x, y, r(1:10000)
Integer :: i, j, max

max      = 10000                      ! set parameters (# of steps)
root2 = 1.4142135623730950488E0
open(6, FILE = 'walk.dat', Status = 'Unknown')          ! open file
                                                        ! clear array
Do j = 1, max
    r(j) = 0
End Do
                                                ! average over 100 trials
Do j = 1, 100
    x = 0.
    y = 0.
    Do i = 1, max
        x = x + (ranDom() - 0.5)*2.0*root2
        y = y + (ranDom() - 0.5)*2.0*root2
        r(i) = r(i) + Sqrt(x*x + y*y)
    End Do
End Do
                                                ! output data for plot of r vs. sqrt(N)
Do i = 1, max
    Write (6, *) Sqrt(Real(i)), ' ', r(i)/100
End Do
close(6)
Stop 'data saved in walk.dat'
End Program walk

```

## D

### Fortran 77 Codes

(Alphabetic order modified somewhat to avoid awkward continuations.)

**Listing D.1: area.f**

```
c  area.f: Area of a circle , r input from terminal
c
c      Program area

c      Double Precision pi, r, A
c                      Best value of pi for IEEE floating point
c      pi = 3.1415926535897932385E0
c                      read r from standard input (terminal)
c      Write(*,*) 'Enter the radius of a circle'
c      Read (*,*)   r
c                      calculate area
c      A = pi * r**2
c                      write area onto terminal screen
c      Write(*, 10) 'radius r = ', r, 'area = ', A
10     Format(a10, f10.5, a10, f12.5)
      Stop 'area'
      End
```

**Listing D.2: bessel.f**

```

c  bessel.f: Spherical Bessel functions via up and down recursion
c              data saved as: x y1 y2

      Program bessel

      Implicit none
c order of Bessel function, x range, stepsize, start for downward alg
      Real*8 step, x, xmin, xmax, up, down, t1, t2
      Integer order, start
      xmin = 0.25
      xmax = 40.0
      step = 0.1
      order = 10
      start = 50
c                                     open output file
      Open(6, File = 'bessel.dat', Status = 'Unknown')           main program
c
      Do 10 x = xmin, xmax, step
          t1 = down(x, order, start)
          t2 = up(x, order)
          Write (6, *) x, t1, t2
10    Continue
      Close(6)
      Stop 'data saved in bessel.dat'
      End
c                                     calculate using downward recursion
      Function down(x, order, start)
      Implicit none
      Integer k, order, start
      Real*8 down, scale, x, j(100)
c                                     arbitrary start
      j(start+1) = 1
      j(start) = 1
      Do 20 k = start, 2, -1
          j(k-1) = ((2*k-1.0)/x)*j(k)-j(k+1)
20    Continue
c                                     scale so that j(1) = sin(x)/x
      scale = (sin(x)/x)/j(1)
      down = j(order+1)*scale
      Return
      End
c                                     calculate using upward recursion
      Function up(x, order)
      Implicit none
      Integer k, order
      Real*8 up, x, one, two, thr
      one = sin(x)/x
      two = (sin(x)-x*cos(x))/(x*x)
      Do 30 k = 1, (order-1)
          thr = ((2*k+1.0)/x)*two-one
          one = two
          two = thr
30    Continue
      up = thr
      Return
      End

```

**Listing D.3: bound.f**

```

c  bound.f: bound states in momentum space of delta shell potential
c      uses LAPACK SGEEV, 16 grid points, l = 0, lambda variable
c      NB: results are NOT stable to continuing increase in Npts!

Program bound

Integer n, Size, job, info, i, j
Real b
Parameter (Size = 100, pi = 3.141592, b = 10.0)
Real lambda, scale
Double Precision Pot
Complex x(Size),V(Size, Size)
Real H(Size,Size), Work(0:Size,0:Size), k(Size), w(Size)
c                                         Enter potential strength
c
c   Write(*,*) 'enter lambda'
c   Read(*,*) lambda
c   Write(*,*) 'enter scaling factor'
c   Read(*,*) scale
c   Write(*,*) 'number of points'
c   Read(*,*) n
c
c   set up Gauss points & weights on the interval [0,inf]
c   scale is the "midpoint" of the integration on the infinite
c
c   Call Gauss(n,2,0.0,scale,k,w)                                Set up V and H
c
DO i = 1,n
  DO j = 1,n
    Pot = -1.0*lambda*SIN(b*k(i))*SIN(b*k(j))
    Pot = Pot/(k(i)*k(j))
    H(i,j) = w(j)*k(j)*k(j)*2.0*Pot/pi
    If (i .EQ. j) Then
      H(i, j) = H(i, j)+k(i)*k(i)
    Endif
  End Do
End Do
job = 1
Call sgeev(h,  Size, n, x, V,Size, Work, job, info)
Write(*,*) info
Do i = 1, n
  Write(*, *) x(i)
End Do
End

```

**Listing D.4: bugs.f**

```

c  bugs.f: Bifurcation diagram for logistic map
c          plot without connecting datapoints with lines

Program bugs

Implicit none
Real*8 m_min, m_max, m, step, y
Integer x
m_min = 1.0
m_max = 4.0
step = 0.01
Open(6, File = 'bugs.dat', Status = 'Unknown')
c           Loop for m values, arbitrary starting value for y
Do 10 m = m_min, (m_max-step), step
    y = 0.5
c           Wait until transients die out
    Do 20 x = 0, 200
        y = m * y * (1 - y)
    20 Continue
c           Record 200 points
    Do 30 x = 201, 401
        y = m*y * (1 - y)
        Write (6, 50) m, y
    30 Continue
    10 Continue
    50 Format (f5.3, f10.6)
    Close(6)
    Stop 'data saved in bugs.dat'
End

```

**Listing D.5: complex.f**

```

c  complex.f: Dealing with complex numbers on a computer
c

Program complex

Implicit none
Complex*16 z, zsqrt, zlog
Real*8 i, pi, phi, x, y, zatan, zatan2
pi = 3.1415926535897932385E0
Write (*,10) 'phi', 'x', 'y', 'sqrt', 'log', 'atan', 'atan2'
Write (*, *) ''
c           loop for angle
    Do 100 i = 0, 2.6, 0.1
        phi = i * pi
c           calculate Cartesian representation
        x = cos(phi)
        y = sin(phi)
        z = cmplx(x, y)
        zsqrt = sqrt(z)
        zlog = log(z)
        zatan = atan(y/x)
        zatan2 = atan2(y, x)
        Write (*, 20) i, '*pi', x, y, zsqrt, 'i', zlog,

```

```

100      Continue
10      Format (a4, 2a9, a14, a18, a14, a10)
20      Format (f3.1, a3, 3f9.4, f8.4, a1, f9.4, f8.4,a1, 2f9.4)
           Stop 'complex'
End

```

#### Listing D.6: decay.f

```

c  decay.f: Spontaneous radioactive decay simulation
c          If compiler complains about drand48, seed48, replace
c          drand48 with rand(seed) and remove the seed48 call
c
c      Program decay

Implicit none
Real*8 r, drand48, lambda
Integer i, j, h, nleft, nloop, start, seed

c Set parameters (decay rate, initial no of atoms, seed), plant seed
lambda = 0.01
start = 1000
seed = 11168
h = 1
nloop = start
nleft = start
call seed48(seed)
Open(6, File = 'decay.dat')
c                                         loop over times and over atoms
Do 20 j = 1, 10000
    Do 10 i = 1, nleft
        r = drand48()

        IF (r .LE. lambda) Then
            nloop = nloop -1
        EndIF
10      Continue
c                                         atom loop ends
        nleft = nloop
        Write (6, *) h, ' ', Real(nleft)/start
        h = h + 1
        If (nleft .eq. 0) Goto 30
20      Continue
30      Close(6)
           Stop 'data saved in decay.dat'
End

```

**Listing D.7: diff.f**

```

c  diff.f: Differentiation; forward, central & extrapolated differnc
c          results saved as x y1 y2 y3

      Program diff

      Implicit None
c          h stepsize for approximation, xrange and xstepsize
      Real*8 f, h, result(3), x, xmin, xmax, xstep

      Open(6, File = 'diff.dat', Status = 'Unknown')
      h = 1.e-5
      xmin = 0.0
      xmax = 7.0
      xstep = 0.01

      Do 10 x = xmin, xmax, xstep
      result(1) = (f(x+h) - f(x))/h
      result(2) = (f(x+h/2) - f(x-h/2))/h
      result(3) = (8*(f(x+h/4)-f(x-h/4))-(f(x+h/2)-f(x-h/2)))/(3*h)
      Write (6, 20) x, result(1), result(2), result(3)
10    Continue
20    Format(F5.3, TR4, F10.8, TR4, F10.8, TR4, F10.8)
      Close(6)
      Stop 'data saved in diff.dat'
      End

c
c          the function to integrate
      Function f(x)
      Implicit none
      Real*8 f, x
      f = cos(x)
      Return
      End

```

**Listing D.8: eqheat.f**

```

c   eqheat.f: Solution of heat equation using with finite differences
c               Output data is saved in 3D grid format used by gnuplot

        Program heat

        Implicit None
        Double Precision cons, ro, sph, thk, u(101,2)
        Integer i, k, max

        Open(9,file = 'eqheat.dat',status = 'Unknown')
c               specific heat, thermal conductivity and density for iron
        sph = 0.113
               thk = 0.12
               ro = 7.8
        cons = thk/(sph*ro)
c               number of iterations
        max = 30000
c               At t = 0 all points are at 100 °C
        Do 10 i = 1,100
               u(i,1) = 100.0
10      Continue
c               endpoints zero
        Do 20 i = 1,2
               u(1,i) = 0.0
               u(101,i) = 0.0
20      Continue
c               start solution, time loop
        Do 100 k = 1,max
c               loop over space, endpoints stay fixed
        Do 30 i = 2,100
               u(i,2) = u(i,1) + cons*(u(i+1,1) + u(i-1,1)-2*u(i,1))
30      Continue
c               output every 1000 time steps
        If ( (Mod(k,1000) .eq. 0) .or. (k .eq. 1) ) Then
               Do 40 i = 1,101,2
                     Write(9,22)u(i,2)
40      Continue
               Write (9,22)
        EndIf
c               new -> old
        Do 50 i = 2,100
               u(i,1) = u(i,2)
50      Continue
100     Continue
22      Format (f10.6)
               Close(9)
        Stop 'data saved in eqheat.dat'
End

```

**Listing D.9: eqstring.f**

```

c  eqstring.f: Solution of wave equation using time stepping
c  comment: Output data is saved in 3D grid format used by gnuplot

      Program string

      Implicit None
      Real*8 x(101,3)
      Integer i, k, max

      max = 100
      Open(9,file = 'eqstring.dat',status = 'Unknown')
c                                     initialize values
      Do 10 i = 1,80
         x(i,1) = 0.00125*i
10   Continue
      Do 20 i = 81,101
         x(i,1) = 0.1-0.005*(i-81)
20   Continue
c                                     the first time step
      Do 30 i = 2,100
         x(i,2) = x(i,1)+0.5*(x(i+1,1)+x(i-1,1)-2.0*x(i,1))
30   Continue
c                                     all other times
      Do 40 k = 1,max
         Do 50 i = 2,100
            x(i,3) = 2.0*x(i,2)-x(i,1)+(x(i+1,2)+x(i-1,2)-2.0*x(i,2))
50   Continue
         Do 60 i = 1,101
c                                     new -> old
            x(i,1) = x(i,2)
            x(i,2) = x(i,3)
60   Continue
c                                     output every 10 steps
      If ( Mod(k,10) .EQ. 0 ) then
         Do 70 i = 1,101
            Write(9,11) x(i,3)
70   Continue
            Write(9,11)
            EndIf
40   Continue
11   Format (e12.6)
      Close(9)
      Stop 'data saved in eqstring.dat'
      End

```

**Listing D.10:** exp-bad.f

**Listing D.11: exp-good.f**

```

c  exp-good.f:  good algorithm for calculating exponential
c                  related programs: exp-bad.f

      Program expgood

      Implicit none
c          limit for accuracy, max in x, step in x
      Real*8 element, min, max, step, sum, x
      Integer n

      min = 1E-10
      max = 10.0
      step = 0.1
      Open(6, File = 'exp-good.dat', Status = 'Unknown')           execution
c
      Do 10 x = 0, max, step
          sum = 1
          element = 1
          Do 20 n = 1, 10000
              element = element*(-x)/n
              sum = sum + element
          if ((abs(element/sum) .lt. min) .AND. (sum .ne. 0)) then
              Write (6,*) x, sum
c                  N.B. since no "while" in f77, need use "goto"
              GoTo 10
          Endif
20      Continue
10      Continue
      Close(6)
      Stop 'data saved in exp-good.dat'
End

```

**Listing D.12: fit.f**

```

c  fit.f: Least-squares fit to decay spectrum

      Program fit

      Implicit none
      Integer i
      Real*8 s, sx, sy, sxx, sxy, delta, inter, slope
      Real*8 x(12), y(12), d(12)
c          input value y, x values
      Data y /32, 17, 21, 7, 88, 6, 5, 2, 2, 80.1, 48, 1/
      Do 10 i = 1, 12
          x(i) = i*10-5
10      Continue
      Do 11 i = 1, 12
          d(i) = 1.0
11      Continue
c          calculate sums
      Do 30 i = 1, 12
          s = s + 1 / (d(i)*d(i))
          sx = sx + x(i) / (d(i)*d(i))

```

```

      sy = sy + y(i) / (d(i)*d(i))
      sxx = sxx + x(i)*x(i) / (d(i)*d(i))
      sxy = sxy + x(i)*y(i) / (d(i)*d(i))
30   Continue
c                                     calculate coefficients
      delta = s*sxx-sx*sx
      slope = (s*sxy-sx*sy) / delta
      inter = (sxx*sy-sx*sxy) / delta
      Write(*,*) 'intercept = ', inter
      Write(*,*) 'slope = ', slope
      Write(*,*) 'correlation = ', -sx/sqrt(sxx*s)
      Stop 'fit'
End

```

#### Listing D.13: fourier.f

```

c fourier.f: Calculates a discrete Fourier Transformation
c program reads input from file input.dat: y(t) separated by
c blanks. Output: frequency index, real part, imaginary part

Program fourier

Implicit none
Integer max
Real*8 pi
Parameter (max = 1000,pi = 3.1415926535897932385E0)
Integer i, j, k
Real*8 input(max), real, imag

Open(9, File = 'fourier.dat', Status = 'Unknown')
c           read from file until end-of-file or max values
Open(8, File = 'input.dat', Status = 'OLD')
Do 10 i = 1,max
    Read(8,* ,End = 20) input(i)
10   Continue
c                                     frequency loop
20   Do 30 j = 1,i
        real = 0
        imag = 0
c                                     sums loop
        Do 40 k = 1,i
            real = real + input(k) * cos( 2*pi*k*j / i )
            imag = imag + input(k) * sin( 2*pi*k*j / i )
40   Continue
        Write (9,*) j, real/i, imag/i
30   Continue
Close(8)
Close(9)
Stop 'data saved in fourier.dat'
End

```

**Listing D.14: gauss.f**

```

c  gauss.f: Points and weights for Gaussian quadrature
c          error message if subroutine called without a main
c          this file must reside in same directory as integ.c
c  npts = number of points
c  job   = 0 rescalling uniformly between (a, b)
c          1 for integral (0, b) with 50% points inside (0, ab/(a+b))
c          2 for integral (a, inf) with 50% inside (a, b+2a)
c  x, w   output integration points and weights.
c
c  subroutine gauss(npts, job, a, b, x, w)

      integer npts, job, m, i, j
      real*8 x(npts), w(npts), a, b, xi
      real*8 t, t1, pp, p1, p2, p3, aj
      real*8 eps, pi, zero, two, one, half, quarter
      parameter (pi = 3.14159265358979323846264338328, eps = 3.E-14)
      parameter (zero = 0.d0, one = 1.d0, two = 2.d0)
      parameter (half = 0.5d0, quarter = 0.25d0)

      m = (npts+1)/2
      do 1020 i = 1, m
         t = cos(pi*(i-quarter)/(npts+half))
1000    continue
         p1 = one
         p2 = zero
         aj = zero
         do 1010 j = 1, npts
            p3 = p2
            p2 = p1
            aj = aj+one
            p1 = ((two*aj-one)*t*p2-(aj-one)*p3)/aj
1010    continue
         pp = npts*(t*p1-p2)/(t*t-one)
         t1 = t
         t = t1-p1/pp
         if ( abs(t-t1) .gt. eps ) goto 1000
         x(i) = -t
         x(npts+1-i) = t
         w(i) = two/((one-t*t)*pp*pp)
         w(npts+1-i) = w(i)
1020    continue
c          rescales the gauss-legendre grid points and weights
c          if (job .eq. 0) then
c                               scale to (a, b) uniformly
         do 1030 i = 1, npts
            x(i) = x(i)*(b-a)/two+(b+a)/two
            w(i) = w(i)*(b-a)/two
1030    continue
         elseif ( job .eq. 1 ) then
c          scale to (0, b) with 50% points inside (0, ab/(a+b))
         do 1040 i = 1, npts
            xi = x(i)
            x(i) = a*b*(one+xi)/(b+a-(b-a)*xi)
            w(i) = w(i)*two*a*b*b/((b+a-(b-a)*xi)*(b+a-(b-a)*xi))
1040    continue

```

```

c
      elseif ( job .eq. 2 ) then
          scale to (a, inf) with 50% points inside (a, b+2a)
      do 1050 i = 1, npts
          xi = x(i)
          x(i) = (b*xi+b+a+a)/(one-xi)
          w(i) = w(i)*two*(a+b)/((one-xi)*(one-xi))
1050    continue
      else
          pause 'Wrong value of job'
      endif
      Return
      end

```

### **Listing D.15: harmos.f**

```

        If ( (n .eq. 1) .or. (Mod(n, 2000) .eq. 0)) Then
          Do 80 i = 1, max+1, 10
            Write(9, 11)p2(i) + 0.0015*v(i)
80        Continue
          Write(9, 11)
        EndIf
c                                               new -> old
        Do 70 i = 1, max+1
          psi(i, 1) = psi(i, 2)
          psr(i, 1) = psr(i, 2)
70        Continue
40        Continue
11        Format(E12.6)
        Close(9)
        Stop 'data saved in harmos.dat for gnuplot'
      End

```

Listing D.16: integ.f

```

c  integ.f: Integrate exp(-x) using trapezoid, Simpson & Gauss rules
c  gauss.f must be included
c  derivation from the exact output as x y1 y2

  Program integrate

    Implicit none
    Real*8 trapez, simpson, quad, r1, r2, r3
    Real*8 theo, vmin, vmax
    Integer i
c                                     theoretical result, integration range
    theo = 0.632120558829
    vmin = 0.
    vmax = 1.
    Open(6, File = 'integ.dat', Status = 'Unknown')
c                                     calc integral using both methods for steps = 3..501
    Do 50 i = 3, 501 , 2
      r1 = trapez(i, vmin, vmax)
      r1 = abs(r1-theo)
      r2 = simpson(i, vmin, vmax)
      r2 = abs(r2-theo)
      r3 = quad(i, vmin, vmax)
      r3 = abs(r3-theo)
      write(6, *) i, r1, r2, r3
50    Continue
    Close(6)
    Stop 'data saved in integ.dat'
  End
c                                     function to integrate
  Function f(x)
    Implicit none
    Real*8 f, x
    f = exp(-x)
    Return
  End
c                                     trapezoid rule
  Function trapez(i, min, max)

```

```

Implicit none
Integer i, n
Real*8 f, interval, min, max, trapez, x
trapez = 0
interval = ((max-min) / (i-1))
c                                     sum midpoints
Do 21 n = 2, (i-1)
    x = interval * (n-1)
    trapez = trapez + f(x)*interval
21 Continue
c                                     add endpoints
trapez = trapez+0.5*(f(min)+f(max))*interval
Return
End
c                                     Simpsons rule
Function simpson(i, min, max)
Implicit none
Integer i, n
Real*8 f, interval, min, max, simpson, x
simpson = 0
interval = ((max-min) / (i-1))
c                                     loop for odd points
Do 31 n = 2, (i-1), 2
    x = interval * (n-1)
    simpson = simpson + 4*f(x)
31 Continue
c                                     loop for even points
Do 32 n = 3, (i-1), 2
    x = interval * (n-1)
    simpson = simpson + 2*f(x)
32 Continue
c                                     add endpoints
simpson = simpson+f(min)+f(max)
simpson = simpson*interval/3
Return
End
c                                     Gauss quadrature
Function quad(i, min, max)
Implicit none
Real*8 w(1000), x(1000)
Real*8 f, min, max, quad
Integer i, job, n
quad = 0
job = 0
call gauss(i, job, min, max, x, w)
Do 41 n = 1, i
    quad = quad+f(x(n))*w(n)
41 Continue
Return
End

```

**Listing D.17: int\_10d.f**

```

c  int_10d.f: 10D integration using Monte Carlo
c          If your compiler complains about drand48, seed48
c          replace drand48 with rand(seed) and remove seed48 call

Program int10d

Implicit none
Real*8 drand48, x, y
Integer i, j, n

Open(6, File = 'int_10d.dat', Status = 'Unknown')
Call seed48(68111)
c          Outer loops determines number of trials = accuracy
Do 10 i = 1, 65536
  x = 0
c          Add 10 random numbers, square, add and save
  Do 20 j = 1, 10
    x = x+drand48()

20      Continue
y = y+x*x
  if (mod(i, 2**n) .eq. 0) then
    n = n+1
    Write (6, *) i, y/i
  endif
10      Continue
  Close(6)
  Stop 'data saved in int_10d.dat'
End

```

**Listing D.18: ising.f**

```

c  ising.f: Ising model of magnetic dipole string
c          If your compiler complains about drand48, seed48
c          replace drand48 with rand(seed) and remove seed48 call
c          Plot without connecting datapoints with lines

Program Ising

Implicit none
Integer max
Parameter(max = 100)
Integer element, i, spins(max), seed, t
Real*8 drand48, energy, kt, new, j, old
c          define number temperature, exchange energy, random seed
Parameter(kt = 100, j = -1, seed = 68111)
c          open files, seed generator

Open(8, file = 'spin-up.dat', Status = 'Unknown')
Open(9, file = 'spin-do.dat', Status = 'Unknown')
Call seed48(seed)
c          generate a uniform configuration of spins
Do 10 i = 1,max

```

```

10      spins(i) = 1
      Continue
c          step through time
Do 20 t = 1, 500
      old = energy(spins, j, max)
      element = drand48() *max+1

c          flip spin
      spins(element) = spins(element)*(-1)
      new = energy(spins, j, max)
c          Metropolis algorithm
      If (new.GT.old .AND. exp((-new+old)/kt) .LT. drand48()) Then
          spins(element) = spins(element)*(-1)
      Endif
      Do 30 i = 1,max
          If (spins(i) .EQ. 1) Then
              Write(8,*) t, i
          Endif
          If (spins(i) .EQ. (-1)) Then
              Write(9,*) t, i
          Endif
      Continue
20      Continue
      Close(8)
      Close(9)
      Stop 'data saved in spin-up.dat, spin-do.dat'
End

Function energy(array, j, max)
Implicit none
Integer array(max), i, max
Real*8 energy, j
energy = 0
Do 22 i = 1,(max-1)
    energy = energy+array(i)*array(i+1)
22 Continue
Return
End

```

**Listing D.19: lagrange.f**

```

c  lagrange.f: Langrange interpolation of cross table
c
      Program lagrange

      Implicit none
      Real*8 inter, x, xin(9), yin(9)
      Integer i, end

      end = 9
      Open(6, File = 'lagrange.dat', Status = 'Unknown')           Input data
c
      Data xin /0, 25, 50, 75, 100, 125, 150, 175, 200/
      Data yin /810.6, 16, 45, 83.5, 52.8, 199., 10.8, 98.25, 48.7/
c
      Do 20 i = 0, 1000                                         Calculate f(x)
         x = i*0.2
         Write (6, *) x, inter(xin, yin, end, x)
20    Continue
         Close(6)
         Stop 'data saved in lagrange.dat'
      End

c
      Function inter(xin, yin, end, x)                           Evaluate interpolation function at x
      Implicit none
      Integer i, j, end
      Real*8 inter, lambda(10), xin(10), yin(10), x
      inter = 0
      Do 200 i = 1, end
         lambda(i) = 1
         Do 300 j = 1, end
            If (i .ne. j) Then
               lambda(i) = lambda(i) *((x - xin(j))/(xin(i) - xin(j)))
            EndIf
300    Continue
         inter = inter + (yin(i) * lambda(i))
200    Continue
         Return
      End

```

**Listing D.20: laplace.f**

```

c  laplace.f: Solution of Laplace equation with finite differences
c              Output in gnuplot 3D grid format used by
      Program laplace

      Implicit none
      Integer max
      Parameter(max = 40)
      Real*8 p(max, max)
      Integer i, j, iter

      Open(8, File = 'laplace.dat', Status = 'Unknown')

```

```

c
      Do 100 i = 1, max
      Do 200 j = 1, max
      p(i, j) = 0.
200    Continue
100    Continue

c                               side with constant potential
      Do 10 i = 1, max
      p(i, 1) = 100.
10     Continue
c                               iteration algorithm
      Do 20 iter = 1, 1000
      Do 30 i = 2, (max-1)
      Do 40 j = 2, (max-1)
      p(i, j) = 0.25*(p(i+1,j)+p(i-1,j) +p(i, j+1) + p(i, j-1))
40     Continue
30     Continue
20     Continue
c                               output in gnuplot 3D format
      Do 50 i = 1, max
      Do 60 j = 1, max
      Write (8, 22) p(i, j)
60     Continue
      Write (8, 22)
50     Continue
22     Format(f10.6)
      Close(8)
      Stop 'data saved in laplace.dat'
End

```

**Listing D.21: limit.f**

```

c  limit.f: determines the machine precision
c
      Program limit

      Implicit none
      Integer I, N
      Real*8 eps, one
c
      N = 60                                number of iterations N
c
      eps = 1                                set initial values
      one = 1
c
      Do 15, I = 1, N
          eps = eps / 2
          one = 1 + eps
          Write (*, *) I, one, eps
15     Continue
      Stop 'limit'
End

```

**Listing D.22: over.f**

```
c  over.f: determine overflow and underflow limits

    Program overflow

        Implicit none
        Integer I, N
        Real*8 under, over
c                           number of iterations N
        N = 1024
c                           set initial values
        under = 1
        over = 1
c                           calculate, print to screen
        Do 15, I = 1, N
            under = under / 2
            over = over * 2
            Write (*, *) I, over, under
15      Continue
            Stop 'over'
        End
```

**Listing D.23: pond.f**

```
c  pond.f: Calculate pi using Monte Carlo integration (throw stones)
c      If your compiler complains about drand48, seed48,
c      replace drand48 with rand(seed) and remove seed48 call

    Program pond

        Implicit none
c      drand48 function, max number of stones, seed for generator
        Real*8 area, x, y, drand48
        Integer i, max, pi, seed

        max = 2000
        seed = 68111
c                           open file, set initial value, seed generator
        Open(6, File = 'pond.dat', Status = 'Unknown')
        pi = 0
        Call seed48(seed)
        Do 10 i = 1, max
            x = drand48()*2-1
            y = drand48()*2-1
            If ((x*x + y*y) .LE. 1) Then
                pi = pi+1
            Endif
            area = 4. * pi/Real(i)
            Write(6, *) i, area
10      Continue
            Close(6)
            Stop 'data saved in pond.dat'
        End
```

**Listing D.24: qmc.f**

```

c  qmc.f: Feynman path integral for ground state wave function
c      If your compiler complains about drand48, srand48,
c          uncomment the define statements further down.

Program qmc

Implicit none
Integer i, j, max, element, prop(100)
Real*8 change, drand48, energy, newE, oldE, out, path(100)

max = 250000
Open(9, file = 'qmc.dat', Status = 'Unknown')
call seed48(68111)

c           initial path and initial probability
Do 10 j = 1, 100
    path(j) = 0.
    prop(j) = 0
10 Continue
c           find energy of initial path
oldE = energy(path, 100)
Do 20 i = 1, max
c           pick one random element
    element = drand48()*100+1
c           change by random value -0.9..0.9
    change = ((drand48() - 0.5)*2)
    path(element) = path(element)+change
c           find the new energy
newE = energy(path, 100)
c           Metropolis algorithm
If ((newE.GT.oldE) .AND. (exp(-newE+oldE).LT.drand48())) Then
    path(element) = path(element)-change
    Endif
c           add up probabilities
Do 30 j = 1, 100
    element = path(j)*10+50
    prop(element) = prop(element)+1
30 Continue
oldE = newE
20 Continue
c           file output
Do 40 j = 1, 100
    out = prop(j)
    Write(9, *) j-50, out/max
40 Continue
Close(9)
Stop 'data saved in qmc.dat'
End

Function energy(array, max)
c           calculate energy of system
Implicit none
Integer i, max
Real*8 energy, array(max)
energy = 0
Do 50 i = 1, (max-1)
    energy = energy + (array(i+1)-array(i))**2 + array(i)**2

```

```
50      Continue
      Return
      End
```

**Listing D.25: random.f**

```
c      random.f: simple random number generator, not for serious work
c
c      Program random

      Implicit none
      Integer i, number, old, seed, x, y
c                           set seed, number generated numbers
      seed = 11
      number = 1000
      Open(6, file = 'random.dat', Status = 'Unknown')
      old = seed
c
c      Do 10 i = 1, number
c           x = Mod((57*old+1), 256)
c           y = Mod((57*x+1), 256)
c           Write (6, *) x, y
c           old = y
10      Continue
      Close(6)
      Stop 'data saved in random.dat'
      End
```

**Listing D.26: rk4.f**

```
c      rk4.f: 4th order Runge-Kutta solution for harmonic oscillator
c
c      Program rk4Program

      Implicit none
      Real*8 dist, min, max, x, y(2)
      Integer n
c n: # eqtns, min/max in x, dist:length of x steps, y(1): x, y(2): v

      n = 2
      min = 0.
      max = 10.
      dist = 0.1
      y(1) = 1.
      y(2) = 0.
      Open(6, File = 'rk4.dat', Status = 'Unknown')
c
c      Do 60 x = min, max, dist
c           Call rk4(x, dist, y, n)
c           Write (6, *) x, y(1)
60      Continue
      Close(6)
      Stop 'data saved in rk4.dat'
      End
c
```

Subroutine rk4

```

Subroutine rk4(x, xstep, y, n)
  Implicit none
  Real*8 deriv, h, x, xstep, y(5)
  Real*8 k1(5), k2(5), k3(5), k4(5), t1(5), t2(5), t3(5)
  Integer i, n
  h = xstep/2.
  Do 10 i = 1, n
    k1(i) = xstep * deriv(x, y, i)
    t1(i) = y(i) + 0.5*k1(i)
10   Continue
  Do 20 i = 1, n
    k2(i) = xstep * deriv(x+h, t1, i)
    t2(i) = y(i) + 0.5*k2(i)
20   Continue
  Do 30 i = 1, n
    k3(i) = xstep * deriv(x+h, t2, i)
    t3(i) = y(i) + k3(i)
30   Continue
  Do 40 i = 1, n
    k4(i) = xstep * deriv(x+xstep, t3, i)
    y(i) = y(i) + (k1(i) + (2.*(k2(i) + k3(i))) + k4(i))/6.
40   Continue
  Return
End
c                                         Return derivatives
Function deriv(x, temp, i)
  Implicit none
  Real*8 deriv, x, temp(2)
  Integer i
  If (i .EQ. 1) deriv = temp(2)
  If (i .EQ. 2) deriv = -temp(1)
  Return
End

```

#### Listing D.27: scatt.f

```

c  scatt.f: scattering phase shift in momentum space from delta
c           shell
c           potential, LU decomposition with partial pivoting.
c           uses gauss.f, LUfactor, LUSolve (included)
c
c Program scatt
c
  Integer n, Size, i, j, Row, Column
  Double Precision b, Pot
  Parameter (Size = 300, pi = 3.1415926535897932384626, b =
             10.)
  Double Precision lambda, scale, ko, Temp, F(Size, Size)
  Double Precision k(Size), w(Size), D(Size), r(Size), V(Size)
  Double Precision L(Size, Size), U(Size, Size), P(Size, Size)
  Integer PivotInfo(Size)
c                                         Input potential strength lambda
  Write(*, *) 'enter lambda'
  Read(*, *) lambda
  Write(*, *) 'enter scaling factor'
  Read(*, *) scale

```

```

        Write(*, *) 'enter ko'
        Read(*, *) ko
        Write(*, *) 'enter grid size'
        Read(*, *) n
c Set up Gaussian pts &wts, intrvl [0, inf], mid-point at 'scale'.
c                                         Set last element in k array to ko
        Call gauss(n, 2, 0d0, scale, k, w)
        Do i = 1, n
            D(i) = 2.d0/pi*w(i)*k(i)*k(i)/(k(i)*k(i)-ko*ko)
        End Do
        D(n+1) = 0.
        Do j = 1, n
            D(n+1) = D(n+1)+w(j)*ko*ko/(k(j)*k(j)-ko*ko)
        End Do
        D(n+1) = D(n+1)*(-2.d0/pi)

c                                         Set up F matrix and V vector
        Do i = 1, n+1
            Do j = 1, n+1
                Pot = -b*b*lambda*SIN(b*k(i))*SIN(b*k(j))
                Pot = Pot/(k(i)*b*k(j)*b)
                F(i, j) = Pot*D(j)
                If (i .EQ. j) Then
                    F(i, j) = F(i, j)+1.d0
                Endif
            End Do
            V(i) = Pot
        End Do

c                                         LU factorization. LU factors of F, store partial pivoting info
        call LUfactor(F, n+1, Size, L, U, PivotInfo)
c                                         Pivot and solve, set P to identity matrix
        Do Row = 1, n+1
            Do Column = 1, n+1
                P(Row, Column) = 0
                If (Row .EQ. Column) P(Row, Column) = 1
            End Do
        End Do
c                                         Interchange rows to get true P matrix
        Do Row = 1, n+1
            Do Column = 1, n+1
                Temp = P(Row, Column)
                P(Row, Column) = P(PivotInfo(Row), Column)
                P(PivotInfo(Row), Column) = Temp
            End Do
        End Do
        Call LUSolve(V, L, U, n+1, Size, PivotInfo, r)
c                                         Output results
        Write(*, *) ko*ko, Datan(-r(n+1)*ko)
    End

c LU factorization, partial pivoting of A in prep for solving Ax = b
Subroutine LUfactor(A, n, Size, L, U, PivotInfo)
    Integer n, Column, CurrentPivotRow, CurrentRow, SwapCol, Row
    Integer ElimCol, Size
    Double Precision A(Size, Size), L(Size, Size), U(Size, Size)
    Integer PivotInfo(Size)

```

```

Double Precision CurrentPivotValue , Swap
Do Column = 1, n-1
    CurrentPivotRow = Column
    CurrentPivotValue = A(CurrentPivotRow , Column)
c           Determine row for largest pivot
Do CurrentRow = Column+1, n
If (DABS(A(CurrentRow , Column)) .GT. CurrentPivotValue) Then
    CurrentPivotValue = DABS(A(CurrentRow , Column))
    CurrentPivotRow = CurrentRow
Endif
End Do
PivotInfo (Column) = CurrentPivotRow
c           Swap rows so largest value at pivot
Do SwapCol = Column, n
Swap = A(Column, SwapCol)
A(Column, SwapCol) = A(PivotInfo (Column) , SwapCol)
A(PivotInfo (Column) , SwapCol) = Swap
End Do
c           Gaussian Elimination
c           Get upper triangular A and un-pivoted lower triangular L
Do Row = Column+1, n
    L(Row, Column) = A(Row, Column)/A(Column, Column)
    Do ElimCol = Column+1, n
        A(Row, ElimCol) = A(Row, ElimCol)
        - L(Row, Column) * A(Column, ElimCol)
    End Do
    End Do
End Do
c           Make sure bottom right value doesnot get pivoted to 0
PivotInfo (n) = n
c           Now pivot the L
Do Row = 2, n-1
    Do Column = 1, Row-1
        Swap = L(Row, Column)
        L(Row, Column) = L(PivotInfo (Row) , Column)
        L(PivotInfo (Row) , Column) = Swap
    End Do
End Do
c           clean up L and U
Do Column = 1, n
    Do Row = 1, Column
        U(Row, Column) = A(Row, Column)
        L(Row, Column) = 0
        IF (Row .EQ. Column) L(Row, Column) = 1
    End Do
Do Row = Column+1, n
    U(Row, Column) = 0
End Do
End Do
Return
End

Subroutine LUSolve(b, L, U, n, Size , PivotInfo , x)
Integer n, Size , Row, Column
Double Precision b(Size) , x(Size)
Integer PivotInfo(Size)
Double Precision L(Size , Size) , U(Size , Size)

```

```

c      Double Precision Temp           Interchange rows of b for pivoting
Do Row = 1, n
      Temp = b(Row)
      b(Row) = b(PivotInfo (Row))
      b(PivotInfo (Row)) = Temp
End Do
c  Solve Ly = b by forward elimination, since L diagonal, y(1) = b(1)
Do Row = 2, n
      Do Column = 1, Row-1
          b(Row) = b(Row)-L(Row, Column)*b(Column)
      End Do
      b(Row) = b(Row)/L(Row, Row)
      End Do
c      solve Ux = y by back substitution
      x(n) = b(n)/U(n, n)
      Do Row = n-1, 1, -1
          x(Row) = b(Row)
          Do Column = Row+1, n
              x(Row) = x(Row)-U(Row, Column)*x(Column)
          End Do
          x(Row) = x(Row)/U(Row, Row)
      End Do
      Return
End

```

**Listing D.28: slit.f**

```

c  slit.f: Solves the time-dependent Schroedinger equation for 2D
c          Gaussian wavepacket entering a slit (which takes some time)
c          Output saved in gnuplot 3D grid format
c
Program slit

Implicit None
Real*8 psr(91, 91, 2), psi(91, 91, 2), v(91, 91), p2(91, 91)
Real*8 a1, a2, dt, dx, k0x, k0y, x0, y0, x, y
Integer i, j, max, n, time
complex exc, zi

c
input pos int proportional to t when want to view wave packet
Write(*, *)'Enter a positive integer from 1 (initial time)'
Write(*, *)'to 800 to get wave packet position at that time'
Read(*, *)time
Write(*, *)'processing data for time', time
Open(9, file = 'slit.dat', status = 'Unknown')
c                                     initialize constants and wave packet
zi = cmplx(0.d0, 1.d0)
dx = 0.2d0
dt = 0.0025/(dx*dx)
c                                     initial momentum, position
k0x = 0.d0
k0y = 2.5d0
x0 = 0.d0
y0 = -7.d0
max = 90

```

```

c                                initial wave function
y = -9.d0
      Do 90 j = 1, max+1
      x = -9.d0
      Do 10 i = 1, max+1
         exc = exp(zi*(k0x*x+k0y*y))
         a1 = exp(-0.5*((x-x0)**2.+((y-y0)**2.))
c           real, imag parts of initial wave function
         psr(i, j, 1) = real(a1*exc)
         psi(i, j, 1) = imag(a1*exc)
         x = x + dx
10    Continue
      y = y + dx
90    Continue
c           set potential slit width = 50-40 = 10 units
c
      Do 220 j = 1, max+1
      Do 190 i = 1, max+1
         If ((j .eq. 35) .and. ((i .le. 40) .or. (i .ge. 51))) Then
            v(i, j) = 0.5
         Else
            v(i, j) = 0.
         EndIf
190   Continue
220   Continue
c           propagate solution through time
      Do 40 n = 1, time
         Do 150 j = 2, max
            Do 50 i = 2, max
c               Re psi and prob
            a2 = v(i, j)*psi(i, j, 1)+2.d0*dt*psi(i, j, 1)
            a1 = psi(i+1,j,1) + psi(i-1,j,1) + psi(i,j+1,1)
                           + psi(i,j-1,1)
            psr(i, j, 2) = psr(i, j, 1) - dt*a1 + 2.*a2
            If ( n .eq. time) Then
               p2(i,j)=psr(i,j,1)*psr(i,j,1)+psi(i,j,1)*psi(i,j,1)
            EndIf
50    Continue
c               derivative = zero at x edge
            psr(1, j, 2) = psr(2, j, 2)
            psr(max+1, j, 2) = psr(max, j, 2)
150   Continue
c               imag psi
            Do 160 j = 2, max
               Do 60 i = 2, max
                  a2 = v(i, j)*psr(i, j, 2)+2.*dt*psr(i, j, 2)
                  a1 = psr(i+1,j,2) + psr(i-1,j,2) + psr(i,j-1,2)
                           + psr(i,j+1,2)
                  psi(i, j, 2) = psi(i, j, 1)+dt*a1-2.*a2
60    Continue
c               derivative = zero at x edge
            psi(1, j, 2) = psi(2, j, 2)
            psi(max+1, j, 2) = psi(max, j, 2)
160   Continue
c               new -> old ones
            Do 180 j = 1, max+1
               Do 70 i = 1, max+1

```

```

        psi(i, j, 1) = psi(i, j, 2)
        psr(i, j, 1) = psr(i, j, 2)
70      Continue
180      Continue
40      Continue
c          output probabilities plus scaled potential
        Do 200 j = 2, max, 3
            Do 210 i = 2, max, 2
                Write(9, 11)p2(i, j)+v(i, j)
210      Continue
        Write(9, 11)
200      Continue
11      Format (E12.6)
        Close(9)
        Stop 'data saved in slit.dat'
End

```

Listing D.29: soliton.f

```

c comment: Output data is saved in 3D grid format used by gnuplot

Program soliton

Implicit None
Real*8 ds, dt, max, mu, eps, u(131, 3)
Parameter(ds = 0.4, dt = 0.1, max = 2000, mu = 0.1, eps = 0.2)
c           delta t, delta x, time steps, mu and eps
Real*8 a1, a2, a3, fac, time
Integer i, j, k

Open (9, file = 'soliton.dat', status = 'Unknown')
c           Initial condition
Do 10 i = 1, 131
    u(i, 1) = 0.5*(1. - tanh(0.2*ds*(i-1)-5.))
10      Continue
c           the endpoints
    u(1, 2) = 1.
    u(1, 3) = 1.
    u(131, 2) = 0.
    u(131, 3) = 0.
    fac = mu*dt/(ds**3.)
    time = dt
c           the first step
Do 20 i = 2, 130
    a1 = eps*dt*(u(i+1, 1)+u(i, 1)+u(i-1, 1))/(ds*6.d0)
    If ((i .gt. 2).and.(i .le. 129)) Then
        a2 = u(i+2, 1)+2.*u(i-1, 1)-2.*u(i+1, 1)-u(i-2, 1)
    Endif
    If ((i .eq. 2) .or. (i .eq. 130)) Then
        a2 = u(i-1, 1)-u(i+1, 1)
    Endif
    a3 = u(i+1, 1)-u(i-1, 1)
    u(i, 2) = u(i, 1)- a1*a3-fac*a2/3.d0
20      Continue
c           other times
Do 30 j = 1, max

```

```

Do 40 i = 2, 130
  a1 = eps*dt*(u(i+1, 2)+u(i, 2)+u(i-1, 2))/(3.d0*ds)
  If ((i .gt. 2).and.(i.le.129)) Then
    a2 = u(i+2, 2)+2.d0*u(i-1, 2)-2.d0*u(i+1, 2)-u(i-2, 2)
  Endif
  If ((i .eq. 2) .or. (i .eq. 130)) Then
    a2 = u(i-1, 2)-u(i+1, 2)
  Endif
  a3 = u(i+1, 2)-u(i-1, 2)
  u(i, 3) = u(i, 1)- a1*a3-2.d0*fac*a2/3.d0
  u(1, 3) = 1.d0
40 Continue
c                                         new -> old
Do 50 k = 1, 131
  u(k, 1) = u(k, 2)
  u(k, 2) = u(k, 3)
50 Continue
c                                         output every 200 steps
If (Mod(j, 200) .eq. 0) Then
  Do 60 k = 1, 131
    Write(9, 22)u(k, 3)
60 Continue
  Write(9, 22)
EndIf
time = time + dt
30 Continue
22 Format(f10.6)
Close(9)
Stop 'data saved in soliton.dat'
End

```

**Listing D.30: spline.f**

```

c  spline.f: uses the SLATEC routines DBINT4 & DBVALU to interpolate
c            a set of x-y values using cubic splines
c            you need the SLATEC library compiled as libslatec.a

      Program spline

      Implicit none
      Integer NDATA
      Parameter(NDATA = 5)
      Real*8 BCOEF(NDATA), X(NDATA), Y(NDATA), T(NDATA+4)
      Real*8 W(5*(NDATA+2)), W2(3*4)
      Real*8 DBVALU, FBCL, FBCR, IN, VAL
      Integer I, IBCL, IBCR, IDERIV, INBV, K, N, KNOTOPT

      Open(6, File = 'spline.dat', Status = 'Unknown')           input values
c
      X(1) = 1.
      X(2) = 2.
      X(3) = 3.
      X(4) = 4.
      X(5) = 5.
      Y(1) = 2.
      Y(2) = 4.1
      Y(3) = 3.
      Y(4) = 5.
      Y(5) = 2.
c
      natural splines, set second derivatives at end points to zero
      IBCL = 2
      IBCR = 2
      FBCL = 0.
      FBCR = 0.
      KNOTOPT = 1
c
      Call DBINT4 (X, Y, NDATA, IBCL, IBCR, FBCL, FBCR, KNOTOPT,
      +             T, BCOEF, N, K, W)           find the spline coefficients
c
      INBV = 1                         initialization
      IN = 1.
      IDERIV = 0
      Do 10 I = 1, 101
          VAL = DBVALU (T, BCOEF, N, K, IDERIV, IN, INBV, W2)
          Write (6, *) IN, VAL
          IN = IN+0.04
10     Continue
      Close(6)
      Stop 'data saved in spline.dat'
      End

```

**Listing D.31: sqwell.f**

```

c  sqwell.f: Solves the time-dependent Schroedinger equation for a
c            Gaussian wavepacket in a infinite square well potential
c            Output data in gnuplot 3D format

```

## Program sqwell

```

Implicit None
Real*8 psr(751, 2), psi(751, 2), p2(751)
Real*8 dx, k0, dt, x, pi
Integer i, n, max
Complex exc, zi
Common /values/dx, dt
Open(9, file = 'sqwell.dat', status = 'Unknown')
max = 750
pi = 3.14159265358979323846
zi = CMPLX(0.d0, 1.d0)
dx = 0.02d0
k0 = 17.d0*pi
dt = dx*dx

c                                initial conditions
c
x = 0.
Do 10 i = 1, max+1
   exc = exp(zi*k0*x)
c                                real, imag initial wave function
   psr(i, 1) = real(exc*exp(-0.5*(2.*((x-5.)**2.)))
   psi(i, 1) = imag(exc*exp(-0.5*(2.*((x-5.)**2.)))
   x = x + dx
10 Continue
c                                propagate solution through time
c
Do 40 n = 1, 6000
c                                real psi and probability
   Do 50 i = 2, max
      psr(i, 2) = psr(i, 1) - dt*(psi(i+1, 1) + psi(i-1, 1)
      -2.*psi(i, 1))/(2.*dx*dx)
      p2(i) = psr(i, 1)*psr(i, 2)+psi(i, 1)*psi(i, 1)
50 Continue
c                                imaginary part of the wave function
   Do 60 i = 2, max
      psi(i, 2) = psi(i, 1) + dt*(psr(i+1, 2) + psr(i-1, 2)
      -2.d0*psr(i, 2))/(2.d0*dx*dx)
60 Continue
c                                output at certain times
   If ( Mod(n, 300) .eq. 0) Then
      Do 80 i = 1, max+1, 15
         write(9, 11)p2(i)
80 Continue
      write(9, 11)
      EndIf
c                                new -> old
   Do 70 i = 1, max+1
      psi(i, 1) = psi(i, 2)
      psr(i, 1) = psr(i, 2)
70 Continue
40 Continue
11 Format (E12.6)
      Close(9)
      Stop 'data saved in sqwell.dat'
End

```

**Listing D.32: twodsol.f**

```

c  twodsol.f:  Solves the sine-Gordon equation for a 2D soliton
c
      Program twodsol

      Implicit None
      Double Precision u(201, 201, 3)
      Integer nint

      Open(9, file = 'twodsol.dat', status = 'new')
      Write(*, *)' Enter an integer from 1 to 100'
      Write(*, *)' this number is proportional to time'
      Write(*, *)' time = 0 is for the integer = 1'
      Read(*, *)nint
      Write(*, *)'working with input = ', nint
c                                     initialization
      Call initial(u)
c                                     solution at time proportional to nint
      Call solution(u, nint)
      Stop
End

Subroutine initial(u)

c           initializes the constants and 2-D soliton
      Implicit None
      Double Precision u(201,201,3), dx, dy, dt, xx, yy, dts, time
      Integer i, j
      Common /values/dx, dy, dt, time, dts
c           initial condition for all xy values
      dx = 14.d0/200.d0
      dy = dx
      dt = dx/dsqrt(2.d0)
      dts = (dt/dx)**2
      yy = -7.d0
      time = 0.d0
      Do 10 i = 1, 201
          xx = -7.d0
          Do 20 j = 1, 201
              u(i, j, 1) = 4.d0*Datan(3.d0-sqrt(xx*xx+yy*yy))
              xx = xx+dx
20      Continue
      yy = yy+dy
10      Continue
      Return
End

Subroutine solution(u, nint)

c           solves SGE for initial conditions in routine initial.
      Implicit None
      Double Precision u(201,201,3), dx, dy, dt, time, a2,zz, dts, a1
      Integer l, m, mm, k, j, i, nint
      Common /values/dx, dy, dt, time, dts
c           these values passed by routine initial
      time = time+dt

```

```

c      2nd iteration using d phi/dt = 0 at t = 0 (G(x, y, 0) = 0)
c      and d U/dx = 0 at -x0, x0, -y0 and y0
Do 80 l = 2, 200
    Do 90 m = 2, 200
        a2 = u(m+1, 1, 1)+u(m-1, 1, 1)+u(m, l+1, 1)+u(m, l-1, 1)
        u(m, l, 2) = 0.5*(dts*a2-dt*dt*Dsin(0.25*d0*a2))
90    Continue
80    Continue
c                                borders in second iteration
Do 130 mm = 2, 200
    u(mm, 1, 2) = u(mm, 2, 2)
    u(mm, 201, 2) = u(mm, 200, 2)
    u(1, mm, 2) = u(2, mm, 2)
    u(201, mm, 2) = u(200, mm, 2)
130   Continue
c                                still undefined terms
u(1, 1, 2) = u(2, 1, 2)
u(201, 1, 2) = u(200, 1, 2)
u(1, 201, 2) = u(2, 201, 2)
u(201, 201, 2) = u(200, 201, 2)
Do 100 k = 1, nint
    Do 60 l = 2, 200
        Do 70 m = 2, 200
            a1 = u(m+1, 1, 2)+u(m-1, 1, 2)+u(m, l+1, 2)+u(m, l-1, 2)
            u(m, 1, 3) = -u(m, 1, 1)+dts*a1-dt*dt*Dsin(0.25*d0*a1)
            u(m, 1, 3) = u(m, 2, 3)
            u(m, 201, 3) = u(m, 200, 3)
70    Continue
60    Continue
Do 140 mm = 2, 200
    u(mm, 1, 3) = u(mm, 2, 3)
    u(mm, 201, 3) = u(mm, 200, 3)
    u(1, mm, 3) = u(2, mm, 3)
    u(201, mm, 3) = u(200, mm, 3)
140   Continue
u(1, 1, 3) = u(2, 1, 3)
u(201, 1, 3) = u(200, 1, 3)
u(1, 201, 3) = u(2, 201, 3)
u(201, 201, 3) = u(200, 201, 3)
c                                new i-> old
Do 110 l = 1, 201
    Do 120 m = 1, 201
        u(l, m, 1) = u(l, m, 2)
        u(l, m, 2) = u(l, m, 3)
120   Continue
110   Continue
c                                output at times proportional to nint
IF (k .eq. nint) Then
Do 30 i = 1, 201, 5
    Do 40 j = 1, 201, 5
        zz = Dsin(u(i, j, 3)/2.d0)
        Write(9, *)zz
40    Continue
c      the three xxx are to separate spatial rows for 3D plotting,
c      they must be replaced by blank lines (not carriage Returns)
        Write(9, *)'xxx'
30    Continue

```

```

        EndIF
        time = time+dt
100    Continue
        Return
End

```

**Listing D.33: walk.f**

```

c  walk.f: random walk simulation
c      If your compiler complains about drand48, seed48,
c      replace drand48 with rand(seed) and remove call to seed48
c      Data output as sqrt(steps), distance
c
c      Program walk
      Implicit none
      Real*8 drand48, root2, theta, x, y, r(1:10000)
      Integer i, j, max, seed
c
c          set parameters (# of steps)
      max = 10000
      seed = 11168
      root2 = 1.4142135623730950488E0
c
c          open file, seed generator
      Open(6, file = 'walk.dat', Status = 'Unknown')
      Call seed48(seed)
c
c          clear array
      Do 1 j = 1, max
         r(j) = 0
1     Continue
c
c          average over 100 trials
      Do 10 j = 1, 100
         x = 0
         y = 0
c
c          take max steps
      Do 20 i = 1, max
         x = x + (drand48() -0.5)*2.*root2
         y = y + (drand48() -0.5)*2.*root2
         r(i) = r(i)+ Sqrt(x * x + y * y)
20    Continue
10    Continue
c
c          output data for plot of r vs. sqrt(N)
      Do 30 i = 1, max
         Write (6, *) Sqrt(Real(i)), ' ', r(i)/100
30    Continue
      Close(6)
      Stop 'data saved in walk.dat'
End

```

## E

### C Language Codes

(Alphabetic order modified somewhat to avoid awkward continuations.)

**Listing E.1: bugs.c**

```
// bugs.c: Bifurcation diagram for logistic map
// comment: plot without connecting datapoints with lines

#include<stdio.h>

#define m_min 0.0          // minimum for m
#define m_max 4.0          // maximum for m
#define step 0.01           // stepsize for m

main() {
    double m, y;
    int i;

    FILE *output;          // save data in bugs.dat
    output = fopen("bugs.dat","w");

    for (m=m_min; m<=m_max; m+=step) {           // loop for m
        y = 0.5;                                // arbitrary starting value
                                                // ignore transients
        for (i=1; i<=200; i++) y = m*y*(1-y);      // then record 200 points
        for (i=201; i<=401; i++) {
            y = m*y*(1-y);
            fprintf(output, "%.4f\t%.4f\n", m, y);
        }
    }
    printf("data stored in bugs.dat.\n");
    fclose(output);
}
```

**Listing E.2: area.c**

```
// area.c: Area of a circle , sample I/O

#include <stdio.h>
#define PI 3.1415926535897932385 E0
main() {
    double radius , area;                                // declare variables

    printf("Enter the radius of a circle \n");           // ask for radius
    scanf("%lf" , &radius);                             // read in radius
    area = radius * radius * PI;                        // area formula
    printf("radius=%f, area=%f\n" , radius , area);     // print results
}
```

**Listing E.3: bessel.c**

```
// bessel.c: Spherical Bessel functions via up and down recursion

#include <stdio.h>
#include <math.h>

#define xmax 40.0                                     // max of x
#define xmin 0.25                                     // min of x >0
#define step 0.1                                      // delta x
#define order 10                                       // order of Bessel function
#define start 50                                       // used for downward algorithm

main() {
    double x;
    double down(double x, int n, int m);             // downward algorithm
    double up(double x, int n);                      // upward algorithm

    FILE *out;
    out = fopen("bessel.dat","w");

    for (x=xmin; x<=xmax; x+=step) fprintf(out, "%f\t%f\t%f\n",
                                                x, down(x,order,start), up(x,order));
    printf("data stored in bessel.dat.\n");
    fclose(out);
}

double down (double x, int n, int m) {                // downward recursion
    double scale , j[start+2];
    int k;
    j[m+1] = j[m] = 1.;                               // start with "something"
    for (k=m; k>0 ; k--)
        j[k-1] = ((2.*(double)k + 1.)/x)*j[k] - j[k+1]; // recur
        scale = ((sin(x))/x)/j[0];                     // scale the result
    return(j[n]*scale);
}

double up (double x, int n) {
```

```
// function using upward recursion
double one, two, thr;
int k;
one = (sin(x))/x; // start with lowest order
two = (sin(x) - x*cos(x))/(x*x); // loop for order of function
for (k = 1 ; k<n ; k+=1) {
    thr = ((2.*k + 1.)/x)*two - one; // recurrence relation
    one = two;
    two = thr;
}
return (thr);
}
```

Listing E.4: call.c

```
// call.c: Creates pseudo-random numbers using drand48 or rand
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// if you don't have drand48 uncomment the following two lines
// #define drand48 1.0/RAND_MAX*rand
// #define srand48 srand

main() {
    int i, seed;
    double x;

    printf("enter seed\n"); // user plants seed
    scanf("%i", &seed);
    srand48(seed); // seed drand 48
    for (i=1; i<=10; i++) {
        x = drand48(); // random number between 0 and 1
        printf("Your random number is: %f\n", x);
    }
}
```

**Listing E.5: bound.c**

```

// bound.c: Bound states in momentum space of delta shell potential
/* using the LAPACK dgeev and Gaussian integration
* comment: LAPACK has to be installed on your system and the file
*          gauss.c has to be in the same directory
*          The energy eigenvalue is printed to standard output and
*          the corresponding eigenvector to the file bound.dat
*/
#include <stdio.h>
#include <math.h>
#include "gauss.c"

#define min 0.0                                // integration limits
#define max 200.0
#define size 64                                 // grid points
#define lambda -1.0                            // parameters for potential
#define u 0.5
#define b 2.0
#define PI 3.1415926535897932385E0

main() {
    int i, j, c1, c2, c5, ok;
    char c3, c4;
    double A[size][size], AT[size*size];           // hamiltonian
    double V[size][size];                          // potential
    double WR[size], WI[size];                    // eigenvalues
    double VR[size][size], VL[1][1];              // eigenvectors
    double WORK[5*size];                         // work space
    double k[size], w[size];                      // points, weights

    FILE *out;                                     // save data in bound.dat
    out = fopen("bound.dat", "w");
    gauss(size, 0, min, max, k, w);               // call gauss integration
                                                    // set up hamiltonian matrix
    for (i=0; i<size; i++) {
        for (j=0; j<size; j++) {
            VR[i][j] = (lambda*b*b/(2*u))*(
                (sin(k[i]*b)/(k[i]*b))*(sin(k[j]*b)/(k[j]*b)));
            if (i==j) A[i][j] = k[i]*k[i]/(2*u)
                        + (2/PI)*VR[i][j]*k[j]*k[j]*w[j];
            else     A[i][j]=(2/PI)*VR[i][j]*k[j]*k[j]*w[j];
        }
    }                                              // transform matrix for fortran routine
    for (i=0; i<size; i++) {
        for (j=0; j<size; j++) AT[j+size*i] = A[j][i];
    }
    c1 = size;                                     // we have to do this so we can
    c3 = 'N';                                      // pass pointers to the lapack
    c4 = 'V';                                      // routine
    c5 = 5*size;                                    // for unreferenced arrays
    c2=1;                                         // for unreferenced arrays
    // call for AIX
    dgeev(&c3,&c4,&c1,AT,&c1,WR,WI,VL,&c2,VR,&c1,WORK,&c5,&ok);
}

```

```

// call for DEC
// dgeev_(&c3,&c4,&c1,AT,&c1,WR,WI,VL,&c2,VR,&c1,WORK,&c5,&ok);
if (ok == 0) { // look for bound state
    for (j=0; j<size; j++) {
        if (WR[j]<0) {
            printf("The eigenvalue of the bound state is\n");
            printf("\tlambda= %f\n", WR[j]);
            for (i=0; i<size; i++) fprintf(out,"%d\t%e\n", i, VR[j][i]);
            break;
        }
    }
    printf("eigenvector saved in bound.dat\n");
    fclose(out);
}

```

Listing E.6: column.c

```

// column.c: Correlated ballistic deposition to form fractalcomment :

#include <stdio.h>
#include <stdlib.h>

// if you don't have drand48 uncomment the following two lines
#define drand48 1.0/RAND_MAX*rand
#define srand48 srand

#define max 100000 // number of iterations
#define npoints 200 // no. of open spaces
#define seed 68111 // seed for number generator

main() {
    int i, hit[200], dist, r, x, y, oldx, oldy;
    double pp, prob;
    FILE *output;
    output = fopen("column.dat","w"); // seed random generator
    srand48(seed);
    for (i=0; i<npoints; i++) hit[i] = 0; // clear the array
    oldx = 100;
    oldy = 0;
    for (i=1; i<=max; i++) {
        r = (int) (npoints*drand48());
        x = r-oldx;
        y = hit[r]-oldy;
        dist = x*x + y*y;
        if (dist == 0) prob = 1.0; // probability of sticking
        else prob = 9.0/(dist); // nu=-2.0, c=0.9
        pp = drand48();
        if (pp < prob) {
            if ((r>0) && (r<(npoints-1))) {
                if ((hit[r] >= hit[r-1]) && (hit[r] >= hit[r+1])) hit[r]++;
                else if (hit[r-1] > hit[r+1]) hit[r] = hit[r-1];
                else hit[r] = hit[r+1];
            }
            oldx = r;
            oldy = hit[r];
            fprintf(output, "%d\t%d\n", r, hit[r]);
        }
    }
}

```

```
    }
}
printf("data stored in column.dat\n");
fclose(output);
}
```

### **Listing E.7: decay.c**

```

// decay.c: Spontaneous radioactive decay simulation

#include <stdio.h>
#include <stdlib.h>

// if you don't have drand48 uncomment the following two lines
// #define drand48 1.0/RAND_MAX*rand
// #define srand48 srand
#define lambda 0.01                                // the decay constant
#define max 1000                                    // number of atoms at t=0
#define time_max 500                                // time range
#define seed 68111                                   // seed for number generator

main() {
    int atom, time, number, nloop;
    double decay;

    FILE *output;                                  // save data in decay.dat
    output = fopen("decay.dat", "w");
    number = nloop = max;                          // initial value
    srand48(seed);                                // seed number generator
    srand48(seed);                                // time loop
    for (time=0; time<=time_max; time++) {         // atom loop
        for (atom=1; atom<=number; atom++) {
            decay = drand48();
            if (decay < lambda) nloop--;
        }
        number = nloop;
        fprintf(output, "%d\t%f\n", time, (double)number/max);
    }
    printf("data stored in decay.dat\n");
    fclose(output);
}

```

**Listing E.8: diff.c**

```

//  diff.c: Differentiation with forward, central and extrapolated
//  comment: results saved as x y1 y2 y3

#include <stdio.h>
#include <math.h>
#define h 1e-5           // stepsize for all methods
#define xmax 7           // range for calculation
#define xmin 0
#define xstep 0.01        // stepsize in x

main() {

    double dc, result, x;
    double f(double);           // function we differentiate

    FILE *output;               // save data in diff.dat
    output = fopen("diff.dat","w");
    for (x = xmin; x<= xmax; x+= xstep) {
        fprintf(output,"%f\t", x);          // forward difference
        result = (f(x+h)-f(x))/h;
        fprintf(output, "%.10f\t", result);   // central difference
        result = (f(x+h/2)-f(x-h/2))/h;
        fprintf(output, "%10f\t", result);
        result = (8*(f(x+h/4)-f(x-h/4))-(f(x+h/2)-f(x-h/2)))/(3.*h);
        fprintf(output, "%10f\n", result);    // extrapolated diff
    }
    printf("data stored in diff.dat\n");
    fclose(output);
}                                // end of main

double f(double x)                // function to differentiate
{ return(cos(x)); }

```

**Listing E.9: dla.c**

```
// dla.c: Diffusion-limited aggregation simulation (fractals)

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// if you don't have drand48 uncomment the following two lines
// #define drand48 1.0/RAND_MAX*rand
// #define srand48 srand
#define max 40000                                // number of iterations
#define size 401                                  // size of grid array
#define PI 3.1415926535897932385E0
#define seed 68111                                 // seed for number generator

main() {
    double angle, rad = 180.0;
    int i, j, x, y, dist, step, trav;
    int grid[size][size], hit;
    int gauss_ran();                                // gaussian random number

    FILE *output;                                   // save data in dla.dat
    output = fopen("dla.dat", "w");
    // clear
    for (i=0; i<size; i++) for (j=0; j<size; j++) grid[i][j] = 0;
    grid[200][200] = 1;                            // one particle at the center
    // seed number generator
    srand48(seed);
    for (i=0; i<max; i++) {                      // choose starting point
        hit = 0;
        angle = (2*PI*drand48());                  // random angle
        x = (200+rad*cos(angle));                  // coordinates
        y = (200+rad*sin(angle));
        dist = gauss_ran();                         // random number gaussian dist
        // move forwards or backwards
        if (dist<0) step = -1;
        else step = 1;
        trav=0;
        while((hit == 0) && (x<399) && (x>1) && (y<399) && (y>1) &&
               (trav<abs(dist))) {
            if (grid[x+1][y]+grid[x-1][y]+grid[x][y+1]+grid[x][y-1]>= 1) {
                hit = 1;
                grid[x][y] = 1;                      // particle sticks, walk is over
            }
            else if (drand48() < 0.5) x+=step;      // move horizontally
            else                           y+=step;      // move vertically
            trav++;
        }
    }
    for (i=0; i<size; i++) for (j=0; j<size; j++) // print grid
        if (grid[i][j] == 1) fprintf(output, "%d\t%d\n", i, j);
    printf("data stored in dla.dat\n");
    fclose(output);
}
```

```

}

int gauss_ran() {
    // Box-Mueller method for gaussian random numbers
    double fac, rr, r1, r2;
    static int old = 0;           // have to be static so information
    static int mem;              // survives between function calls
    if (old == 0) {               // no random number left
        // choose random point in unit circle
        do {
            r1 = 2.0*drand48() - 1.0;
            r2 = 2.0*drand48() - 1.0;
            rr = r1*r1+r2*r2;
        }
        while ((rr>=1) || (rr==0));
        fac = sqrt(-2*log(rr)/rr);
        mem = 5000*r1*fac;           // save for next call
        old = 1;                     // set flag
        return ((int)(5000*r2*fac));
    } else {                      // return second number
        old = 0;                   // unset flag
        return mem;                // return number from last call
    }
}

```

Listing E.10: eqheat.c

```

// eqheat.c: Solution of heat equation using with finite differences

#include <stdlib.h>
#include <stdio.h>
#define size 101                         // grid size
#define max 30000                         // iterations
#define thc 0.12                           // thermal conductivity
#define sph 0.113                          // specific heat
#define rho 7.8                            // density

main() {
    int i,j;
    double cons, u[101][2];

    FILE *output;                         // save data in eqheat.dat
    output= fopen("eqheat.dat","w");       // t = 0 points are 100C
    for (i = 0; i<size; i++) u[i][0] = 100.; // except the endpoints
    for (j = 0; j<2; j++) {
        u[0][j] = 0.;
        u[size-1][j]= 0.;
    }
    cons = thc/(sph*rho);                // material constants
    for (i = 1; i<= max; i++) {          // loop over max timesteps
        // loop over space
    }
}

```

```

for (j = 1; j <(size-1); j++)
{u[j][1]= u[j][0]+cons*(u[j+1][0]+u[j-1][0]-2.0*u[j][0]); }
                                // save every 1000 steps
if ((i%1000 == 0) || (i == 1)) {
    for (j = 0 ; j<size; j++) fprintf(output, "%f\n", u[j][1]);
    fprintf(output, "\n");           // empty line for gnuplot
}
    for (j = 0; j<size; j++) u[j][0] = u[j][1];           // new to old
}
printf("data stored in eqheat.dat\n");
fclose(output);
}

```

Listing E.11: eqstring.c

```

// eqstring.c: Solution of wave equation using time stepping
// comment: Output data is saved in 3D grid format used by gnuplot

#include <stdio.h>
#include <math.h>
#define rho 0.01                         // density per length
#define ten 40.0                          // tension
#define max 100                           // time steps

main() {
    int i,k;
    double x[101][3];

    FILE *out;                           // save data in string.dat
    out = fopen("eqstring.dat","w");      // initial config
    for (i = 0; i<81; i++) x[i][0]= 0.00125*i;
    for (i = 81; i<101; i++) x[i][0]= 0.1-0.005*(i-80);        // 1st step
    for (i = 1; i<100; i++)
        {x[i][1]= x[i][0]+0.5*(x[i+1][0]+x[i-1][0]-2.0*x[i][0]);}
                                // all later time steps
    for (k = 1; k<max; k++) {
        for (i = 1; i<100; i++) x[i][2]= 2.*x[i][1]-x[i][0]
                                    +(x[i+1][1]+x[i-1][1]-2.*x[i][1]);
        for (i = 0; i<101; i++)
            x[i][0]= x[i][1];
            x[i][1]= x[i][2];
        }
        if ((k%5) == 0) {                  // print every 5th point
            for (i = 0; i<101; i++) fprintf(out, "%f\n",x[i][2]);
            fprintf(out, "\n");           // empty line for gnuplot
        }
    }
printf("data stored in eqstring.dat\n");
fclose(out);
}

```

**Listing E.12: exp-bad.c**

```
// exp-bad.c: A bad algorithm for calculating exponential
// related programs: exp-good.c

#include <stdio.h>
#include <math.h>
#define min 1E-10                                // limit for accuracy
#define max 10.                                     // maximum for x
#define step 0.1                                    // intervals

main () {
    double sum, x, up, down;
    int i, j;

    FILE *output;
    output = fopen("exp-bad.dat", "w");           // save results in
                                                    // exp-bad.dat

    for (x = 0.0; x <= max; x += step) {          // step through x
        sum = 1.;                                  // reset variables
        i = 0;

        do {
            i++;
            up = down = 1;                          // reset variables
            for (j = 1; j <= i; j++) {             // numerator
                up *= -x;
                down *= j;                         // denominator
            }
            sum += up/down;
        }
        while ((sum == 0) || ((fabs((up/down)/sum)) > min));
        fprintf(output, "%f\t%e\n", x, sum);
    }
    printf("results saved in exp-bad.dat\n");
    fclose(output);
}
```

**Listing E.13: exp-good.c**

```
// exp-good.c: A good algorithm for calculating exponential
// related programs: exp-bad.c.c

#include <stdio.h>
#include <math.h>
#define min 1E-10                                // limit for accuracy
#define max 10.                                     // maximum for x
#define step 0.1                                    // interval

main () {
    double x, sum, element;
    int n;

    FILE *output;                                 // file output
```

```

output = fopen("exp-good.dat", "w");
for (x = 0.0; x<= max; x+= step) {
    sum= element= 1.;                                // reset variables
    n = 0;                                         // sum till accuracy reached
    do {
        n++;
        element *= -x/n;                            // calculate next element
        sum += element;
    }
    while ((sum == 0) || (fabs(element/sum) > min));
    fprintf(output, "%f\t%e\n", x, sum);
}
printf("results saved in exp-good.dat\n");
fclose(output);
}

```

**Listing E.14: film.c**

```

// film.c: Ballistic deposition simulation (fractal)
// Plot data without connecting datapoints with lines
#include <stdio.h>
#include <stdlib.h>

// if you don't have drand48 uncomment the following two lines
#define drand48 1.0/RAND_MAX*rand
#define srand48 srand
#define max 30000                                // number of iterations
#define seed 68111                                // seed for number generator

main() {
    int i, hit[200], r;

    FILE *output;                                // save data in film.dat
    output= fopen("film.dat","w");
    srand48(seed);                             // clear array
    for (i = 0; i<200; i++) hit[i]= 0;
    for (i = 1; i<= max; i++) {
        r= (int)(199*drand48());                // r = 0..199
        if ((hit[r] >= hit[r-1]) && (hit[r] >= hit[r+1])) hit[r]++;
        else if (hit[r-1] > hit[r+1]) hit[r]= hit[r-1];
        else hit[r]= hit[r+1];
        fprintf(output, "%d\t%d\n", r, hit[r]);
    }
    printf("data stored in film.dat\n");
    fclose(output);
}

```

**Listing E.15: fern.c**

```

//  fern.c: Create fractal , fern-like pattern

#include <stdio.h>
#include <stdlib.h>

// if you don't have drand48 uncomment the following two lines
#define drand48 1.0/RAND_MAX*rand
#define srand48 srand
#define max 30000           // number of iterations
#define seed 68111          // seed for number generator

main() {
    int i;
    double x, y, xn, yn, r;

    FILE *output;           // save data in fern.dat
    output = fopen("fern.dat","w");
                           // seed number generator
    srand48(seed);         // starting point

    x = 0.5;
    y = 0.0;
    for (i = 1; i<= max; i++) {           // iterations
        r = drand48();                  // case 1
        if (r <= 0.02) {
            xn= 0.5;
            yn= 0.27*y;
        }
                           // case 2
        else if ((r>0.02) && (r<= 0.17)) {
            xn= -0.139*x + 0.263*y + 0.57;
            yn= 0.246*x + 0.224*y - 0.036;
        }
                           // case 3
        else if ((r>0.17) && (r<= 0.3)) {
            xn= 0.17*x - 0.215*y + 0.408;
            yn= 0.222*x + 0.176*y + 0.0893;
        }
                           // case 4
        else {
            xn= 0.781*x + 0.034*y + 0.1075;
            yn= -0.032*x + 0.739*y + 0.27;
        }
        fprintf(output, "%f %f\n", x, y);
        x = xn;
        y = yn;
    }
    printf("data stored in fern.dat\n");
    fclose(output);
}

```

**Listing E.16: fit.c**

```
//    fit.c:      Least-squares fit

#include <stdio.h>
#include <math.h>
#define data 12           // number of data points

main() {
    int i, j;
    double s, sx, sy, sxx, sxy, delta, inter, slope;
    double x[data], y[data], d[data];
                                // input data
    for (i = 0; i<data; i++) x[i] = i*10+5;
    y[0] = 382; y[1] = 187; y[2] = 281; y[3] = 87;
    y[4] = 8;   y[5] = 6;   y[6] = 5;   y[7] = 28;
    y[8] = 2;   y[9] = 80.1; y[10] = 4; y[11] = 18;
                                // reset sums
    for (i = 0; i<data; i++) d[i] = 1.;

    s = sx = sy = sxx = sxy = 0;
                                // calculating sums
    for (i = 0;i<data;i++) {
        s += 1 / (d[i]*d[i]);
        sx += x[i] / (d[i]*d[i]);
        sy += y[i] / (d[i]*d[i]);
        sxx += x[i]*x[i] / (d[i]*d[i]);
        sxy += x[i]*y[i] / (d[i]*d[i]);
    }
    delta= s*sxx-sx*sx;
    slope= (s*sxy-sx*sy) / delta;           // calculating all
    inter = (sxx*sy-sx*sxy) / delta;         // coefficients
    printf("intercept = %f\t +/- %f\n", inter, sqrt(sxx/delta));
    printf("slope = %f\t +/- %f\n", slope, sqrt(s/delta));
    printf("correlation = %f\n", -sx/sqrt(sxx*s));
}
```

**Listing E.17: fourier.c**

```

// fourier.c: Discrete Fourier Transformation
/* comment: The program reads its input data from a file in the
 * same directory called input.dat. This file has to contain
 * only y(t) values separated by whitespaces which are real
 * The output is the direct output from the algorithm which
 * will probably look very different than what you are used
 * to. The output has the form
 *      frequency index \t real part \t imaginary part
 * related programs: invfour.c */
#include <stdio.h>
#include <math.h>
#define max 1000      // max number of input data
#define PI 3.1415926535897932385E0

main() {
    double imag, real, input[max+1];
    int i = 0, j, k;
    FILE *data;
    FILE *output;

    // read data from input.dat, save data in fourier.dat
    data = fopen("input.dat", "r");
    output = fopen("fourier.dat", "w");

    while ((fscanf(data, "%lf", &input[i]) != EOF) && (i<max)) i++; // loop for frequency
    for (j = 0; j<i; j++) { // clear variables
        real = imag = 0.0; // loop for sums
        for (k = 0; k<i; k++) {
            real+= input[k]*cos((2*PI*k*j)/i);
            imag+= input[k]*sin((2*PI*k*j)/i);
        }
        fprintf(output, "%d\t%f\t%f\n", j, real/i, imag/i );
    }
    printf("data stored in fourier.dat.\n");
    fclose(data);
    fclose(output);
}

```

**Listing E.18: gauss.c**

```

// gauss.c: Points and weights for Gaussian quadrature
// comment: this file has to reside in the same directory as integ.c
#include <math.h>
void gauss(int npts, int job, double a, double b,
           double x[], double w[]) {
    //      npts      number of points
    //      job= 0   rescaling uniformly between (a,b)
    //              1   for integral (0,b) with 50% pts inside (0, ab/(a+b))
    //              2   for integral (a,inf) with 50% inside (a,b+2a)
    //      x, w      output grid points and weights.

    int     m, i, j;
    double t, t1, pp, p1, p2, p3;
    double pi= 3.1415926535897932385E0, eps= 3.e-10;
                           // eps= accuracy to adjust
    m= (npts+1)/2;
    for (i = 1; i<= m; i++) {
        t = cos(pi*(i-0.25)/(npts+0.5));
        t1= 1;
        while((fabs(t-t1))>= eps) {
            p1= 1.0;
            p2= 0.0;
            for (j = 1; j<= npts; j++) {
                p3= p2;
                p2= p1;
                p1= ((2*j-1)*t*p2-(j-1)*p3)/j;
            }
            pp= npts*(t*p1-p2)/(t*t-1);
            t1= t;
            t = t1 - p1/pp;
        }
        x[i-1]= -t;
        x[npts-i]= t;
        w[i-1] = 2.0/((1-t*t)*pp*pp);
        w[npts-i]= w[i-1];
    }
    if (job == 0) {
        for (i = 0; i<npts ; i++) {
            x[i]= x[i]*(b-a)/2.0+(b+a)/2.0;
            w[i]= w[i]*(b-a)/2.0;
        }
    }
    if (job == 1) {
        for (i = 0; i<npts; i++) {
            x[i]= a*b*(1+x[i]) / (b+a-(b-a)*x[i]);
            w[i]= w[i]*2*a*b*b /((b+a-(b-a)*x[i])*(b+a-(b-a)*x[i]));
        }
    }
    if (job == 2) {
        for (i = 0; i<npts; i++) {
            x[i]= (b*x[i]+b+a+a) / (1-x[i]);
            w[i]= w[i]*2*(a+b) /((1-x[i])*(1-x[i]));
        }
    }
}

```

**Listing E.19: invfour.c**

```

// invfour.c: Inverse Discrete Fourier Transformation
/* comment: The program reads its input data from a file in the
 * same directory fourier.dat which must have the same format
 * frequency index real part imaginary part
 * as created with the program fourier.c
 * The output has the same format.
 * related programs: fourier.c */
#include <stdio.h>
#include <math.h>
#define max 10000
#define PI 3.1415926535897932385 E0

main() {
    double imag, real, input [2][max];
    int i = 0, j, k;

    FILE *data;
    FILE *output;
    data = fopen("fourier.dat", "r");           // read data from
    output = fopen("invers.dat", "w");           // save data in
    while (fscanf(data, "%d %lf %lf", &j, &input[0][i],
                  &input[1][i]) != EOF)
    { i++; }                                // input[0][x]: real, input[1][x]: imaginary
    for (j = 0; j<i; j++) {                   //loop for the frequency index
        real = imag = 0.0;                    // clear variables
        for (k = 0; k<i; k++) {              //loop for sum
            real+= input[0][k]*cos(2*PI*k*j/i)+input[1][k]*sin(2*PI*k*j/i);
            imag+= input[1][k]*cos(2*PI*k*j/i)-input[0][k]*sin(2*PI*k*j/i);
        }
        fprintf(output, "%i\t %f\t%f\n", j, real, imag);
    }
    printf("data saved in invers.dat\n");
    fclose(output);
    fclose(data);
}

```

**Listing E.20: int10d.c**

```
// int10d.c: Ten dimensional Monte Carlo integration

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// if you don't have drand48 uncomment the following two lines
#define drand48 1.0/RAND_MAX*rand
#define srand48 srand
#define max 65536                                // number of trials

main() {
    int i, j;
    double n = 1.0, x, y = 0;
    FILE *output;                               // save data in int_10d.dat
    output= fopen("int_10d.dat", "w");
    for (i = 1; i <= max; i++) {                // reset x
        x = 0;                                  // sum of 10 x values
        for (j = 1; j <= 10; j++) x+= drand48();
        y+= x*x;                                // save after 2, 4, 8,
        if (i%(int)(pow(2.0,n)) == 0) {
            n++;
            fprintf( output, "%i\t\t%f\n", i, y/i);
        }
    }
    printf("data saved in int_10d.dat\n");
    fclose(output);
}
```

**Listing E.21: integ.c**

```
// integ.c: Integration using trapezoid, Simpson and Gauss rules
/* comment: The derivation from theoretical result for each method
   is saved in x y1 y2 y3 format.
   Program needs gauss.c in the same directory. */
#include <stdio.h>
#include <math.h>
#include "gauss.c"                            // returns Legendre pts, weights
#define max_in 501                             // max number of intervals
#define vmin 0.0                               // ranges of integration
#define vmax 1.0
#define ME 2.7182818284590452354E0          // Euler's number

main() {
    int i;
    float result;
    float f(float x);
    float trapez (int no, float min, float max);    // trapezoid rule
    float simpson (int no, float min, float max);    // Simpson's rule
    float gaussint(int no, float min, float max);    // Gauss' rule
```

```

FILE *output;                                // save data in integ.dat
output= fopen("integ.dat","w");
// Simpson requires odd N

for (i = 3; i<= max_in; i+= 2) {
    result= trapez(i, vmin, vmax);
    fprintf(output, "%i\t%e\t", i, fabs(result-1+1/ME));
    result= simpson(i, vmin, vmax);
    fprintf(output, "%e\t", fabs(result-1+1/ME));
    result= gaussint(i, vmin, vmax);
    fprintf(output, "%e\n", fabs(result-1+1/ME));
}
printf("data stored in integ.dat\n");
fclose(output);
}                                              // end

float f (float x)                           // function to integrate
{ return (exp(-x)); }

float trapez (int no, float min, float max) {           // trapezoid rule
    int n;
    float interval, sum = 0., x;
    interval= ((max-min) / (no-1));
    for (n = 2; n<no; n++) {                      // sum the midpoints
        x = interval * (n-1);
        sum += f(x)*interval;
    }
    sum += 0.5 *(f(min) + f(max)) * interval;      // add the endpoints
    return (sum);
}

float simpson (int no, float min, float max) {          // Simpson's rule
    int n;
    float interval, sum = 0., x;
    interval= ((max-min) / (no-1));
    for (n = 2; n<no; n+= 2) {                    // loop for odd points
        x= interval * (n-1);
        sum += 4 * f(x);
    }
    for (n = 3; n<no; n+= 2) {                    // loop for even points
        x= interval * (n-1);
        sum += 2 * f(x);
    }
    sum += f(min) + f(max);                         // add first and last value
    sum *= interval/3.;                            // Gauss' rule
    return (sum);
}

float gaussint (int no, float min, float max) {
    int n;
    float quadra= 0.;
    double w[1000], x[1000];
    void gauss(int npts,int job,double a,double b,double x[],double w[]); // for points and weights

```

```

gauss (no, 0, min, max, x, w);           // Gauss Legendre points & wts
for (n = 0; n< no; n++) quadra += f(x[n])*w[n];    // Calc integral
return (quadra);
}

```

**Listing E.22: ising.c**

```

// Ising.c: ising model of magnetic dipole string
// Plot without connecting datapoints with lines

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// if you don't have drand48 uncomment the following two lines
#define drand48 1.0/RAND_MAX*rand
#define srand48 srand

#define max 100                                // number of objects
#define kt 100.0                               // temperature
#define J -1                                    // exchange energy
#define seed 68111                             // seed for srand48

main() {
    int i, j, element, array[max];
    double olden, newen;
    double energy(int array[]);                // energy of system

    FILE *output1, *output2;                   // save spin ups and
    output1= fopen("spin-up.dat", "w");        // downs in two files
    output2= fopen("spin-down.dat", "w");
    srand48(seed);                          // seed generator
                                            // uniform start

    for (i = 0; i<max; i++) array[i] = 1;      // time loop

    for (i = 0; i<= 500; i++) {
        olden = energy(array);                // initial energy
        element= drand48()*max;               // pick one element
        array[element] *= -1;                 // change spin
        newen = energy(array);                // calculate new energy
                                                // reject/accept change
        if ( (newen>olden) && (exp((-newen + olden)/kt) <= drand48()) )
            { array[element]= array[element]*(-1); }
                                                // save "map" of spins
        for (j = 0; j<max; j++) {
            if (array[j] == 1) fprintf(output1, "%d %d\n", i, j);
            if (array[j] == -1) fprintf(output2, "%d %d\n", i, j);
        }
    }
    fclose (output1);
    fclose (output2);
    printf("data saved in spin-up.dat, spin-down.dat\n");
}                                         // end of main program

double energy (int array[]) {              // function returns energy

```

```

int i;
double sum= 0.;
for (i = 0; i<(max-1); i++)           // loop through elements
    sum += array[i]*array[i+1];
return (J*sum);
}

```

### Listing E.23: lineq.c

```

// lineq.c: Solve matrix equation Ax = b using LAPACK routine sgesv
// LAPACK has to be installed on your system

#include<stdio.h>
#define size 3                         // dimension of matrix

main()  {

    int i, j , c1, c2, pivot[size], ok;
    float A[ size ][ size ], b[size], AT[ size *size ];

    A[0][0]= 3.1; A[0][1]= 1.3; A[0][2]= -5.7;                      // matrix A
    A[1][0]= 1.0; A[1][1]= -6.9; A[1][2]= 5.8;
    A[2][0]= 3.4; A[2][1]= 7.2; A[2][2]= -8.8;
    b[0]= -1.3;                                         // vector b
    b[1]= -0.1;
    b[2]= 1.8;

    // Transform matrix for Fortran
    for (i = 0; i<size; i++)
        {for (j= 0 ; j < size ; j += 1) AT[j + size*i]= A[j][i];}
    c1 = size;                                // define variable so we can pass pointer
    c2 = 1;
    sgesv(&c1 , &c2 , AT, &c1 , pivot , b , &c1 , &ok);

    // parameters in the order as they appear in the function call:
    // order of matrix A, number of right hand sides (b), matrix A,
    // leading dimension of A, array records pivoting,
    // result vector b on entry, x on exit, leading dimension of b
    // return value = 0 for success

    if (!ok) for (j = 0; j<size; j++) printf("%e\n", b[j]); // print x
    else printf("An error occurred\n");
}

```

**Listing E.24: laplace.c**

```
// laplace.c: Solution of Laplace equation with finite differences
// Output data is saved in 3D grid format used by gnuplot
#include <stdio.h>
#define max 40                                // number of grid points

main() {
    double x, p[max][max];
    int i, j, iter, y;

    FILE *output;                            // save data in laplace.dat
    output= fopen("laplace.dat","w");
    for (i = 0; i<max; i++) for (j = 0; j<max; j++) p[i][j]= 0; // clear
    for (i = 0; i<max; i++) p[i][0]= 100.0;           // p[i][0]= 100 V
    for (iter = 0; iter<1000; iter++) {             // iterations
        for (i = 1; i<(max-1); i++) {               // x-direction
            for (j = 1; j<(max-1); j++) {           // y-direction
                p[i][j]= 0.25*(p[i+1][j]+p[i-1][j]+p[i][j+1]+p[i][j-1]); }
            }
        for (i = 0; i<max ; i++) {                // write data gnuplot 3D format
            for (j = 0; j<max; j++) fprintf(output, "%f\n",p[i][j]);
            fprintf(output, "\n");                  // empty line for gnuplot
        }
        printf("data stored in laplace.dat\n");
        fclose(output);
    }
}
```

**Listing E.25: LaplaceSOR.c**

```
// LaplaceSOR.c:Solve Laplace equation with finite difference method
// Output data saved in 3D grid format of gnuplot

#include<stdio.h>
#include<math.h>
#include<stdlib.h>

main() {
    int max= 40;                                // number of grid points
    double tol,omega,r;
    double p[40][40];
    int i, j, iter;

    FILE *output1;
    output1 = fopen("laplaceR.dat","w"); // save data in laplaceR.dat
    omega = 1.4;                                // The SQR parameter
    for (i = 0; i<max; i++) for (j = 0; j<max; j++) p[i][j]= 0; // clear
    for (i = 0; i<max; i++) p[i][0]= +100.0;      // p[i][0]= 100 V
    tol= 1.0;                                    // tolerance
    iter= 0;                                     // iterations
    while ( (tol > 0.000001) && (iter <= 140) ){
        tol= 0.0;
        for (i = 1; i<(max-1); i++) {           // x-direction
            for (j = 1; j<(max-1); j++) {           // y-direction
                p[i][j]= (p[i+1][j]+p[i-1][j]+p[i][j+1]+p[i][j-1])/4;
            }
        }
        for (i = 0; i<max ; i++) {                // write data gnuplot 3D format
            for (j = 0; j<max; j++) fprintf(output1, "%f\n",p[i][j]);
            fprintf(output1, "\n");                  // empty line for gnuplot
        }
        iter++;
    }
}
```

```

    for (j = 1; j <(max-1); j++) {
        r= omega * ( p[i][j+1] + p[i][j-1] + p[i+1][j] +
                      p[i-1][j] - 4.0 * p[i][j] ) / 4.0;
        p[i][j] += r;
        if ( fabs(r) > tol ) tol= fabs(r);
    }
    iter++;
}
for (i = 0; i<max ; i++) {           // write data gnuplot 3D format
    for (j = 0; j<max; j++) fprintf(output1,"%f\n",p[i][j]);
    fprintf(output1,"\\n");                // empty line for gnuplot
}
printf("data stored in laplaceSOR.dat");
}

```

**Listing E.26: limit.c**

```

// limit.c: Determine machine precision , smallest e for 1 + e != 1

#include <stdio.h>
#define N 60

main() {
    double eps = 1.0, one;                         // starting values
    int i;

    for (i = 0; i<N; i++) {
        eps /= 2.;                                // divide by two
        one = 1.0+eps;
        printf("%.18f \t %.16e \n",one, eps);
    }
}

```

**Listing E.27: over.c**

```

// over.c: Determine overflow and underflow limits

#include <stdio.h>
#define N 1024           // might not be big enough to cause
                        // over and underflow

main() {
    double under = 1., over = 1.;                  // starting values
    int i;

    for (i = 0; i<1024; i++) {
        under /= 2.;                            // divide by two
        over *= 2.;                            // multiply by two
        printf("%d. under: %e over: %e \n",i+1,under,over);
    }
}

```

**Listing E.28: pond.c**

```
// pond.c: *Monte Carlo integration to determine pi (stone throwing

#include <stdio.h>
#include <stdlib.h>

// if you don't have drand48 uncomment the following two lines
// #define drand48 1.0/RAND_MAX*rand
// #define srand48 srand

#define max 1000                                // number of stones to be thrown
#define seed 68111                               // seed for number generator

main()  {

    int i, pi = 0;
    double x, y, area;
    FILE *output;                                // save data in pond.dat
    output = fopen("pond.dat","w");
    srand48(seed);                             // seed the number generator
    for (i = 1; i<= max; i++) {
        x = drand48()*2-1;                      // creates floats between
        y = drand48()*2-1;                      // 1 and -1
        if ((x*x + y*y)<1) pi++;                // stone hit the pond
        area = 4*(double)pi/i;                   // calculate area
        fprintf(output, "%i\t%f\n", i, area);
    }
    printf("data stored in pond.dat\n");
    fclose(output);
}
```

**Listing E.29: qmc.c**

```
// qmc.c: Feynman path integral for ground state wave function

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// if you don't have drand48 uncomment the following two lines
// #define drand48 1.0/RAND_MAX*rand
// #define srand48 srand

#define max 250000                                // number of trials
#define seed 68111                               // seed for number generator

main()  {

    double change, newE, oldE, path [101];
    int i, j, element, prop [101];
    double energy( double array []);             // finds energy of path
    FILE *output;                                // save data in qmc.dat
    output = fopen("qmc.dat","w");
    srand48(seed);                            // seed number generator
    for (j = 0; j<= 100; j++) path[j] = 0.0;    // initial path
    for (j = 0; j<= 100; j++) prop[j] = 0;       // initial probability
    oldE= energy(path);                         // find energy of path
```

```

for (i = 0; i<max; i++) {                                // pick one random element
    element = drand48() *101;
    change = (int)((drand48() - 0.5)*20) / 10.0;      // change -0.9..0.9
                                                        // change path, find new E
    path[element] += change;
    newE = energy(path);                                // Metropolis Algorithm
    if ((newE>oldE) && (exp(-newE+oldE) <= drand48()))
        { path[element] -= change; }                      // reject
                                                        // add up probabilities
    for (j = 0; j <= 100; j++) {
        element = path[j]*10+50;
        prop[element]++;
    }
    oldE= newE;
}
for (i = 0; i <= 100; i++)
    { fprintf(output, "%d\t%f\n", i-50, (double) prop[i]/max);}
printf("data stored in qmc.dat\n");
fclose(output);
}                                                       // end of main

double energy (double array[]) {                         // energy of path configuration
    int i;
    double sum = 0.;
    for (i = 0; i < 100; i++)
        { sum += pow(array[i+1]-array[i], 2.0) + array[i]*array[i]; }
    return (sum);
}

```

Listing E.30: random.c

```

// random.c: A simple random number generator, not for serious work
#include <stdio.h>
#define max 1000                                // number of numbers generated
#define seed 11                                  // seed for number generator

main () {
    int i, old, newx, newy;

    FILE *output;                               // save data in badrand.dat
    output= fopen("badrand.dat","w");
    old= seed;                                 // the seed
    for (i = 0; i < max; i++) {                // generating #max numbers
        newx= (57*old+1) % 256;                 // x-coordinate
        newy= (57*newx+1) % 256;                 // y-coordinate
        fprintf (output, "%i\t%i\n", newx, newy);
        old = newy;
    }
    printf("data stored in badrand.dat.\n");
    fclose(output);
}

```

**Listing E.31: RK4.c**

```
// rk4.c: 4th order Runge-Kutta solution for harmonic oscillator

#include <stdio.h>
#define N 2                                // number of equations
#define dist 0.1                            // stepsize
#define MIN 0.0                             // minimum x
#define MAX 10.0                            // maximum x

main() {
    double x, y[N];
    int j;
    void runge4(double x, double y[], double step);           // header
    double f(double x, double y[], int i);

    FILE *output;                         // save data in rk4.dat
    output= fopen("rk4.dat","w");
    y[0]= 1.0;                           // initial position
    y[1]= 0.0;                           // initial velocity
    fprintf(output, "%f\t%f\n", x, y[0]);

    for (x= MIN; x <= MAX ; x += dist) {
        runge4(x, y, dist);
        fprintf(output, "%f\t%f\n", x, y[0]);           // position vs. time
    }
    printf("data stored in rk4.dat\n");
    fclose(output);
}                                         // -end of main program

void runge4(double x, double y[], double step) { // rk subroutine
    double f(double x, double y[], int i);
    double h = step/2.0,                      // the midpoint
    t1[N], t2[N], t3[N], k1[N], k2[N], k3[N],k4[N]; // for Runge-Kutta
    int i;
    for (i = 0; i < N; i++) t1[i]= y[i]+0.5*(k1[i] = step*f(x, y, i));
    for (i = 0; i < N; i++) t2[i]= y[i]+0.5*(k2[i] = step*f(x+h, t1, i));
    for (i = 0; i < N; i++) t3[i]= y[i]+(k3[i] = step*f(x+h, t2, i));
    for (i = 0; i < N; i++) k4[i]= step * f(x + step, t3, i);
    for (i = 0; i < N; i++) y[i] += (k1[i]+2*k2[i]+2*k3[i]+k4[i]) / 6.0;
}

double f(double x, double y[], int i) {           // RHS function
    if (i == 0) return(y[1]);                     // RHS of first equation
    if (i == 1) return(-y[0]);                    // RHS of second equation
}
```

**Listing E.32: rk45.c**

```
/* rk45.c Ordinary Differential Equations Solver (ODES).
   Solve the differential equation using Runge-Kutta-Fehlberg Method
   with variable step size. */
#include<stdio.h>
#include<stdlib.h>
```

```

#include<math.h>

main()  {

FILE *out1;           // open file rk45.dat for output data
out1 = fopen("rk45.dat","w");

double h,t,s,s1,hmin,hmax;
double y[2], fReturn[2], ydumb[2];
double k1[2], k2[2], k3[2];
double k4[2], k5[2], k6[2];
double err[2];
double Tol = 1.0E-8;           //error control tolerance
double a = 0.0;                //endpoints
double b = 10.0;
int i,j,n = 20;
void f(double t, double y[], double fReturn[]);

y[0] = 1.0; y[1]= 0.0;
h = (b-a)/n;                  //tentative number of steps
hmin = h/64;
hmax = h*64;                  //minimum and maximum step size
t = a;
j = 0;
fprintf(out1,"%f\t%f\n",t,y[0],y[1]); //output answer to file

while (t<b) {
    if ( (t + h) > b ) h= b - t;           // the last step
    f(t, y, fReturn); // evaluate both RHS's and return in fReturn
    k1[0] = h*fReturn[0]; //compute the function values
    k1[1] = h*fReturn[1];

    for (i = 0;i<= 1;i++) ydumb[i]= y[i] + k1[i]/4;
    f(t + h/4, ydumb, fReturn);
    k2[0] = h*fReturn[0];
    k2[1] = h*fReturn[1];

    for (i = 0;i<= 1;i++) ydumb[i]= y[i] + 3*k1[i]/32 + 9*k2[i]/32;
    f(t + 3*h/8, ydumb, fReturn);
    k3[0]= h*fReturn[0];
    k3[1]= h*fReturn[1];

    for (i = 0;i<= 1;i++) ydumb[i]= y[i] + 1932*k1[i]/2197
        -7200*k2[i]/2197. +7296*k3[i]/2197;
    f(t+ 12*h/13, ydumb, fReturn);
    k4[0]= h*fReturn[0];
    k4[1]= h*fReturn[1];

    for (i = 0;i<= 1;i++) ydumb[i]= y[i] +439*k1[i]/216 -8*k2[i]
        +3680*k3[i]/513 -845*k4[i]/4104;
    f(t+h, ydumb, fReturn);
    k5[0]= h*fReturn[0];
    k5[1]= h*fReturn[1];

    for (i = 0;i<= 1;i++) ydumb[i]= y[i] -8*k1[i]/27 +2*k2[i]
        -3544*k3[i]/2565 +1859*k4[i]/4104 -11*k5[i]/40;
    f(t+ h/2, ydumb, fReturn);
}

```

```

k6[0]= h*fReturn[0];
k6[1]= h*fReturn[1];

for ( i = 0;i<= 1;i++) err[i] = abs( k1[i]/360 - 128*k3[i]/4275
- 2197*k4[i]/75240 + k5[i]/50.0 + 2*k6[i]/55 );
if ((err[0]<Tol)||((err[1]<Tol)||((h<= 2*hmin))) { // accept
    for ( i = 0;i<= 1;i++) y[i]= y[i] + 25*k1[i]/216.
+ 1408*k3[i]/2565.
+ 2197*k4[i]/4104. - k5[i]/5. ;
    t= t + h;
    j++;
}
if (( err[0] == 0)||((err[1] == 0)) s = 0; //trap division by 0
else s = 0.84*pow(Tol*h/err[0],0.25); //step size scalar
if ( (s < 0.75) && (h > 2*hmin) ) h /= 2.; //reduce step
else if ( (s > 1.5) && (2* h < hmax) ) h *= 2.; //increase step
fprintf(out1,"%f\t%f\t%f\n",t,y[0],y[1]); //output answer to file
}
}

void f(double t, double y[], double fReturn[]) { // RHS function
fReturn[0] = y[1]; // RHS of first equation
fReturn[1] = -6.0*pow(y[0],5.0); // RHS of second equation
return;
}

```

### **Listing E.33: sierpin.c**

```

// sierpin.c: Creates Sierpinsky gasket fractal
// Plot data without connecting datapoints with lines.

#include <stdio.h>
#include <stdlib.h>

// if you don't have drand48 uncomment the following two lines
#define drand48 1.0/RAND_MAX*rand
#define srand48 srand

#define max 30000          // number of iterations
#define seed 68111          // seed for number generator
#define a1 20.0             // vertex 1
#define b1 20.0             // vertex 2
#define a2 320.0            // vertex 2
#define b2 20.0             // vertex 3
#define a3 170.0            // vertex 3
#define b3 280.0

main() {
    int i;
    double x, y, r;

    FILE *output;           // save data in sierpin.dat
    output= fopen("sierpin.dat","w");
    x = 180.;               // starting point

```

```

y = 150.;
srand48(seed);                                // seed number generator
for (i = 1 ; i <= max ; i++) {                // draw the gasket
    r = drand48();
    if (r <= 0.3333) {
        x = 0.5*(x + a1);
        y = 0.5*(y + b1);
    }
    else if (r > 0.3333 && r <= 0.6666) {
        x = 0.5*(x + a2);
        y = 0.5*(y + b2);
    }
    else {
        x = 0.5*(x + a3);
        y = 0.5*(y + b3);
    }
    fprintf(output, "%f %f\n", x, y);
}
printf("data stored in sierpin.dat\n");
fclose(output);
}

```

Listing E.34: walk.c

```

// area.c: Area of a circle , sample I/O

#include <stdio.h>
#define PI 3.1415926535897932385E0
main() {

    double radius, area;                      // declare variables

    printf("Enter the radius of a circle \n");   // ask for radius
    scanf("%lf", &radius);                     // read in radius
    area = radius * radius * PI;               // area formula
    printf("radius=%f, area=%f\n", radius, area); // print results
}

```

**Listing E.35: soliton.c**

```

// soliton.c: Solves KdV Equation using finite difference method
// comment: Output data is saved in 3D grid format used by gnuplot
#include <stdio.h>
#include <math.h>
#define ds 0.4                                // delta x
#define dt 0.1                                 // delta t
#define max 2000                               // time steps
#define mu 0.1                                 // mu from KdV equation
#define eps 0.2                                // epsilon from KdV eq

main() {
    int i, j, k;
    double a1,a2,a3, fac, time, u[131][3];
    FILE *output;                            // save data in soliton.dat
    output= fopen("soliton.dat","w");
    for (i = 0; i<131; i++) u[i][0]= 0.5*(1.-tanh(0.2*ds*i-5.));
    u[0][1] = 1.;                           // end points
    u[0][2] = 1.;
    u[130][1] = 0.;
    u[130][2] = 0.;
    fac = mu*dt/(ds*ds*ds);
    time= dt;
    for (i = 1; i<130; i++) {           // first time step
        a1= eps*dt*(u[i+1][0] + u[i][0] + u[i-1][0]) / (ds*6.0);
        if ((i>1)&&(i<129)) a2=u[i+2][0]+2.0*u[i-1][0]
                                     -2.0*u[i+1][0]-u[i-2][0];
        else a2= u[i-1][0] - u[i+1][0];
        a3= u[i+1][0]-u[i-1][0];
        u[i][1]= u[i][0] - a1*a3 - fac*a2/3.;
    }
    for (j = 1; j<max; j++) {           // all other time steps
        time+= dt;
        for (i = 1; i<130; i++) {
            a1= eps*dt*(u[i+1][1] + u[i][1] + u[i-1][1]) / (3.0*ds);
            if ((i>1) && (i<129)) a2= u[i+2][1] + 2.0*u[i-1][1]
                                         - 2.0*u[i+1][1] - u[i-2][1];
            else a2= u[i-1][1] - u[i+1][1];
            a3= u[i+1][1] - u[i-1][1];
            u[i][2]= u[i][0] - a1*a3 - 2.*fac*a2/3.;
        }
        for (k = 0; k<131; k++) {         // move one step ahead
            u[k][0]= u[k][1];
            u[k][1]= u[k][2];
        }
        if ((j%200) == 0) {             // plot every 200th step
            for (k = 0; k<131; k+= 2) fprintf(output, "%f\n", u[k][2]);
            fprintf(output, "\n");        // empty line for gnuplot
        }
    }
    printf("data stored in soliton.dat\n");
    fclose(output);
}

```

**Listing E.36: tree.c**

```

// tree.c: Creates a fractal pattern that looks like a tree
// Plot data without connecting datapoints with lines

#include <stdio.h>
#include <stdlib.h>

// if you don't have drand48 uncomment the following two lines
#define drand48 1.0/RAND_MAX*rand
#define srand48 srand
#define max 30000           // number of iterations
#define seed 68111          // seed for number generator

main() {
    int i;
    double x,y,r,xn,yn;

    FILE *output;
    output = fopen("tree.dat","w");
    srand48(seed);           // seed number generator
    x = 0.5;                 // initial position
    y = 0.0;

    for (i = 1 ; i <= max; i++) {
        r = drand48();
        if (r<= 0.1) {
            xn = 0.05*x;
            yn = 0.6*y;
        }
        else if ((r>0.1) && (r<0.2)) {
            xn = 0.05*x;
            yn = -0.5*y+1.0;
        }
        else if ((r>0.2) && (r<0.4)) {
            xn = 0.46*x-0.32*y;
            yn = 0.39*x+0.38*y+0.6;
        }
        else if ((r>0.4) && (r<0.6)) {
            xn = 0.47*x-0.15*y;
            yn = 0.17*x+0.42*y+1.1;
        }
        else if ((r>0.6) && (r<0.8)) {
            xn = 0.43*x+0.28*y;
            yn = -0.25*x+0.45*y+1.0;
        }
        else {
            xn = 0.42*x+0.26*y;
            yn = -0.35*x+0.31*y+0.7;
        }
        fprintf(output, "%f %f\n", xn, yn);
        x = xn;
        y = yn;
    }
    printf("data stored in tree.dat\n");
    fclose(output);
}

```

**Listing E.37: SplineAppl.c**

```

/* SplineAppl.c Cubic Spline fit to data, based on Press et al
input array x[n], y[n] = tabulation function y(x), x0 < x1,
output yout for given xout yp1 and ypn: 1st derivs at endpoints,
evaluated internally, y2[n]: array of 2nd derivatives
(setting yp1 or ypn >0.99e30 produces natural spline) */

#include<stdio.h>
#include<math.h>
#define n 9
#define np 15
#define Nfit 3000 // enter the desired number of points to fit

main() {
    FILE *output1, *output2;
    output1 = fopen("Spline.dat","w"); // save data in Spline.dat
    output2 = fopen("Input.dat","w");
    // input data, enter your own data here!
    double x[] = {0., 0.12, 0.25, 0.37, 0.5, 0.62, 0.75, 0.87, 0.99};
    double y[] = {108.6, 16., 845., 883.5, 52.8, 19.9, 10.8, 88.25, 84.7};
    double xout, yout;
    double y2[9];
    int i, klo, khi, k;
    double h,b,a,p,qn,sig,un,yp1,ypn, u[n];
    // input
    for (i = 0;i<n;i++) fprintf (output2,"%f\t%f\n",x[i],y[i]);
    // 1st deriv at initial point
    yp1 = (y[1]-y[0])/(x[1]-x[0])
        - (y[2]-y[1])/(x[2]-x[1]) + (y[2]-y[0])/(x[2]-x[0]);
    // 1st deriv at end point
    ypn = (y[n-1]-y[n-2])/(x[n-1]-x[n-2]) - (y[n-2]-y[n-3])/(
        (x[n-2]-x[n-3]) + (y[n-1]-y[n-3])/(x[n-1]-x[n-3]));
    // natural spline test
    if (yp1 > 0.99e30) y2[0]= u[0]= 0.0;
    else {
        y2[0] = (-0.5);
        u[0] = (3.0/(x[1]-x[0]))*((y[1]-y[0])/(x[1]-x[0])-yp1);
    }
    // decomposition loop;
    for (i = 1; i<= n-2; i++) {
        sig = (x[i]-x[i-1])/(x[i+1]-x[i-1]);
        p = sig*y2[i-1]+2.0;
        y2[i] = (sig-1.0)/p;
        u[i] = (y[i+1]-y[i])/(x[i+1]-x[i])-(y[i]-y[i-1])/(x[i]-x[i-1]);
        u[i] = (6.0*u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;
    }
    // test for natural
    if (ypn > 0.99e30) qn = un = 0.0;
    else {
        qn = 0.5;
        un = ((3.0)/(x[n-1]-x[n-2]))
            *(ypn-(y[n-1]-y[n-2])/(x[n-1]-x[n-2]));
    }
    y2[n-1] = (un-qn*u[n-2])/(qn*y2[n-2]+1.0);
    for (k= n-2;k>= 0;k--) { y2[k] = y2[k]*y2[k+1]+u[k]; }
}

```

```

// splint (initialization) ends, Begin *spline* fit
for ( i = 1; i<= Nfit; i++ ) { // loop over xout values
    xout = x[0] + (x[n-1]-x[0])*(i-1)/(Nfit);
    klo = 0; // Bisection algor to find place in table
    khi = n-1; // klo , khi bracket xout value
    while (khi-klo > 1) {
        k = (khi+klo) >> 1;
        if (x[k] > xout) khi = k;
        else klo= k;
    }
    h = x[khi]-x[klo];
    if (x[k] > xout) khi= k;
    else klo= k;
    h = x[khi]-x[klo];
    a = (x[khi]-xout)/h;
    b = (xout-x[klo])/h;
    yout = (a*y[klo]+b*y[khi]+((a*a*a-a)*y2[klo]
        +(b*b*b-b)*y2[khi])*(h*h)/6.0);
    fprintf (output1,"%f\t%f\n",xout,yout); // gnuplot 3D format
}
fclose(output1);
fclose(output2);
printf("data stored in Spline.dat");
}

```

Listing E.38: twodsol.c

```
// twodsol.c: Solves SGE for 2D soliton

#include<stdio.h>
#include<math.h>
#define D 201

main() {

    double u[D][D][3];
    int nint;
    void initial(double u[][D][3]);
    void solution(double u[][D][3], int nint);

        // Input +int proportional to viewing time
    printf(" Enter a positive integer from 1(initial time)\n");
    printf("to 1800 to get wave packet position at that time:\n");
    scanf("%d", &nint);
    initial(u);                                // Initializes constant values wave
    solution(u,nint);                          // Solve equation
}

void initial(double u[][D][3]) {                // Initialize function
    double dx, dy, dt, xx, yy, dts, time, tmp;
    int i, j;
    dx = 14./200.; dy = dx;
    dt = dx/sqrt(2.); dts = (dt/dx)*(dt/dx);
    yy = -7.; time = 0.;
    for (i = 0;i<= D-1;i++) {
        xx = -7.;
        for (j = 0;j<= D-1;j++) {
            tmp = 3.-sqrt(xx*xx+yy*yy);
            u[i][j][0] = 4.*atan(tmp);
            xx = xx+dx;
        }
        yy = yy+dy;
    }
}

void solution(double u[][D][3], int nint) {           // Solution
    double dx, dy, dt, time, a2, zz, dts, a1, tmp;
    int l, m, mm, k, j, i;
    FILE *pf;
    pf = fopen("2dsol.dat", "w");
    dx = 14./200.;
    dy = dx;
    dt = dx/sqrt(2.);
    time = 0.;
    time = time+dt;
    dts = (dt/dx)*(dt/dx);
    tmp = 0.;

    for (m = 1;m<= D-2;m++) {
        for (l = 1;l<= D-2;l++) {
            a2 = u[m+1][l][0]+u[m-1][l][0]+u[m][l+1][0]+u[m][l-1][0];
            tmp = .25*a2;
            u[m][l][1] = .5*(dts*a2-dt*dt*sin(tmp));
        }
    }
}
```

```

    }
}

for (mm = 1;mm<= D-2mm++) { // borders of 2nd iteration
    u[mm][0][1] = u[mm][1][1];
    u[mm][D-1][1] = u[mm][D-2][1];
    u[0][mm][1] = u[1][mm][1];
    u[D-1][mm][1] = u[D-2][mm][1];
}
u[0][0][1] = u[1][0][1]; // still undefined terms
u[D-1][0][1] = u[D-2][0][1];
u[0][D-1][1] = u[1][D-1][1];
u[D-1][D-1][1] = u[D-2][D-1][1];
tmp = 0.; // 3rd etc iterations
for (k = 0;k<= nint;k++) {
    for (m = 1;m<= D-2m++) {
        for (l = 1;l<= D-2;l++) {
            a1 = u[m+1][1][1]+u[m-1][1][1]+u[m][1+1][1]+u[m][1-1][1];
            tmp = .25*a1;
            u[m][1][2] = -u[m][1][0]+dts*a1-dt*dt*sin (tmp);
            u[m][0][2] = u[m][1][2];
            u[m][D-1][2] = u[m][D-2][2];
        }
    }
    for (mm = 1;mm<= D-2mm++) {
        u[mm][0][2] = u[mm][1][2];
        u[mm][D-1][2] = u[mm][D-2][2];
        u[0][mm][2] = u[1][mm][2];
        u[D-1][mm][2] = u[D-2][mm][2];
    }
    u[0][0][2] = u[1][0][2];
    u[D-1][0][2] = u[D-2][0][2];
    u[0][D-1][2] = u[1][D-1][2];
    u[D-1][D-1][2] = u[D-2][D-1][2];
    for (l = 0;l<= D-1;l++) { // recycle
        for (m = 0;m<= D-1m++) {
            u[1][m][0] = u[1][m][1];
            u[1][m][1] = u[1][m][2];
        }
    }
    if (k == nint) {
        for (i = 0;i<= D-1;i = i+4) {
            for (j = 0;j<= D-1;j = j+4)
                fprintf(pf,"%e\n",sin(u[i][j][2]/2.));
                fprintf(pf,"\\n");
        }
    }
    time = time+dt;
}
fclose(pf);
}

```

## References

- 1** UNDERGRADUATE COMPUTATIONAL ENGINEERING AND SCIENCE, <http://www.krellinst.org/UCES/>.
- 2** LANDAU, R. H. (2005), *A First Course in Scientific Computing*, Princeton University Press, Princeton.
- 3** LANDAU, R. H., LANDAU, M. J. PÁEZ, AND C. C. BORDEIANU, (2007), *A Survey of Computational Physics*, Princeton University Press, Princeton.
- 4** FOSDICK L. D., E. R. JESSUP, C. J. C. SCHAUBLE, AND G. DOMIK (1996), *An Introduction to High Performance Scientific Computing*, MIT Press, Cambridge.
- 5** PINSON, L. J. AND R. S. WIENER (1999), *Objective-C Object-Oriented Programming Techniques*, Addison-Wesley, Reading, MA.
- 6** SMITH, D. N. (1991), *Concepts of Object-Oriented Programming*, McGraw-Hill, New York.
- 7** ABRAMOWITZ, M. AND I. A. STEGUN (1972), *Handbook of Mathematical Functions*, 10th ed. U.S. Government Printing Office, Washington.
- 8** GOTTFRIED, K. (1966), *Quantum Mechanics*, Benjamin, New York.
- 9** PRESS, W. H., B. P. FLANNERY, S. A. TEUKOLSKY, AND W. T. VETTERLING (1994), *Numerical Recipes*, Cambridge University Press, Cambridge, UK.
- 10** PRESS, W. H., B. P. FLANNERY, S. A. TEUKOLSKY, AND W. T. VETTERLING (2000), *Numerical Recipes in C++*, 2nd ed., Cambridge University Press, Cambridge, UK.
- 11** JAMA, A Java matrix package; Java Numerics, <http://math.nist.gov/javanumerics/>
- 12** PENNA, T. J. P. (1994), *Comput. Phys.* **9**, 341.
- 13** BEVINGTON, P. R. AND D. K. ROBINSON (2002), *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed., McGraw-Hill, New York.
- 14** THOMPSON, W. J. (1992), *Computing for Scientists and Engineers*, Wiley, New York.
- 15** STETZ, A., J. CARROLL, N. CHIRAPATPI-MOL, M. DIXIT, G. IGO, M. NASSER, D. ORTENDAHL, AND V. PEREZ-MENDEZ (1973), "Determination of the Axial Vector Form Factor in the Radiative Decay of the Pion", LBL 1707, invited paper at the Symposium of the Division of Nuclear Physics, Washington, DC, April.
- 16** MATHEWS, J. AND R. L. WALKER (1965), *Mathematical Methods of Physics*, Benjamin, Reading, MA.
- 17** GOULD, H., J. TOBOCHNIK, AND W. CHRISTIAN (2006), *An Introduction to Computer Simulation Methods*, 3rd ed., Addison-Wesley, Reading, MA.
- 18** PLISCHKE, M. AND B. BERGERSEN (1994), *Equilibrium Statistical Physics*, 2nd ed., World Scientific, Singapore.
- 19** HUNAG, K. (1987), *Statistical Mechanics*, Wiley, New York.
- 20** YANG, C. N. (1952), *The Spontaneous Magnetization of a Two-Dimensional Ising Model* *Phys. Rev.* **85**, 809.
- 21** METROPOLIS, M., A. W. ROSENBLUTH, M. N. ROSENBLUTH, A. H. TELLER, AND E. TELLER (1953), *J. Chem. Phys.* **21**, 1087.
- 22** JOSÉ, J. V AND E. J. SALATAN, (1988) *Classical Dynamics*, Cambridge University Press, Cambridge, UK.

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5

- 23** SCHECK, F. (1994), *Mechanics, from Newton's Laws to Deterministic Chaos*, 2nd ed., Springer, New York.
- 24** TABOR, M. (1989), *Chaos and Integrability in Nonlinear Dynamics*, Wiley, New York.
- 25** MATHEWS, J., (2002), *Numerical Methods for Mathematics, Science and Engineering*, Prentice-Hall, Upper Saddle River.
- 26** MARION, J. B. AND S. T. THORNTON (2003), *Classical Dynamics of Particles and Systems*, 5th ed., Harcourt Brace Jovanovich, Orlando, FL.
- 27** WARBURTON, R. D. H. AND J. WANG, (2004), *Analysis of asymptotic projectile motion with air resistance using the Lambert W function*, Am. J. Phys. **72**, 1404.
- 28** KOONIN, S. E. (1986), *Computational Physics*, Benjamin, Menlo Park, CA.
- 29** SCHMID, E. W., G. SPITZ, AND W. LÖSCH (2000), *Theoretical Physics on the Personal Computer*, 2nd ed., Springer, Berlin.
- 30** LANDAU, R. H. (1996), *Quantum Mechanics II, A Second Course in Quantum Theory*, 2nd ed., Wiley, New York.
- 31** LANDAU, L. D. AND E. M. LIFSHITZ (1976), *Mechanics*, 3rd ed., Butterworth-Heinemann, Oxford, UK.
- 32** BRIGGS, W. L. AND V. E. HENSON (1995), *The DFT, An Owner's Manual*, SIAM, Philadelphia.
- 33** PEDERSEN, N. F. AND A. DAVIDSON (1981), Appl. Phys. Lett. **39**, 830.
- 34** PHYSICS TODAY, Special issue on chaos, December 1988.
- 35** RASBAND, S. N. (1990), *Chaotic Dynamics of Nonlinear Systems*, Wiley, New York.
- 36** FEIGENBAUM, M. J. (1979), J. Stat. Physics **21**, 669.
- 37** PHATAK, S. C. AND S. S. RAO (1995), *Logistic map: A possible random-number generator*, Phys. Rev. E **51**, 3670.
- 38** DE JONG, M. L. (1992), *Chaos and the simple pendulum*, The Physics Teacher **30**, 115.
- 39** ABBARANE, H. D. I., M. I. RABI-NOVICH, AND M. M. SUSHCHIK (1993), *Introduction to Nonlinear Dynamics for Physicists*, World Scientific, Singapore.
- 40** MOON, F. C. AND G.-X. LI (1985), Phys. Rev. Lett. **55**, 1439.
- 41** ARMIN, B., AND H. SHLOMO, EDS. (1991), *Fractals and Disordered Systems*, Springer, Berlin.
- 42** EUGENE, S. H. AND M. PAUL (1988), Nature **335**, 405.
- 43** SANDER, E., L. M. SANDER, AND R. M. ZIFF (1994), Comput. Phys. **8**, 420.
- 44** MANDELBROT, B. (1982), *The Fractal Geometry of Nature*, 29, Freeman, San Francisco.
- 45** VOLD, M. J. (1959), J. Collod. Sci. **14**, 168.
- 46** FEREYDOON, F. AND V. TAMÁS (1990), Comput. Phys., 44; (1985), J. Phys. A **18**, L75.
- 47** MANDELBROT, B. (1967), *How long is the coast of Britain?*, Science, **156**, 638.
- 48** RICHARDSON, L. F., (1961) *Problem of contiguity: an appendix of statistics of deadly quarrels*, General Systems Yearbook, **6**, 139.
- 49** TAIT, R. N., T. SMY, AND M. J. BRETT (1990), Thin Solid Films **187**, 375.
- 50** WITTEN, T. A. AND L. M. SANDER (1981), Phys. Rev. Lett. **47**, 1400; (1983), Phys. Rev. B **27**, 5686.
- 51** QUINN, M. J. (2004), *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Higher Education, New York.
- 52** MATH. & COMPUTER SCIENCE DIV., ARGONNE NAT. LAB., *The Message Passing Interface (MPI) Standard*, (updated May 9, 2006) <http://www-unix.mcs.anl.gov/mpi/>.
- 53** MATHEMATICS AND COMPUTER SCIENCE DIVISION, ARGONNE NATIONAL LABORATORY, *Web Pages for MPI and MPE*, (updated August 4, 2004) <http://www-unix.mcs.anl.gov/mpi/www>.
- 54** ACADEMIC COMPUTING & COMMUNICATIONS CENTER, UNIVERSITY OF ILLINOIS AT CHICAGO, *Argo Beowulf Cluster: MPI Commands and Examples*, (Updated 3 December 2004) [http://www.uic.edu/depts/accc/hardware/argo/mpi\\_routines.html](http://www.uic.edu/depts/accc/hardware/argo/mpi_routines.html); PACS TRAINING GROUP, *Introduction to MPI*, (updated 2001) <http://webct.ncsa.uiuc.edu:8900>

- /public/MPI/.
- 55** LUSK, W. E. AND A. SKJELUM (1999) *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed., MIT Press, Cambridge, MA.
- 56** PACHECO, P. S., (1997), *Parallel Programming with MPI*, Morgan Kaufmann, San Diego.
- 57** FOX, G., *Parallel Computing Works!*, (1994) Morgan Kaufmann, San Diego.
- 58** PANCAKE, C. M., (1996), *Is Parallelism for You?*, IEEE Computational Sci. & Eng., **3**, 18.
- 59** STERLING, T., J. SALMON, D. BECKER, AND D. SAVARESE (1999), *How to Build a Beowulf*, MIT Press, Cambridge, MA.
- 60** VAN DE VELDE, E. F. (1994), *Concurrent Scientific Computing*, Springer, New York.
- 61** DONGARRA, J., T. STERLING, H. SIMON, AND E. STROHMAIER (2005), *High-Performance Computing*, Comp. Sci. & Eng. **7**, 51.
- 62** AMDAHL, G., *Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities*, Proc. AFIPS., 483 (1967).
- 63** KERNIGHAN, B. AND D. RITCHIE (1988) *The C Programming Language*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ.
- 64** QUALLINE, S., *Practical C Programming*, (1997) O'Reilly & Associates, Sebastopol.
- 65** FOX, G., *HPJava: A Data Parallel Programming Alternative*, (2003), Compt. in Sci. & Eng., **5**, 60.
- 66** SUN N1 GRID ENGINE,  
[http://www.sun.com/software/  
gridware/](http://www.sun.com/software/gridware/).
- 67** GRAY, P. AND T. MURPHY (2006), Computing in Science & Engineering **8**, 82;  
<http://bccd.cs.uni.edu/>.
- 68** ARFKEN, G. B. AND H. J. WEBER (2001), *Mathematical Methods for Physicists*, Harcourt/Academic Press, San Diego.
- 69** MORSE, P. M. AND H. FESHBACH (1953), *Methods of Theoretical Physics*, McGraw-Hill, New York.
- 70** JACKSON, J. D. (1988), *Classical Electrodynamics*, 3rd ed., Wiley, New York.
- 71** KREYSZIG, E. (1998), *Advanced Engineering Mathematics*, 8th ed., Wiley, New York.
- 72** GARCIA, A. L. (2000), *Numerical Methods for Physics*, 2nd ed., Prentice-Hall, Upper Saddle River.
- 73** ANCONA, M. G. (2002), *Computational Methods for Applied Science & Engineering*, Rinton Press, Princeton, NJ.
- 74** RAWITSCHER, G., I. KOLTRACHT, H. DAI, AND C. RIBETTI (1996), Comput. Phys., **10**, 335.
- 75** ARGYRIS, J., M. HAASE AND J. C. HEINRICH (1991), Comput. Meth. Appl. Mech. Eng., **86**, 1.
- 76** CHRISTIANSEN, P. L. AND P. S. LOMDAHL (1981), Physica **2D**, 482.
- 77** CHRISTIANSEN, P. L. AND O. H. OLSEN (1978), Phys. Lett. **68A**, 185; (1979), Physica Scripta **20**, 531.
- 78** RUSSELL, J. S. (1844), *Report of the 14th Meeting of the British Association for the Advancement of Science*, John Murray, London.
- 79** KORTEWEG, D. J. AND G. DEVRIES (1895), Phil. Mag. **39**, 4.
- 80** ZABUSKY, N. J. AND M. D. KRUSKAL (1965), Phys. Rev. Lett. **15**, 240.
- 81** ASKAR, A. AND A. S. CAKMAK (1977), J. Chem. Phys. **68**, 2794.
- 82** VISSCHER, P. B. (1991), Comput. Phys. **5**, 596.
- 83** FEYNMAN, R. P. AND A. R. HIBBS (1965), *Quantum Mechanics and Path Integrals*, McGraw-Hill, New York.
- 84** MACKEOWN, P. K. (1985), Am. J. Phys. **53**, 880.
- 85** MACKEOWN, P. K. AND D. J. NEWMAN (1987) *Computational Techniques in Physics*, Adam Hilger, Bristol, UK.
- 86** MANNHEIM, P. D. (1983), Am. J. Phys. **51**, 328.
- 87** POTVIN, J. (1993), Comput. Phys. **7**, 149.
- 88** SINGH, P. P. AND W. J. THOMPSON (1993), Comput. Phys. **7**, 388.
- 89** HAFTEL, M. I. AND F. TABAKIN (1970), Nucl. Phys. **158**, 1.
- 90** PTPL, a 2D data plotter and histogram tool implemented in Java,  
[http://ptolemy.eecs.berkeley.edu/  
java/ptplot/](http://ptolemy.eecs.berkeley.edu/java/ptplot/).

## Index

- Abstraction, 46
- Accuracy, 467
- Address, 467
- Algorithm, 10, 467
- Alias, 257
- Amdahl's law, 318
- Analog, 467
- Animations, 467
- Annealing, 171
- Antiferromagnet, 168
- Applications, 467
- Architecture, 193, *see* Memory, 467
- Archive, 467
- Arithmetic unit, 468
- Arrays, 468
  - dimension, 471
- Attractors, 270–272, 290
  - predictable, 283, 286
  - strange, 286
- Backtracking, 86, 87, 94
- Ballistic deposition, 301, 307
- Base, 468
- BASIC, 8, 9
- Basic machine language, 8, 468
- Batch, 327, 468
- Baud, 468
- Beating, 225, 226
- Beowulf, 316, 322
- Bessel functions, 34, 35, 37, 215, 444, 449
- Bias, 17
- Bifurcation, 271, 272, 290
  - diagram, 272
  - dimension, 311
- Binary numbers, 13, 468
- Binary point, 16
- Binning, 274
- Binomial distribution, 121
- BIOS, 468
- Bisection algorithm, 83, 84, 238
- Bits, 13, 468
- Block walls, 168
- Blocking, 347
- Boltzmann distribution, 166
- Boolean, 468
- Boot, 468
- Bootable cluster, 326
- Bound states, 81, 235, 430, 431
- Boundary conditions, 211
- Box counting, 304, 305
- Box–Muller method, 163
- Break, 366
- Broadcast, 334
- Buffer, 182
- Bus, 316, 469
- Byte, 13, 468, 469
  - code, 9, 191, 469
- C language, 9
- Cache, 182, 203, 204, 469
  - data, 203
  - flow, 204
  - line, 182
  - misses, 203, 204
  - programming, 204
- Calling sequence, 469
- Capacitors, 353, 363
- Cauchy principal value, 453
- Central difference, 77
- Central processing unit, *see* CPU, 184
- Central storage, 182
- Chaos, 267, 286
  - Fourier analysis, 288
  - pendulum, 277
  - phase space, 285
- Chi squared, 126
- Child, 469
- Child class, 47
- CISC, 185, 186
- Class structure, 47
- Classes, 469
- Clock speed, 469, 470
- Code, 469
- Column-major order, 97, 181, 469
- Command-line interpreter, 8
- Communications, 347

*Computational Physics. Problem Solving with Computers (2nd edn).*

Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

ISBN: 978-3-527-40626-5

- collective, 347
- overhead, 319
- time, 319
- Communicator, 331
- Compilers, 9, 469, 470
  - just-in-time, 473
  - of languages, 9
- Computational
  - physics, 1
  - science, 1, 2
- Computer
  - basics, 7
  - languages, 7
- Concurrent processing, *see* Parallel Conflicts, 184
  - Control structures, 24, 470
- Correlations, 307
  - coefficient, 127
  - growth, 306, 307
- Covariance, 127
- CPU, 179, 182, 184, 189, 469
  - designs, 185
  - RISC, 185
  - time, 186
- Cubic splines, 116
- Cumulative distribution, 159
- Curie temperature, 168
- Curve fitting, *see* Data fitting
- Cycle time, 186, 470
  
- Data
  - cache, 203, *see* cache, 204
  - channels, 182
  - dependence, 314
  - dependency, 470
  - fitting, 111
  - lines, 182
  - parallel, 314
  - streams, 314
  - types, 470
- Deadlock, 346
- Decay
  - exponential, 121
  - spontaneous, 121
- Dependence, 314
- Dependency, 470
- Deposition, 301
  - ballistic, 302, 304
  - correlated ballistic, 307
- Derivative, 75
- Derivatives, 59, 75
  - central difference, 77
  - error, 78
  - extrapolated difference, 77
  - forward difference, 75, 76
  - second, 80
  
- DFT, 252
- Diagonalization, 96
- Differential equations, 207, 235
  - algorithms, 214
  - boundary conditions, 211
  - dynamical form, 212
  - Euler's rule, 215
  - initial conditions, 211
  - order, 210, 212
  - partial, 210, 351
  - Runge-Kutta, 215
  - types, 209, 351
- Differentiation, 75
- Diffusion-limited aggregation, 308
- Digital, 471
- Dimension, 471
  - array, 471
  - fractional, 293, 305, 310
  - Hausdorff-Besicovitch, 293
  - logical, 99
  - physical, 97, 99
- Directories, 471
- Discrete Fourier transform, 252, 259
- Dispersion, 399–401, 410
  - relation, 410
- Distributed memory, 317
- Double precision, 17, 22, 471
- double, 17
- Drag, 228, 230
- DRAM, 182, 471
- drand, 140
- Driving force, 226
  
- Eigenenergies, 235
- Eigenvalues, 95, 100, 102, 107, 211, 235, 236, 262, 417, 430
- Electrostatic potential, 353
- Elliptic integral, 280
- Encapsulation, 46
- Environmental variables, 324
- Equation
  - wave, 379
- Equations
  - Korteweg-de Vries, 410
  - differential, 207, 235
  - discrete, 148, 268
  - integral, 443, 444, *see* Integral, 451, 454, 456
  - integral solution, 455
  - motion, 228, 229, 231
  - Schrödinger, 417
  - Schrödinger, 417
  - Van der Pool, 291
- Ergodic, 171
- Errors, 29, 35, 79
  - algorithmic, 30, 39, 65

- approximation, 30, 39, 65
- empirical, 39, 71
- integration, 65, 71
- minimum, 41
- multiplicative, 33
- N-D integration, 156
- random, 30
- roundoff, 30, 34
- total, 39
- types, 29
- Ethernet, 471
- Euler's rule, 215
- Exchange energy, 167
- Executive
  - system, 9
  - unit, 182
- Exponent, 471
- Exponential decay, 121, 148
- Extrapolated difference, 77
- Fast Fourier transform, *see* FFT
- Feigenbaum constant, 276
- Ferromagnet, 168
- Fetch, 187
- Feynman
  - path integrals, 427
  - postulates, 428
  - propagator, 427
- FFT, 257, 471
- Fine grain, 315, 316
- Finite
  - difference equation, 149
  - differences, 149, 357, 419
- Fitting
  - best, 112
  - global, 126
  - goodness, 126
  - least squares, 124, 126
  - local, 126
  - Newton–Raphson, 132
  - nonlinear, 130
- Fixed points, 269, 284
- Fixed-point numbers, 14
- Floating-point numbers, 14, 471
  - float*, *see* Floating
- FLOP, 471
- Forth, 185
- Fortran, 9
- Forward difference, 76
- Fourier, 382
  - analysis, 246
  - chaos, 288
  - decompositon, 246
  - discrete transform, 252
  - fast transform (FFT), 257
  - integral, 245, 250
- PDE solution, 354
- power spectrum, 251
- sawtooth, 250
- series, 245, 246, 250
- theorem, 247
- transform, 245, 250, 251
- Fractals, 293
  - coastline, 303
  - dimension, 293, 295–297, 304
  - fern, 298
  - plants, 297
  - trees, 300
- Friction, 225, 228
- Functionals, 429
  - integration, 434
- G, 472
- Garbage, 29
- Gaussian
  - distribution, 160
  - elimination, 456
  - quadrature, 60, 66, 67
- Gibbs overshoot, 250
- Giga, 472
- Global optimization, 184
- Glossary, 467
- Gnuplot, 362, 363, 465
- Grace, 465
- Granularity, 315
- Green's function, 428
- Grid engine, 323, 326
- Growth Models, 293
- Guests, 329–331, 333
- GUI, 472
- Half-wave function, 249
- Hamilton's principle, 428
- Hardware, 179, 180, 189, 193
- Harmonics, 246
- Heap, 199
- Heat equation, 369
- Heat Flow, 351, 369
- Hexadecimal, 472
- High performance computing, 179, 189
- Hilbert transform, 454
- Host, 330, 332, 334, 341
- HPC, 189
- Huygens's principle, 428
- Hyperbolic point, 284
- IEEE floating point, 14, 16
- Importance sampling, 158
- Inheritance, 46
- Initial conditions, 211
- Instructions
  - stack, 182, 472

- streams, 314
- Integral equations, 445
- Integration, 59
  - error, 65, 71
  - Gaussian, 66, 69
  - mapping points, 68
  - mean value, 154
  - Monte Carlo, 153, 155
  - multidimensional, 155, 156
  - rejection techniques, 153
  - scaling, 68
  - Simpson’s rule, 63
  - splines, 118
  - trapezoid rule, 61
  - von Neumann rejection, 162
- Integro-differential equation, 443, 451
- Interpolation
  - Lagrange, 112, 114
  - splines, 116
- Interpreted language, 9
- Interpreter, 473
- Inverse, 96
- Ising model, 165, 166
- Jacobi method, 359
- Jacobian matrix, 93
- JAMA, 101
- Java
  - virtual machine, 191
- Just-In-Time compiler, 191
- Kernel, 8, 428, 473
- Korteweg-de Vries equation, 409
- Lagrange interpolation, 112, 114
- Languages
  - basic, 9
  - compiled, 9, 469, 470
  - computer, 7
  - high-level, 8, 472
  - interpreted, 9
  - low-level, 473
- LAPACK, 100, 132
- Laplace’s equation, 353
- Latency, 182, 319
- Lattice computations, 427, 432, 433
- Leap frog, 369
- Least-squares fit, 124, 126
- Length of coastline, 303
- Libraries, 98, *see Subroutines*
- Lifetime, 120, 121
- Limit cycles, 285
- Linear
  - algebra, 89, 101
  - congruent method, 138
  - regression, 127
- superposition, 211
- Link, 9
- Linux, 9
- Lippmann–Schwinger equation, 451
- Load, 9
  - module, 9
- Logical size, 99
- Logistics map, 267, 268
- Loop unrolling, 197, 202
- Machine
  - numbers, 15, 30
  - precision, 22, 23, 474
- Macro, 474
- Magnetic materials, 165, 166
- Mantissa, 15, 474
- Maple, 9
- Master and Slave, 330
- Matrices, 89, 95, 468
  - computing, 96
  - equations, 454
  - exercises, 98
  - inversion, 455, 456
  - libraries, 101
  - row-major order, 476
  - subroutine libraries, 100
- Mean value theorem, 154
- Memory, 179, 180, 189, 193
  - architecture, 89, 179, 189
  - conflicts, 184, 190
  - distributed, 316
  - dynamic allocation, 98
  - heap, 199
  - hierarchy, 181
  - pages, 97
  - physical , 98
  - virtual, 97, 183, 184, 187
- Messages, 317
  - passing, 314–317, 321
- Metropolis algorithm, 165, 169, 170, 427
- Microcode, 185
- Miller’s device, 35, 37
- MIMD, 314, 317
- Mode locking, 227, 285
- Model independence, 262
- Momentum space, 443, 451
- Monte Carlo
  - integration, 153
  - simulations, 137, 145
  - techniques, 137, 145, 148
- MPI, 313, 321, 326
  - commands, 349
- Multitasking, 183, 184
- NAN, 20
- Netlib, 105

- Newton–Raphson
  - algorithm, 84
  - backtracking, 86
  - N-D, 92
- Newton–Cotes methods, 60
- Nodes, 117, 314
- Nonlinear
  - dynamics, 267–269, 271, 277
  - equations, 211
  - harmonics, 245
  - limit cycles, 285
  - maps, 269, 276
  - oscillations, *see* Oscillations, 223
  - pendulum, 277
- Nonlocal potentials, 443, 451
- Normal distribution, 160
- Normal mode expansions, 245–247, 354, 381
- Normal numbers, 16
- Numbers
  - binary, 13
  - fixed-point, 14, 15
  - floating-point, 14, 15, 471
  - hexadecimal, 13, 472
  - IEEE, 16
  - machine, 15
  - normal, 16
  - octal, 13
  - range, 13
  - representation, 13
  - subnormal, 16
  - uniform, 142
- Numerical quadrature, 59
- Nyquist criterion, 258
- Object-oriented programming\*, 45
- Objects, 46, 474
  - code, 9
  - oriented programming, 45
- Octal numbers, 13
- ODEs, 207, 213, 235
  - second order, 229
- One cycle, 269
- Operands, 182
- Operating system, 8, 9
- Optimization, 189, 190
- Oscillations
  - anharmonic, 208
  - damped, 225
  - harmonic, 222
  - isochronous, 222
  - nonlinear, 207, 208, 222, 226, 245
- Overdetermined, 95
- Overflows, 14, 20
- Overhead, 319
- Overrelaxation, 359
- Padding of signal, 259
- Page, 97, 183
  - fault, 183
- Parallel computing, 313
  - Beowulf, 322
  - granularity, 315
  - performance, 317
  - subroutines, 316
  - tuning, 340
  - types, 314
- Parallelism, 314
- Parent class, 47
- Partial differential equations, 351
- Pascal, 185
- Path integration, 427, 428, 430, 432–434
- PDEs, 210, 351, 353, 369, 379, 399, 417
  - elliptic, 353
  - explicit solution, 419
  - hyperbolic, 379
  - implicit solution, 419
  - nonlinear, 409
  - parabolic, 369
  - types, 351
  - .pdf, 475
- Pendulum, 290
  - analytic solution, 280
  - bifurcation diagram, 290
  - chaotic, 277, 286, 290
  - coupled, 399
  - free, 279
  - realistic, 277
- Performance, 193
- Period doubling, 270, 271
- Phantom bit, 17
- Phase space, 282, 285, 291
- Phase transition, 165
- Pipeline, 184, 187
- Planetary motion, 228, 232
- Plots
  - complex, 362
  - data, 461
  - surface, 362
- Pointers, 99, 330, 344
- Polymorphism, 46
- PostScript, 363
- Potential
  - delta shell, 447
- Power
  - residue method, 138
  - spectrum, 251
- Precision, 29
  - empirical, 23
  - machine, 22, 23
  - tests, 224
- Principal values, 453
- `printf`, 10

- Probability, 121
- Problem solving
  - environments, 3
  - paradigm, 2
- Programming, 9, 12
  - cache, 204
  - design, 24
  - registers, 204
  - structured, 24
  - virtual memory, 184
- Projectile motion, 228
- Propagator, 428, 434
  - .ps, 363, 475
- Pseudo-random, 138
- Pseudocode, 9, 10, 26, 27, 476
- PtPlot, 461
  - comments, 464
- Pulsons, 404
- PVM, 313, 321
- Quadrature, 59
- Quantum mechanics, 81, 427, 443, 451
  - bound states, 235
  - scattering, 443, 451
- Radioactive decay, 148
- RAM, 182
- Random, 137
  - generators, 138, 163, 275
  - linear congruent, 138
  - nonuniform, 158, 159
  - numbers, 137, 275
  - pseudo, 138
  - sequences, 137, 138, 140
  - tests, 141, 144
  - walk, 145, 147
- Rank, 330
- Reduce, 335
- Reference calls, 133
- Registers, 22, 182, 204, 476
  - working, 22
- Rejection, 153
- Relaxation, 359
- Resonances, 226
  - nonlinear, 225
- RISC, 185, 186, 476
- rk, 215
- Romberg extrapolation, 72
- Root mean square, 145, 146
- Row-major order, 97, 181, 476
- Runge-Kutta, 219
- Runge-Kutta, 215
- Sampling, 153, 171, 252
  - importance, 158
- Sawtooth function, 248
- scanf, 10
- Java, 10
- Scattering, 443, 451, 458
- Schrödinger equation, 417, 423, 443, 445, 451
  - time dependent, 417
- Searching, 81, 83, 84, 238
- Section, 187
- Secular equation, 96
- Seed, 138, 269, 271
- Self
  - affine connection, 297
  - affinity, 300
  - limiting, 291
  - similar, 275, 296
- Self similarity, 297
- Separatrix, 224, 281, 415
- Serial, 318
- Series summation, 26
- Shells, 8, 9, 12
- Shock waves, 410
- Sierpiński gasket, 293, 294
- Sign bit, 20
- Significant figures (parts), 31
- SIMD, 314
- Simpson's rule, 63, 64
- Simulated annealing, 111, 165
- Simulation, 137
- Sinc filter, 258
- Sine-Gordon equation, 402, 403
- Single precision, 22
- single, 17
- Singular integrals, 452
- SISD, 314
- SLATEC, 100, 105
- Slave, 330
- Solitons, 399, 409, 410
  - crossing, 415
  - KdV, 411
  - ring, 404
  - Sine-Gordon, 403
  - water wave, 409
- Spectral function, 251
- Speed, 179, 189
- Speedup, 336
- Splines, 116
  - cubic, 116
  - natural, 117
- Spontaneous decay, 148
- SRAM, 182
- Stack, 472
- Statements
  - assignment, 468
- Statistics, 121
- Stochastic, 148
- Storage, 187

- Strange attractors, 286
- Stride, 203, 205, 477
- Subnormal numbers, 16
- Subroutines, 9, 477
  - libraries, 89, 100, 101, 105, 132
- Subscripts, 97
  - schemes, 97
- Subtractive cancellation, 31
- Supercomputer, 477
- Superscalar, 477
- Swap space, 183, 184
- TCP/IP, 477
- Thermodynamics, 165, 175
- Three-body problem, 233
- Time stepping, 369, 382
- Trapezoid rule, 61
- Trial and error searching, 81, 83
- Trials, 121
- Trivial solutions, 96, 447
- Tuning, 179, 189
- Two cycle, 271
- Two's compliment, 21
- Underflows, 20
- Uniform
  - distribution, 137
  - sequences, 138, 142
  - tests, 144
  - weight, 160
- Uniformity, 141
- Unix, 9
- Van der Pool equation, 291
- Variance, 127, 158
  - reduction, 158
- Vector processors, 186
- Vectors, 101, 187, 477
- Virtual Machine, 191
- Virtual memory, 97, 183, 184, 187, 190, 192
- Visualization, 5, 274
  - animations, 422
- von Neumann
  - rejection, 162, 172
  - stability assessment, 373
- Wave
  - equation, 379, 390
  - functions, 427, 439, 449, 458
  - packets, 250, 417, 422
  - shallow water, 409
  - shock, 410
  - solitons, 399
  - string, 379
- Web, XX
- Windows, 9
- Word length, 13
- Worker, 330
- Working set size, 190
- World Wide Web, XX
- Wrappers, 327
- Zero finding, 235