Python Multimodal eTextBook β4.1 (ComPADRE)

A Survey of Computational Physics

Introductory Computational Science

RUBIN H. LANDAU

Oregon State University

MANUEL JOSÉ PÁEZ

University of Antioquia

CRISTIAN C. BORDEIANU

University of Bucharest

Video Production by **Sally Haerer**

Applets by Páez CP Group, Hans Kowallik, David Wolf and Bert Laubsch

PRINCETON UNIVERSITY PRESS

PRINCETON AND OXFORD

Copyright © 2012 by Princeton University Press
Copyright © 2012 by Landau, Páez & Bordeianu

Published by Princeton University Press,
41 William Street, Princeton, New Jersey 08540
In the United Kingdom: Princeton University Press,
3 Market Place, Woodstock, Oxfordshire OX20 1SY
pup.princeton.edu

All Rights Reserved

Paper Version ISBN 978–0–691–13137–5
Library of Congress Control Number 2007061029 (Paper)
British Library Cataloging-in-Publication Data is available

Archived with National Science Digital Library, Compadre. ∞

Supported by the US National Science Foundation.

In memory of the parents

Bertha Israel Landau, Philip Landau, and Sinclitica Bordeianu

Contents

Preface **ii**

1. Computational Science Basics 1 **1**

1.1 Software Needed to Use This eBook 1
1.2 Using The Features of This eBook 4
1.3 Computational Physics and Computational Science 7
1.4 Viewing the Subjects to be Covered 8
1.5 Making Computers Obey; Languages (Theory) 11
1.6 Programming Warmup 14
1.7 Computer Number Representations (Theory) 21
1.8 Problem: Summing Series 30

2. Errors & Uncertainties in Computations 2 **32**

2.1 Types of Errors (Theory) 32
2.2 Errors in Spherical Bessel Functions (Problem) 37
2.3 Experimental Error Investigation (Problem) 40

3. Visualization Tools 3 **45**

3.1 Data Visualization (General) 46
3.2 Python Matplotlib, Visual (VPython) Packages 46
3.3 Using Packages in Python Programs 47
3.4 Matplotlib for 2-D Graphs in Python 48
3.5 2-D Plots with Visual (VPython) 53
3.6 Animations with Visual (VPython) 56
3.7 Gnuplot: Reliable 2-D and 3-D Plots 57
3.8 Texturing and 3-D Imaging 65
3.9 Grace/ACE: Superb 2-D Graphs for Unix/Linux 65

4. Python Object-Oriented Programs: Impedance & Batons 4 **71**

4.1 ***Unit I. Basic Objects: Complex Impedance*** 71
4.2 Complex Numbers (Math) 72
4.3 Resistance Becomes Impedance (Theory) 73
4.4 Abstract Data Structures, Objects (CS) 74
4.5 Python's Built-in Complex Number Type 80
4.6 Complex Currents (Solution) 80
4.7 OOP Worked Examples 81
4.8 Subclasses and Class Inheritance 85
4.9 ***Unit II. Advanced Objects: Baton Projectiles*** ⊙ 85
4.10 Trajectory of a Thrown Baton (Problem) 86
4.11 OOP Design Concepts (CS) 88

4.12 Multiple Classes and Multiple Inheritances 89
4.13 Multiple Inheritances, Classes in One File 92
4.14 Multiple Inheritance, Separate Files 93
4.15 OOP Example: Superposition of Motions 101
4.16 Newton's Laws of Motion (Theory) 101
4.17 OOP Class Structure (Method) 102
4.18 Python Implementation 102

5. Monte Carlo Simulations (Nonthermal) 5 **104**
5.1 ***Unit I. Deterministic Randomness*** 104
5.2 Random Sequences (Theory) 104
5.3 **Unit II. Monte Carlo Applications** 109
5.4 A Random Walk (Problem) 109
5.5 Radioactive Decay (Problem) 112
5.6 Decay Implementation and Visualization 115

6. Integration 6 **116**
6.1 Integrating a Spectrum (Problem) 116
6.2 Quadrature as Box Counting (Math) 116
6.3 Experimentation 125
6.4 Higher-Order Rules (Algorithm) 125
6.5 Monte Carlo Integration by Stone Throwing 126
6.6 High-Dimensional Integration (Problem) 128
6.7 Integrating Rapidly Varying Functions (Problem) 129
6.8 Nonuniform Assessment ⊙ 132

7. Differentiation & Searching 7 **135**
7.1 ***Unit I. Numerical Differentiation*** 135
7.2 Forward Difference (Algorithm) 136
7.3 Central Difference (Algorithm) 136
7.4 Extrapolated Difference (Method) 137
7.5 Error Analysis (Assessment) 138
7.6 Second Derivatives (Problem) 139
7.7 ***Unit II. Trial-and-Error Searching*** 140
7.8 Quantum States in a Square Well (Problem) 140
7.9 Trial-and-Error Roots via Bisection Algorithm 140
7.10 Newton–Raphson Searching (Improved Algorithm) 142

8. Matrix Equation Solutions; Data Fitting 8 **146**
8.1 ***Unit I. Systems of Equations, Matrix Computing*** 146
8.2 Two Masses on a String 147
8.3 Classes of Matrix Problems (Maths) 150
8.4 Numerical Python and Python Matrix Library 155
8.5 ***Unit II. Data Fitting*** 162
8.6 Fitting Exponential Decay (Problem) 167
8.7 Least-Squares Fitting (Method) 170

9. Differential Equation Applications 9 **178**
9.1 ***Unit I. Free Nonlinear Oscillations*** 178
9.2 Nonlinear Oscillators (Models) 179
9.3 Types of Differential Equations (Math) 180

9.4 Dynamic Form for ODEs (Theory) 181
9.5 ODE Algorithms 183
9.6 Solution for Nonlinear Oscillations (Assessment) 191
9.7 Extensions: Nonlinear Resonances, Beats, Friction 192
9.8 Extension: Time-Dependent Forces 194
9.9 ***Unit II. Binding A Quantum Particle*** 194
9.10 Theory: Quantum Eigenvalue Problem 195
9.11 Algorithm: Eigenvalues via ODE Solver + Search 196
9.12 Explorations 202
9.13 ***Unit III. Scattering, Projectiles & Orbits*** 203
9.14 Problem 1: Classical Chaotic Scattering 203
9.15 Problem 2: Balls Falling Out of the Sky 206
9.16 Theory: Projectile Motion with Drag 207
9.17 Problem 3: Planetary Motion 208

10. Fourier Analysis: Signals and Filters 10 211

10.1 ***Unit I. Fourier Analysis of Nonlinear Oscillations*** 211
10.2 Fourier Series (Math) 212
10.3 Exercise: Summation of Fourier Series 214
10.4 Fourier Transforms (Theory) 215
10.5 ***Unit II. Filtering Noisy Signals*** 223
10.6 Noise Reduction via Autocorrelation (Theory) 224
10.7 Filtering with Transforms (Theory) 227
10.8 ***Unit III. Fast Fourier Transform Algorithm (FFT)*** ⊙ 231
10.9 FFT Implementation 234
10.10 FFT Assessment 234

11. Wavelet Analysis & Data Compression 11 238

11.1 ***Unit I. Wavelet Basics*** 238
11.2 Wave Packets and Uncertainty Principle (Theory) 240
11.3 Short-Time Fourier Transforms (Math) 241
11.4 The Wavelet Transform 242
11.5 ***Unit II. Discrete Wavelet Transforms, MRA*** ⊙ 246

12. Discrete & Continuous Nonlinear Dynamics 12 259

12.1 ***Unit I. Bug Population Dynamics (Discrete)*** 259
12.2 The Logistic Map (Model) 259
12.3 Properties of Nonlinear Maps (Theory) 260
12.4 Mapping Implementation 262
12.5 Bifurcation Diagram 263
12.6 Logistic Map Random Numbers (Exploration)· 265
12.7 Other Maps (Exploration) 266
12.8 Signals of Chaos: Lyapunov Coefficients ⊙ 266
12.9 Unit I Quiz 268
12.10 ***Unit II. Pendulums Become Chaotic*** 270
12.11 Chaotic Pendulum ODE 270
12.12 Visualization: Phase Space Orbits 273
12.13 Exploration: Bifurcations of Chaotic Pendulums 279
12.14 Alternate Problem: The Double Pendulum 280
12.15 Assessment: Fourier/Wavelet Analysis of Chaos 282
12.16 Exploration: Alternate Phase Space Plots 283

12.17 Further Explorations 283
12.18 ***Unit III. Coupled Predator–Prey Models*** ⊙ 284
12.19 Lotka–Volterra Model 285

13. Fractals & Statistical Growth 13 **290**
13.1 Fractional Dimension (Math) 290
13.2 The Sierpiński Gasket (Problem 1) 291
13.3 Beautiful Plants (Problem 2) 293
13.4 Ballistic Deposition (Problem 3) 296
13.5 Length of British Coastline (Problem 4) 298
13.6 Correlated Growth, Forests, Films (Problem 5) 301
13.7 Globular Cluster (Problem 6) 301
13.8 Fractals in Bifurcation Plot (Problem 7) 305
13.9 Fractals from Cellular Automata 305
13.10 Perlin Noise Adds Realism ⊙ 307
13.11 Quiz 311

14. HPC Hardware, Tuning, Parallel Computing 14 **312**
14.1 ***Unit I. High-Performance Computers (CS)*** 312
14.2 Memory Hierarchy 313
14.3 The Central Processing Unit 316
14.4 CPU Design: Reduced Instruction Set Computer 316
14.5 CPU Design: Multiple-Core Processors 317
14.6 CPU Design: Vector Processor 318
14.7 ***Unit II. Parallel Computing*** 319
14.8 Parallel Semantics (Theory) 319
14.9 Distributed Memory Programming 321
14.10 Parallel Performance 322
14.11 Parallelization Strategy 324
14.12 Practical Aspects of MIMD Message Passing 325
14.13 Example of a Supercomputer: IBM Blue Gene/L 327
14.14 Exascale Computing with Multinode-Multicores 329
14.15 ***Unit III. HPC Program Optimization*** 331
14.16 Programming for the Data Cache (Method) 340
14.17 Practical Tips for Multicore, GPU Programming 342

15. Thermodynamic Simulations, Quantum Path Integration 15 **345**
15.1 ***Unit I. Magnets via the Metropolis Algorithm*** 345
15.2 An Ising Chain (Model) 346
15.3 Statistical Mechanics (Theory) 347
15.4 Metropolis Algorithm 348
15.5 ***Unit II. Magnets via Wang–Landau Sampling*** ⊙ 354
15.6 Wang–Landau Sampling 356
15.7 ***Unit III. Feynman Path Integrals*** ⊙ 361
15.8 Feynman's Space-Time Propagation (Theory) 361
15.9 Exploration: Quantum Bouncer's Paths ⊙ 370

16. Simulating Matter with Molecular Dynamics 16 **374**
16.1 Molecular Dynamics (Theory) 374
16.2 Verlet and Velocity-Verlet Algorithms 380
16.3 1-D Implementation and Exercise 381

16.4 Trajectory Analysis 384
16.5 Quiz 384

17. PDEs for Electrostatics & Heat Flow 17 386

17.1 PDE Generalities 386
17.2 ***Unit I. Electrostatic Potentials*** 387
17.3 Fourier Series Solution of a PDE 388
17.4 Solution: Finite-Difference Method 391
17.5 Assessment via Surface Plot 394
17.6 Alternate Capacitor Problems 395
17.7 Implementation and Assessment 397
17.8 Electric Field Visualization (Exploration) 398
17.9 Laplace Quiz 398
17.10 ***Unit II. Finite-Element Method*** ⊙ 399
17.11 Electric Field from Charge Density (Problem) 399
17.12 Analytic Solution 400
17.13 Finite-Element (Not Difference) Methods 400
17.14 FEM Implementation and Exercises 404
17.15 Exploration 406
17.16 ***Unit III. Heat Flow via Time-Steps (Leapfrogs)*** 406
17.17 The Parabolic Heat Equation (Theory) 407
17.18 Assessment and Visualization 412
17.19 Improved Heat Flow: Crank–Nicolson Method 413

18. PDE Waves: String, Quantum Packet, E&M 18 418

18.1 ***Unit I. Vibrating String*** 418
18.2 The Hyperbolic Wave Equation (Theory) 418
18.3 Waves with Friction (Extension) 424
18.4 Waves for Variable Tension and Density (Extension) 425
18.5 ***Unit II. Quantum Wave Packets*** 429
18.6 Time-Dependent Schrödinger Equation (Theory) 429
18.7 Algorithm for the 2-D Schrödinger Equation 433
18.8 ***Unit III. E&M Waves, Finite-Difference t Domain*** ⊙ 435
18.9 Maxwell's Equations 435
18.10 FDTD Algorithm 436

19. Solitons & Computational Fluid Dynamics 19 443

19.1 ***Unit I. Advection, Shocks, Russell's Soliton*** 443
19.2 Theory: Continuity and Advection Equations 444
19.3 Theory: Shock Waves via Burgers' Equation 445
19.4 Including Dispersion 448
19.5 Shallow-Water Solitons; the KdeV Equation 449
19.6 ***Unit II. River Hydrodynamics*** 453
19.7 Hydrodynamics, the Navier–Stokes Equation (Theory) 453
19.8 2-D Flow over a Beam 459
19.9 Theory: Vorticity Form of Navier–Stokes Equation 459

20. Integral Equations in Quantum Mechanics 20 468

20.1 ***Unit I. Bound States of Nonlocal Potentials*** 468
20.2 Momentum-Space Schrödinger Equation (Theory) 469
20.3 ***Unit II. Nonlocal Potential Scattering*** ⊙ 473

20.4 Lippmann–Schwinger Equation (Theory) 473

A. Glossary A **479**

B. Installing Python, Matplotlib, NumPy B **485**

C. Software Directories C **486**

D. Compression via DWT with Thresholding D **488**

D.1 More on Thresholding 490

D.2 Wavelet Implementation and Assessment 491

Bibliography **492**

Index **502**

Preface

Creating this eTextBook in a time of rapidly-changing technology has been like seeing a dream come true. Starting in 1995, while we were writing our first *Computational Physics* (CP) text, we envisioned the nascent World Wide Web and its technologies as providing enhancements to a text book by permitting interactions with the materials via a variety of senses, modes of learning and technologies. Even during (what seems to us as) the short times of our own careers we have seen changes in the way science is done due to rapidly-changing technology. Likewise, we have seen in these last few years changes in what is viewed as a textbook and even in what young people view as their the real world. Nevertheless, over the last 16 years we have continued to develop CP courses, record video lectures and develop enhancements to learning using Web technologies. Our aim with this eTextBook is to explore a new model for textbooks and courses, and to use this new model to promote and develop Computational Physics

While we have peacefully continued our developments during the emeritus years of MJP and RHL, technology and society's views as to how one reads and where is the boundary between virtual and true reality. At the moment we see a generation of students who view video and on-line interactions as much closer to reality than their elders, and who do much of their reading without touching paper. In addition, the trade journals and technology pages of our beloved New York Times indicate that the publishing and technology sectors are taking an increasing interest in eBooks and in new devices that can present eBooks with more advanced Web technologies. So while we do not feel like our views have changed that much in recent years, it is a pleasure to think, however immodestly, that the world is now providing the means to make our vision of an eTextBook achievable.

Even in 2006, while completing the Java version of our *Survey* text, we realized that it made good sense to also have a Python version of our text as well. Python seems to be the easiest and most accessible language for beginning programming, while at the same time being used for interactive and exploratory computations in scientific research. In addition, Chabay and Sherwood's new introductory physics text [C&S 10] takes a computational view based on VPython, and thus serves as an excellent segue for a CP course using Python. Finally, we believe that the important aspects of computational modeling should transcends any particular computer language and should not be strongly focused on programming, and so having a Python alternative is a way of supporting this view.

All the pieces in this eTextBook came together when, upon completing the year-long editing of the Java-based paper text, we approached our editor Vickie Kearn at Princeton University Press with the possibility of having a Python version of *A Survey*. After economic realities were explained to us, we hit upon the idea of creating an eTextBook using Python that could also contain many of the the multimodal enhancements possible with modern technologies. In particular, this would let us push eBooks beyond just images of paper pages into also including video lectures, active simulations, editable codes, animations and sounds within a framework that could be made available to large numbers of students and researchers. With help from

the NSF's CCLI/TUES program, Oregon State University College of Science and Microsoft Corporation, we have been able to bring the project to completion.

In the spring of 2010 we purchased an Amazon Kindle and an Apple iPad in order to test their effectiveness for reading eBooks and eTextBooks. Although the present-day Kindle does not have much in the way of multimodal capabilities, we found it excellent for reading "flat" books such as mystery and spy novels. The iPad 1, on the other hand, while clumsy for bed-time reading, has impressive multimodal capabilities with its brilliant screen, and with its bigger size is more appropriate for the display of equations and figures. However, we found none of the native eBook languages appropriate for a text containing mathematics and tables, but did find that they could read a pdf version of the text in which the technical stuff looked just fine. Accordingly, we have prepared an alternative version of the text (option B) meant for mobile devices in which the pdf pages are formatted to fit a small screen better and in which the multimedia are stored on the cloud.

As the use of the iPad has skyrocketed in this last year we realized that, like-it-or-not, some good fraction of the "readers" of our eBook might be reading on an iPad, where our video-based lectures should look great. Unfortunately, the video modules that took years to make used Adobe Flash, a technology that Apple refused to implement on the iPad. Accordingly are converting all of our lectures to an mp4 format which will play on the iPad, but with some reduction in functionality. We envision that readers may want the flexibility of having both the PC and mobile-device versions of the text. The PC version would have all local files, and so all should load quickly, while the mobile device (option B) version links to media "in the cloud", which avoids the use of large amounts of storage (15 GB) on small devices, but is slower to load. More information can be found in Chapter 1. [We are just beginning to explore use of the book on Android devices, and so will update these pages when we can report on the results.]

While the eBook presented here is certainly a prototype, it is a complete and serious text meant for immediate classroom or online use. Its features, and especially the new ones for an eBook, are explained in Chapter 1. Also given in Chapter 1 are several graphical concept maps with active links to chapters and sections, the different maps appropriate for different level readers. Although these maps are a step towards a personalizable text, this book does not advance as far as we would prefer in that direction. We have reorganized chapters and units, but have not yet laid out paths with prerequisites to individual modules created from the text. However, because this is an eBook, changes will probably be made continually and we encourage your opinions and suggestions.

Python eBook Acknowledgments

We are particularly thankful to Bruce Sherwood, who has assisted us in making the Python codes run faster and look better. The errors in the codes remain our responsibility. The content in the text has benefitted from the contributions of C. E. Yaguna, J. Zuluaga, Oscar A. Restrepo, Guillermo Avendano-Franco, Sally Haerer (video production), Hans Kowallik (applets), Bertrand Laubsch (Java sound, decay simulation), Connelly Barnes (OOP), Zlatko Dimcovic (Wavelets, Java I/O) and Joel Wetzel (figures).

Preface to the Paper Java Edition

In the decade since two of us wrote the book *Computational Physics* (CP), we have seen a good number of computational physics texts and courses come into existence. This is good. Multiple texts help define a still-developing field and provide a choice of viewpoints and focus. After perusing the existing texts, we decided that the most worthwhile contribution we could make was to extend our CP text so that it surveys many of the topics we hear about at computa-

tional science conferences. Doing that, while still keeping the level of presentation appropriate for upper-division undergraduates or beginning graduate students, was a challenge.

As we look at what we have assembled here, we see more than enough material for a full year's course (details in Chapter 1 "Computational Science Basics"). When overlapped with our new lower-division text, *A First Course in Scientific Computing*, and when combined with studies in applied mathematics and computer science, we hope to have created a path for undergraduate computational physics/science education to follow.

The ensuing decade has also strengthened our view that the physics community is well served by having CP as a prominent member of the broader *computational science and engineering* (CSE) community. This view affects our book in two ways. First, we present CP as a multidisciplinary field of study that contains elements from applied mathematics and computer science, as well as physics. Accordingly, we do not view the pages we spend on these subjects as space wasted not studying physics but rather as essential components of a multidisciplinary education. Second, we try to organize and present our materials according to the steps in the scientific problem-solving paradigm that lie at the core of CSE:

$$\text{Problem} \leftrightarrow \text{theory} \leftrightarrow \text{model} \leftrightarrow \text{method} \leftrightarrow \text{implementation} \leftrightarrow \text{assessment}.$$

This format places the subject matter in its broader context and indicates how the steps are applicable to a wider class of problems. Most importantly, educational assessments and surveys have indicated that some students learn science, mathematics, and technology better when they are presented together in context rather than as separate subjects. (To some extent, the loss of "physics time" learning math and CS is made up for by this more efficient learning approach.) Likewise, some students who may not profess interest in math or CS are motivated to learn these subjects by experiencing their practical value in physics problem solving.

Though often elegant, we view some of the new CP texts as providing more of the theory behind CP than its full and practical multidisciplinary scope. While this may be appropriate for graduate study, when we teach from our texts we advocate a learn-by-doing approach that requires students to undertake a large number of projects in which they are encouraged to make discoveries on their own. We attempt to convey that it is the students' job to solve each problem professionally, which includes understanding the computed results. We believe that this "blue-collar" approach engages and motivates the students, encompasses the fun and excitement of CP, and stimulates the students to take pride in their work.

As computers have become more powerful, it has become easier to use complete problem-solving environments like Mathematica, Maple, Matlab, and Femlab to solve scientific problems. Although these environments are often used for serious work, the algorithms and numerics are kept hidden from the user, as if in a black box. Although this may be a good environment for an experienced computational scientist, we think that if you are trying to *learn* scientific computation, then you need to look inside the black box and get your hands dirty. This is probably best done through the use of a compiled language that forces you to deal directly with the algorithm and requires you to understand the computer's storage of numbers and inner workings.

Notwithstanding our viewpoint that being able to write your own codes is important for CP, we also know how time-consuming and frustrating debugging programs can be, especially for beginners. Accordingly, rather than make the reader write all their codes from scratch, we include basic programs to modify and extend. This not only leaves time for exploration and analysis but also more realistically resembles a working environment in which one must incorporate new developments with preexisting developments of others.

The choice of Java as our prime programming language may surprise some readers who

know it mainly for its prowess in Web computing (we do provide Fortran77, Fortran95, and C versions of the programs on the CD). Actually, Java is quite good for CP education since it demands proper syntax, produces useful error messages, and is consistent and intelligent in its handling of precision (which C is not). And when used as we use it, without a strong emphasis on object orientation, the syntax is not overly heavy. Furthermore, Java now runs on essentially all computer systems in an identical manner, is the most used of all programming languages, and has a universal program development environment available free from Sun [SunJ], where the square brackets refer to references in the bibliography. (Although we recommend using shells and jEdit [jEdit] for developing Java programs, many serious programmers prefer a development platform such as Eclipse [Eclipse].) This means that practitioners can work just as well at home or in the developing world. Finally, Java's speed does not appear to be an issue for educational projects with present-day fast computers. If more speed is needed, then conversion to C is straightforward, as is using the C and Fortran programs on the CD.

In addition to multilanguage codes, the CD also contains animations, visualizations, color figures, interactive Java applets, MPI and PVM codes and tutorials, and OpenDX codes. More complete versions of the programs, as well as programs left for exercises, are available to instructors from RHL. There is also a digital library version of the text containing streaming video lectures and interactive equations under development.

Specific additions to this book, not found in our earlier CP text, include chapters and appendixes on visualization tools, wavelet analysis, molecular dynamics, computational fluid dynamics, MPI, and PVM. Specific subjects added to this text include shock waves, solitons, IEEE floating-point arithmetic, trial-and-error searching, matrix computing with libraries, object-oriented programming, chaotic scattering, Lyapunov coefficients, Shannon entropy, coupled predator–prey systems, advanced PDE techniques (successive overrelaxation, finite elements, Crank–Nicholson and Lax–Wendroff methods), adaptive-step size integrators, projectile motion with drag, short-time Fourier transforms, FFT, filtering, Wang–Landau simulations of thermal systems, Perlin noise, cellular automata, and waves on catenaries.

Acknowledgments

This book and the courses it is based upon could not have been created without financial support from the National Science Foundation's CCLI, EPIC, and NPACI programs, as well as from the Oregon State University Physics Department and the College of Science. Thank you all and we hope you are proud.

> *Immature poets imitate;*
> *mature poets steal.*
>
> — T. S. Elliot

Our CP developments have followed the pioneering path paved with the books of Thompson, Koonin, Gould and Tobochnik, and Press *et al.*; indubitably, we have borrowed material from them and made it our own. We wish to acknowledge the many contributions provided by Hans Kowallik, who started as a student in our CP course, continued as a researcher in early Web tutorials, and has continued as an international computer journeyman. Other people have contributed in various places: Henri Jansen (early class notes), Juan Vanegas (OpenDX), Connelly Barnes (OOP and PtPlot), Phil Carter and Donna Hertel (MPI), Zlatko Dimcovic (improved codes and I/O), Joel Wetzel (improved figures and visualizations), Oscar A. Restrepo (QMCbouncer), and Justin Elser (system and software support).

It is our pleasure to acknowledge the invaluable friendship, encouragement, helpful discussions, and experiences we have had with our colleagues and students over the years. We are

particularly indebted to Guillermo Avendaño-Franco, Saturo S. Kano, Bob Panoff, Guenter Schneider, Paul Fink, Melanie Johnson, Al Stetz, Jon Maestri, David McIntyre, Shashikant Phatak, Viktor Podolskiy, and Cherri Pancake. Our gratitude also goes to the reviewers Ali Eskanarian, Franz J. Vesely, and John Mintmire for their thoughtful and valuable suggestions, and to Ellen Foos of Princeton University Press for her excellent and understanding production work. Heartfelt thanks goes to Vickie Kearn, our editor at Princeton University Press, for her encouragement, insight, and efforts to keep this project alive in various ways.

In spite of everyone's best efforts, there are still errors and confusing statements for which we are responsible.

Finally, we extend our gratitude to the wives, Jan and Lucia, whose reliable support and encouragement are lovingly accepted, as always.

Chapter One

Computational Science Basics

Some people spend their entire lives reading but never get beyond reading the words on the page; they don't understand that the words are merely stepping stones placed across a fast-flowing river, and the reason they're there is so that we can reach the farther shore; it's the other side that matters.

—José Saramago

As an introduction to the eBook to follow, we start this chapter with a discussion of the software needed to utilize the various multimedia features it contains, and the symbols used to indicate and activate these features. (A discussion of the motivation for and history of this eBook can be found in preface.) Then we give a description of how computational physics (CP) fits into the broader field of computational science, and what topics constitute the contents of CP. We get down to basics by examining computing languages, number representations, and programming. Related topics dealing with hardware basics are found in Chapter 14, "High-Performance Computing Hardware, Tuning, and Parallel Computing." *Although it is tucked away inside the text, some readers may benefit from reading at least the first half of Chapter 14 dealing with the basics of computer memory right after they have read about the basics of software.*

1.1 SOFTWARE NEEDED TO USE THIS EBOOK

This eBook is designed to be read as a hyperlinked *portable document format* (pdf) file. So, the first thing you need is a pdf reader. The standard is *Adobe Acrobat Reader* (9 or later), which is free. We recommend using *Adobe Reader Pro* so that you will be able to write comments in the book, customize its presentation in manifold ways, and even add your own materials and remove ours as needed for your particular purpose. (We still maintain copyright over our materials and do appreciate seeing that acknowledged as one might a reference in a scientific paper.) We have developed this eBook with support from the US National Science foundation on Windows operating systems, but have tested it with Macs, Linux, Kindle and iPad and Android. Please let us know if something does not work.

Because this document links to *HTML* pages, you also will need a Web browser such as *Firefox, Chrome, Internet Explorer*, or *Safari*. In fact, you may want to have two so that you can try the other one when something does not work right on the first one. Furthermore, because some of the media types used in this text opens in browsers (more about that later), it is likely that you *will need to set your browser options so that it has* ***permissions*** *to open file types, like applets, that execute.* (This is in addition to granting similar permissions in your pdf reader.) In order to run the applets you must have a copy of the *Java Runtime Environment (JRE)* on your system. If you want to display the *xml* versions of the equations, then you need to set the defaults on your operating system so that files ending in *.xml* are opened by *Firefox*, or an xml browser such as *Amaya*. Some of the links in this book open media files and so you will also need a media player or two (don't blame us, but not every file runs well on

every player). Mpeg/mpg movie players and a sound player are packaged with most operating systems. Free players are available from *Real, VideoLan (VLC)* and *Apple (Quicktime)*, with the movie players usually playing sound as well.

To attend our video lectures you will need a Flash player. The Adobe Flash player is free, yet it does no good on an iPad which deliberatively will not run Flash. (The Flash-Java-based video lecture modules run fine on recent Android tablets.) So we have made alternative mp4/Mpeg-4 versions of our lectures which look fine on the iPad, but lack the some interactivity and dynamic table of contents. (The mobile-device versions are not available on ComPADRE.) Yet technology moves on and Adobe has announced their plans to replace Flash with HTML5, which will be more appropriate for mobile devices. However, HTML5 is not yet implemented uniformly on different browsers and on different platforms, and so the reader will have to look forward to the video modules using HTML5 in the future versions of this text.

Permissions Needed Adobe and their Acrobat pdf reader is quite serious about security. Indeed, they have recently blocked our use of executeable codes from within the eBook, and this limitation is likely to remain until the Computer Science community adopts some protocols for so called "executeable papers". Even if you do not run codes in the book, you personally will have to grant permission before Acrobat will run multimedia components. You might also want to add this document to your *Trust* list. So keep an eye out for the requests, particularly as they may be buried behind an open window. Specifically, if the media is not loading when you click on the picture or link, look for a colored band asking for your permission. At some point in time you may be able to just set the security/Trust settings on your reader and not have to go through all this rigamarole, but the technology is not quite there yet.

1.1.1 The Python Computing Language

The codes in this version of *A Survey* employ *Python* 2.6. It is a high-level interpretive language that is easy (maybe the easiest) for beginners. Yet it is also the standard for the type of explorative and interactive computing that is now the hallmark of scientific research. Python is free, robust, universal, portable and available for most popular platforms. In fact, *Computing in Science & Engineering* [CiSE] has their March/April 2011 and their May/June 2007 issues focused on Scientific Python, and we recommend reading them to learn about the state of the art in things Python. For basic textbooks on Python, we recommend those by Langtangen [Lang 08, Lang 09]

The Python language is actually a core programming language plus a family of packages. Together they comprise an ecosystem for computing. These packages and combination of them extend the language to meet the specialized needs of science and engineering, and the specialized disciplines therein. For example, symbolic and mathematical computing (like that in Maple and Mathematica) can be done with the packages, *Sympx, mpmath* and *Sage*. *Sympx* does algebraic computing. *Mpmath* is a library for multiprecision floating-point arithmetic that also contains many high-level mathematical operations, while *Sage* combines many open-source packages to create a viable, free and in some ways superior alternative to Maple, Mathematica and Matlab. *In fact, if you do no mind some extra baggage, loading Sage into your computer is a good way to get Python and many useful packages all in one fell swoop.*

Because this book is mainly concerned with numerical computing we will use the *NumPy*, *matplotlib*, *SciPy* and *Visual* Python packages. *NumPy* brings high-level multidimensional arrays into Python, and is used as the basis for many of the numerical procedures in its

libraries. *Matplotlib* provides the tools for data visualization via 2-D and 3-D plots of publication quality, and is very much like Matlab (except it's free and runs for longer than one year before you have to renew its license). *SciPy* is a collection of tools that provides wrapper for many existing libraries in other languages, such as LAPACK [LAP 00] (linear algebra) and FFT (fast Fourier transforms). Finally, we use the *Visual* package (sometimes called "Vpython" when combined with Python) for its graphics tools and for its GUI *VIDLE*.

All of these packages are free and mainly open source (which means continual improvements). We give instructions on where and how to download them in Appendix B.

1.2 USING THE FEATURES OF THIS EBOOK

You can read this book just as you might a paper one. However, we recommend that you take advantage of its multimedia features to assist your learning. Although studies show that different people learn in different ways, most learners appear to benefit from experiencing multiple views of a subject, and this book encourages that. As in Web documents, this eBook contains *links* to objects signified by words in blue or icons. For example, clicking on 1.1 in "Figure 1.1" will jump you to the specified figure (actually to the caption, which is above the figure). Equations, tables, listings, pages and chapter sections have similar links throughout the text and in the Table of Contents and Index. To get back to the page from whence you came, the easiest way is to have Acrobat's Previous View (backarrow) button activated (View/Toolbars/More Tools/Previous View or Page Navigation Toolbar), and then to use it. Alternatively, on Windows you can Alt plus ←, or right click on the page you are viewing with your mouse and select Previous View. In either case, you should be duly transported[1].

If you are using *Acrobat Pro*, other options when you right click your mouse is Add Sticky Notes and Add Bookmark. Both are useful for personalizing the text, as is the Typewriter tool. Although links to other parts of this document should not illicit any complaints from Acrobat, if a link takes you outside of these pdf pages, say to a Web page or to a movie file, then Acrobat may ask your permission before proceeding. Furthermore, you may need to modify some of the Preferences in Acrobat relating to Trust so that it will be easier to open external links.

At the beginning of each chapter there is a table indicating which video lectures, slides, applets and animations are appropriate for that chapter (we have delayed that table in this chapter so we can explain it first). The names of the video lectures are links, for example, N-D Searching, where the accompanying small image of the lecturer , indicates, and usually links to, a lecture. These links open a Web page within a browser that contains multiple components in addition to the video. [For the iPad you want to open the mp4 version instead since the Web links use Flash, but the iPad does not.] There is a window showing the lecturer sitting for an office hour, another window with annotated slides synchronized to the lecture, a linked table of contents to that lecture, and video controls that let you stop and jump around. Separate links lead to the pdf versions of the slides that may be useful as a device for preview, review, or to have on a screen while watching the lectures. So, why don't you go ahead and try it! We suggest that you first read the text and review the slides before "attending" lecture, but feel free to find whatever combination works best for you.

Applets are small application programs written in Java that run through a Java-enabled Web browser. [Since Java does not run on iPad, these applets do not either; however they do run on PC.] The user does not deal with the Java code directly, but rather interacts with it via buttons and sliders on the screen. This means that the reader does not have to know anything at all about Java to run the applet (in fact, programming the graphical aspects of applets is so complicated that we do not recommend looking at the source unless you want to learn how to write applets). We use the applets to illustrate the results to be expected for projects in the book and to help understand some abstract concepts. Usually we just give the name of the Applet as a link, such as Chaotic Scattering, although sometimes we place the link in an icon, such as here Applet. Click on the link or the icon to initiate the applet, noting that it may take some time to load a browser and start the applet. Table C in the Appendix lists all of the applets available.

The Python codes are listed within shaded boxes with some formatting to improve readability. For example, look at Listing 1.1 (where 1.1 is a link to this listing). Note that we have

[1] On a Mac right clicking is accomplished by Control + click.

structured the codes so that a line is skipped before major elements like functions, and that indentations indicate structures in Python (where Java and C may use braces).

Listing 1.1 A sample code, **Walk3D.py** that produces a 3-D visualization of a random walk.

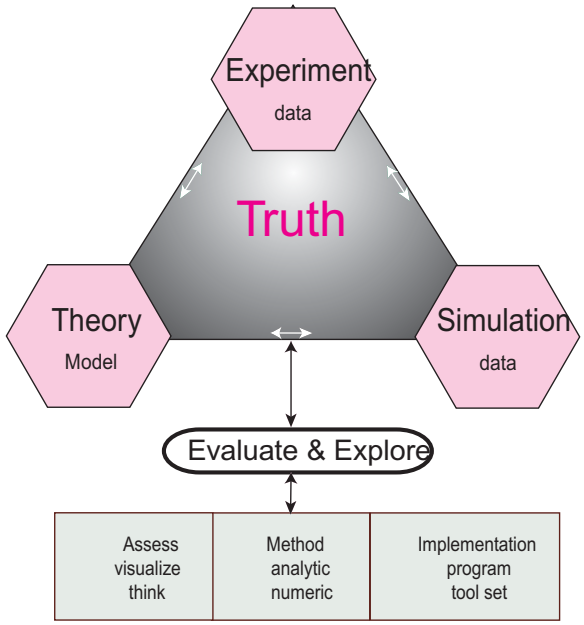
```
# Walk3D.py  3-D Random walk with graph

# Walk3D.py  3-D Random walk with 3-D graph

from visual import *
import random

random.seed(None)                    # Seed generator, None => system clock
jmax = 1000
xx =yy = zz =0.0                                          # Start at origin

graph1 = display(x=0,y=0,width = 600, height = 600, title = '3D Random Walk',
                 forward=(-0.6,-0.5,-1))
#  Curve, its parameters and labels
pts   = curve(x=list(range(0, 100)), radius=10.0,color=color.yellow)
xax   = curve(x=list(range(0,1500)), color=color.red, pos=[(0,0,0),(1500,0,0)], radius=10.)
yax   = curve(x=list(range(0,1500)), color=color.red, pos=[(0,0,0),(0,1500,0)], radius=10.)
zax   = curve(x=list(range(0,1500)), color=color.red, pos=[(0,0,0),(0,0,1500)], radius=10.)
xname = label( text = "X", pos = (1000, 150,0), box=0)
yname = label( text = "Y", pos = (-100,1000,0), box=0)
zname = label( text = "Z", pos = (100, 0,1000), box=0)

pts.x[0] =  pts.y[0] =  pts.z[0] =0            # Starting point
for i in range(1, 100):
    xx += (random.random() - 0.5)*2.           # -1 =< x =< 1
    yy += (random.random() - 0.5)*2.           # -1 =< y =< 1
    zz += (random.random() - 0.5)*2.           # -1 =< z =< 1
    pts.x[i] = 200*xx - 100
    pts.y[i] = 200*yy - 100
    pts.z[i] = 200*zz - 100
    rate(100)
print("This walk's distance R =", sqrt(xx*xx + yy*yy+zz*zz))
```

While these listings look great, their formatting makes them inappropriate for cutting, pasting and running. If you want to cut and paste a code, you can go to the **Codes** directory on the commercial version and copy codes from there, or you can click on the caption in the listing (usually in blue). That link opens a browser containing text version of the code that you can copy and paste into a Python IDE (integrated development environment), such as VIDLE, to run. In some places we also show an image of a python . Clicking on the icon also takes you to a Web page version of the code. If in the future a Python plugin is available for browsers, then it should be possible to run the code directly from the Web page.

Why don't you try running the **Walk3D.py** code given in Listing 1.1. It should result in a 3-D plot of a random walk in three dimensions. With details possibly depending upon your operating system, you can rotate this plot by selecting it with a click and then holding down your *left* mouse button as you move your mouse. You can zoom in and out of the plot by holding down your *right* mouse button (possibly also with your left one) and moving your mouse. The buttons on the bottom of the window present further options for viewing and saving the plot. Rotating a plot is important since it clues our brain into viewing it as a 3-D object.

As is true for the listing, the equations in this document may look great, but the pdf format does not convey any of their mathematical meaning. For instance, you cannot take a pdf equation and process it in a symbolic manipulation program such as Maple, Mathematica or Sage, or feed it to a reader that can speak the equation for the visually impaired. Having a *MathML* or *xml* version of the entire text (our original plan) would permit this, but very few people would have the software to read it. So we have compromised by just linking some key equations to their xml versions. For example, the equation below has the color of a link, and so clicking on it will open up a browser (which should be Mozilla Firefox or Amaya for a proper view) that displays the same equation but derived from an xml source file. (In some cases we

Figure 1.1 Simulation has been added to experiment and theory as a basic approach in the search for scientific truth. Although this book focuses on simulation, we present it as part of the scientific process.



use an xml icon to indicate the link.) Don't be afraid, try clicking on the equation below now!

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

Once you have the equation in the browser window, you can view the xml source and port its xml version to other programs (for example, with the *MathML[Import]* command in Maple, or the *Import* command in Mathematica).

Even though we try to be careful to define each term the first time we use it, we also have included as reference a Glossary in Appendix A. You will find various specialized words within this eBook linked to the glossary. These links are denoted by a word in blue and a speaker icon, for example: "An *algorithm* Clicking on the word will take you to the Glossary, while clicking on the speaker will read the definition to you (just close your eyes and enjoy the mellifluous tones of a matured Bronx accent).

In addition to the lectures that have video components (opt A) or are movie versions of a Flash production (opt B), we have also included animations into the text either as links that open in a movie player, or as movies actually embedded into the text as a figure. For example, clicking on the movie projector icon will open a movie player with an animation of a double pendulum simulation. Try it and give permissions if requested. Here in Figure 1.2 is an example, in reduced size, from Chapter 12 of actually imbedding a movie into the text (read the next warning that follows before trying to load and then look for "Loading, (permission may be pending?))". *Note, to have your pdf reader launch a player embedded within text, you may have to tell your reader that it has your permission to run the reader. To do that you need to be in a viewing mode in which you can see popups, buttons and messages, and then give permission to load the player.* Once you have given permission for one movie, you should not have to repeat that for other movies.

The text also uses various symbols and fonts to help clarify the type of material being dealt with. These include:

Figure 1.2 An embedded movie of a double pendulum simulation. (You may have to reduce reader's size to give permission to load.)

Loading DoublePend.mpg

⊙	Optional material
▌ at line's end	End of exercise or problem
`Monospace font`	Words as they would appear on a computer screen
Italic font	Note to reader at beginning of a chapter saying what's to follow
Sans serif font	Program commands from drop-down menus

We also indicate a user–computer dialog via three different fonts on a line:

`Monospace computer's output >` **`Bold monospace user's command`** Comments

VIDEO LECTURES, APPLETS & ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Intro to Computational Physics	pdf	1.2	Computing Basics	pdf	1.3-1.4
Number Representations	pdf	1.5	IEEE Floating Point	pdf	1.5–1.6
Machine Precision (IEEE)	pdf	1.53–1.6	Seminar: CP Education	pdf	–

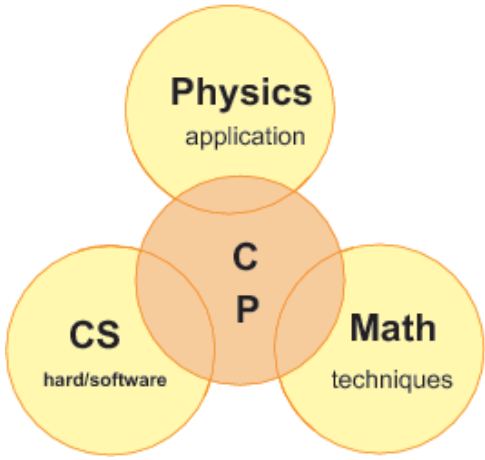
1.3 COMPUTATIONAL PHYSICS AND COMPUTATIONAL SCIENCE

This book adopts the view that CP is a subfield of computational science. This means that CP is a multidisciplinary subject combining aspects of physics, applied mathematics, and computer science (CS) (Figure 1.3), with the aim of solving realistic physics problems. Other computational sciences replace the physics with biology, chemistry, engineering, and so on, and together face grand challenge problems such as

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 systems	Oil and gas recovery
Ocean science	Vehicle signature	Undersea surveillance

Whereas 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 algorithms that computational scientists use. As much as we too find math and computer science interesting for their own sakes, our focus is on solving physical problems; we need to understand the CS and math tools well enough to be able to solve our problems correctly. As CP has

Figure 1.3 A representation of the multidisciplinary nature of computational physics both as an overlap of physics, applied mathematics, and computer science and as a bridge among them. Accordingly this book does not focus on programming, algorithms or the physics.



matured, we have come to realize that it is more than the overlap of physics, computer science, and mathematics (Figure 1.3). It is also a bridge among them (the central region in Figure 1.3) containing core elements of it own, such as computational tools and methods. To us, CP's commonality of tools and a problem-solving mindset draws it toward the other computational sciences and away from the subspecialization found in so much of physics.

In order to emphasize our computational science focus, to the extent possible, we present 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 (Figure 1.1 left). Traditionally, physics employs both experimental and theoretical approaches to discover scientific truth (Figure 1.1 right). Being able to transform a theory into an algorithm requires significant theoretical insight, detailed physical and mathematical understanding, and a mastery of the art of programming. The actual debugging, testing, and organization of scientific programs is analogous to experimentation, with the numerical simulations of nature being 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 believe that the scientific paradigm has been extended to include simulation as an additional dimension (Figure 1.1).

1.4 VIEWING THE SUBJECTS TO BE COVERED

As we indicated in the preface, we believe that the ideal eBook text should let the reader customize the selection and order of topics to best meet the reader's background and needs. Although this book is not quite there yet, we take a step in that direction with the *concept maps* [VUE] shown in Figures 1.4 and 1.5, with an overview of the entire text presented as a concept map in Figure 1.6. These maps display the contents of a document as a *knowledge field*, roughly analogous to an electric field, with field lines connecting the various areas and subareas.

Each of these concept map figures is linked to a stand alone map, which is not only easier to read, but also has its concepts linked to sections of the text. Technically, this means that these stand alone maps are *content* maps. This in turn gives the reader the option to use

Figure 1.4 A concept map of the **Basic Computational Science Topics** covered in this book (click on figure for larger and linked view). The green hexagons indicate major subjects, the pink ellipses indicate subareas, and the blue triangles indicate subjects contained on another map. The round bubbles are used to group general areas. In the stand alone "content" map linked to this concept map, each concept is linked to corresponding content within the text.

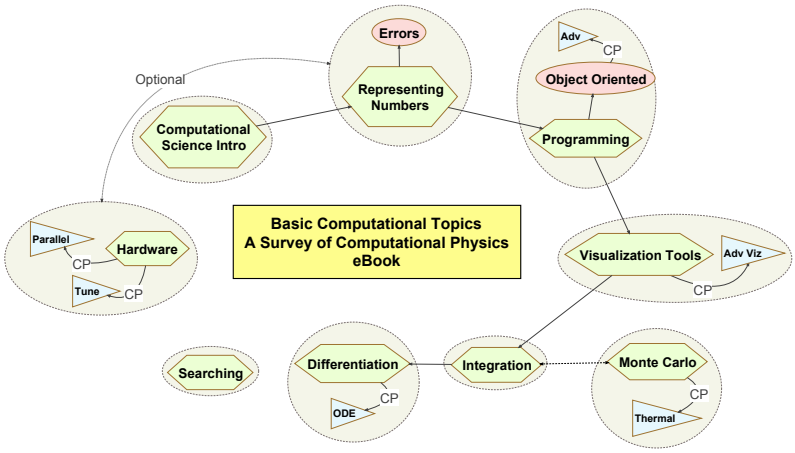


Figure 1.5 A concept map of the **Computational Physics Topics** covered in this book (click on it for a larger and linked view). The green hexagons indicate major subjects, the pink ellipses indicate subareas, and the blue triangles indicate subjects contained on another map. The round bubbles are used to group general areas. In the stand alone "content" map linked to this concept map, each concept is linked to corresponding content within the text.

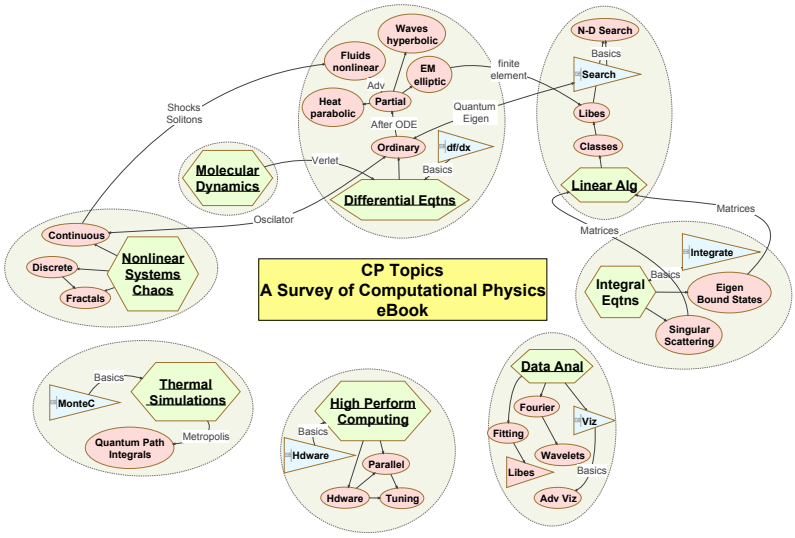
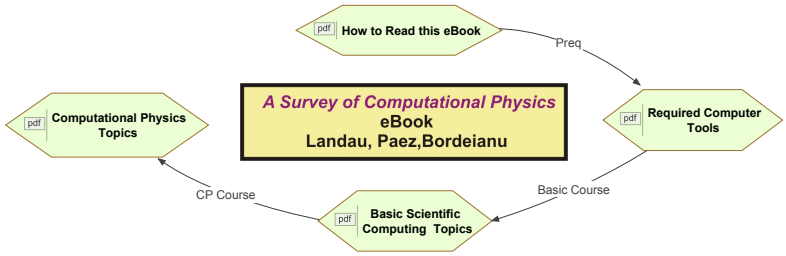


Figure 1.6 A concept map overview of this entire eBook (click on it for larger and linked view). Each green angular box is linked to an appropriate concept map.



the content map and its links as a guide to read the text[2].

You will find in the Basic Topics concept map in Figure 1.4, the elementary software tools needed for computational science. Notice how the general areas (bubbles) are linked together with arrows indicating the recommended order in which the subjects should be followed. However, the Hardware bubble's link is labeled "Optional" because although it fits in logically at the beginning of a course, some readers or instructors may prefer to cover it later with High Performance Computing as part of the CP topics. Likewise, the "Searching" bubble sits like an island unconnected to the mainland. It is an essential topic that fits in well to an introductory course, but does not require a specific logical placement. You as a reader, of course, are free to construct your own path through the materials.

In contrast to the Basic Topics Map, the CP Topics map in Figure 1.5 has many more topics, but fewer directed field lines among them. This displays the greater freedom in which these topics may be approached, although the essential prerequisites from the basic topics are indicated. Notice also in Figure 1.5 that much of the CP we present appears to involve differential equations, both ordinary and partial. In practise, many of those applications employ linear algebra libraries, for which links are indicated, as well as scientific libraries, also include under linear algebra. In contrast, this text is rather unique in also covering some topics in integral equations and their applications in physics. While not seen in many physics courses, it is an important topic in research and applications and we suggest that at least the Eigen Bound State problem be approached.

A more traditional way to view the materials in this text is in terms of its use in courses. In our classes [CPUG] we have used approximately the first third of the text, with its emphasis on computing tools, for a course in scientific computing (after students have acquired familiarity with a compiled language). Typical topics covered in the 10 weeks of such a course are given in Table 1.1. Some options are indicated in the caption, and, depending upon the background of the students, other topics may be included or substituted. The latter two-thirds of the text includes more physics, and, indeed, we use it for a two-quarter (20-week) course in computational physics. Typical topics covered for each term are given in Table 1.4. What with many of the topics being research level, we suspect that these materials can easily be used for a full year's course as well.

For these materials to contribute to a successful learning experience, we assume that the reader will work through the problem at the beginning of each chapter or unit. This entails studying the text, writing, debugging and running programs, visualizing the results, and then

[2] Although many people find much value in these maps, others find them confusing, and especially at first. In any case, it may take some study before the reader "sees" the concept connections.

Table 1.1 **Topics for one quarter (10 Weeks) of a scientific computing course.** Units are indicated by I, II, and III, and the visualization, here spread out into several laboratory periods, can be completed in one. Options: week 3 on visualization; postpone matrix computing; postpone hardware basics; devote a week to OOP; include hardware basics in week 2.

Week	*Topics*	*Chapter*
1	OS tools, limits	1, (4)
2	Errors, visualization	2, 3
3	Monte Carlo, visualization	5, 3
4	Integration, visualization	6, (3)
5	Derivatives,searching	7I, II
6	Matrices, N-D search	8I
7	Data fitting	8II
8	ODE oscillations	9I
9	ODE eigenvalues	9II
10	Hardware basics	14I, III$\odot$

expressing in words what has been done and what can be concluded. Further exploring is encouraged. Although we recognize that programming is a valuable skill for scientists, we also know that it is incredibly exacting and time-consuming. In order to lighten the workload somewhat, we provide "bare bones" programs in the text and via hyperlinks. We recommend that these be used as guides for the reader's own programs or tested and extended to solve the problem at hand. As part of this approach we suggest that the learner write up a mini lab report for each problem containing

Equations solved	Numerical method	Code listing
Visualization	Discussion	Critique

The report should be an executive summary of the type given to a boss or manager; make it clear that you understand the materials but do not waste everyone's time.

One of the most rewarding uses of computers is *visualizing* and analyzing the results of calculations with 2-D and 3-D plots, with color, and with animation. This assists in the debugging process, hastens the development of physical and mathematical intuition, and increases the enjoyment of the work. It is essential that you learn to use visualization tools as soon as possible, and so in Chapter 3, "Visualization Tools," we describe a number of free visualization tools that we use and recommend. We include many color figures showing visualizations (the printed text, unfortunately, permits only gray scale).

1.5 MAKING COMPUTERS OBEY; LANGUAGES (THEORY)

> Computers are incredibly fast, accurate, and stupid; humans are incredibly slow, inaccurate, and brilliant; together they are powerful beyond imagination.
>
> — *Albert Einstein*

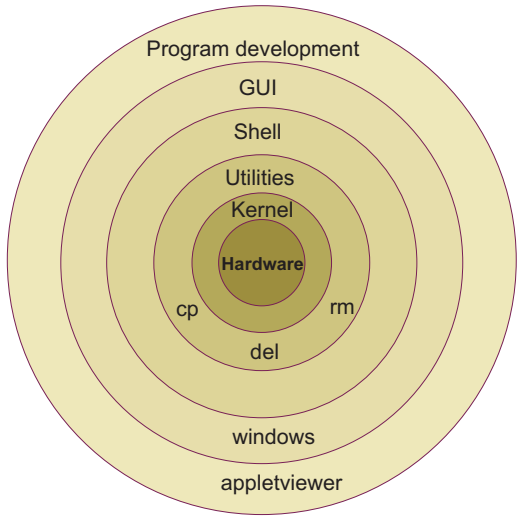
As anthropomorphic as your view of your computer may be, keep in mind that computers always do exactly as they are told. This means that you must tell them exactly everything they have to do. Of course the programs you run may have such convoluted logic that you may not

Table 1.2 **Topics for two quarters (20 Weeks) of a computational physics course.** Units are indicated by I, II, and III. Options: include OpenDX visualization (§3.5, Appendix C); include multiresolution analysis (11II); include FFT (10III) in place of wavelets; include FFT (10III) in place of parallel computing; substitute Feynman path integrals (15III) for integral equations (20); add several weeks on CFD (hard); substitute coupled predator-prey (12III) for heat PDE (17III); include quantum wave packets (18II) in place of CFD; include finite element method (17II) in place of heat PDE.

Computational Physics I		
Week	*Topics*	*Chapter*
1	Nonlinear ODEs	9I, II
2	Chaotic scattering	9III
3	Fourier analysis, filters	10I, II
4	Wavelet analysis	11I
5	Nonlinear maps	12I
6	Chaotic/double pendulum	12II
7	Project completion	12I, II
8	Fractals, growth	13
9	Parallel computing, MPI	14II
10	More parallel computing	14III

Computational Physics II		
Week	*Topics*	*Chapter*
1	Ising model, Metropolis algorithm	15I
2	Molecular dynamics	16
3	Project completions	—
4	Laplace and Poisson PDEs	17I
5	Heat PDE	17III
6	Waves, catenary, friction	18I
7	Shocks and solitons	19I
8	Fluid dynamics	19 II
9	Quantum integral equations	20I (II)
10	Feynman path integration	15III

Figure 1.7 A schematic view of a computer's kernel and shells. The hardware is in the center surrounded by increasing higher-level software.



have the endurance to figure out the details of what you have told the computer to do, but it is always possible in principle. So your first **problem** is to obtain enough understanding so that you feel well enough in control, no matter how illusionary, to figure out what the computer is doing.

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* [3] 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. Very few computational scientists talk to computers in a language computers can understand. When writing and running programs, we usually communicate to the computer through *shells*, in *high-level languages* (Python, Java, Fortran, C), or through *problem-solving environments* (Maple, Mathematica, and Matlab). Eventually these commands or programs are translated into the basic machine language that the hardware understands.

A *shell* is a *command-line interpreter*, that is, a set of small programs run by a computer that 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 (OS) (Figure 1.5), within which lies a *kernel* of elementary operations. (The user seldom interacts directly with the kernel, except possibly when installing programs or when building an operating system from scratch.) It is the job of the shell to run programs, compilers, and utilities that do things like copying files. There can be different types of shells on a single computer or multiple copies of the same shell running at the same time.

Operating systems 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;

[3] The Beginner's All-Purpose Symbolic Instruction Code (BASIC) programming language of the original PCs should not be confused with basic machine language.

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 can 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 translate it into a fast code. The translated statements form an *object* or compiled 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 compilers to 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 Python and Java languages are a mix of the two. When you first compile your program, Python interprets it into an intermediate, universal *byte code* which gets stored as a PYC (or PYO) file. This file can be transported to and used on other computers, although not with different versions of Python. Then, when you run your program, Python recompiles the byte code into a machine-specific compiled code, which is faster than starting from source code.

1.6 PROGRAMMING WARMUP

Before we go on to serious work, we want to ensure that your local computer is 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 use any specific language, write that program in pseudocode that can be converted to your favorite language later. The first program tells the computer:[4]

```
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
```

[4]Comments placed in the field to the right are for your information and *not* for the computer to act upon.

This is a better program, and so let's see how to implement it in Python (other language versions are on the CD from the paper text). In Listing 1.2 we give a Python version of our Area program. This is a simple program that outputs to the screen, with its input built into the program.

Listing 1.2 **Area.py** outputs to the screen, with its input built into the program.

```
# Area.py: Area of a circle, simple program

from math import pi
from sys import version
if int(version[0]) > 2:                # raw_input deprecated in Python 3
    raw_input=input
modelN = 1
radius = 1.
circum = 2. *pi*radius
area = radius * radius *pi

print ('Program number = ', modelN)
print ('Radius = ', radius)
print ('Circumference = ', circum)
print ('Area = ', area)

# OUTPUT:
'''
Program number =  1
Radius =  1.0
Circumference =  6.28318530718
Area = 3.14159265359
'''
print("press a key to finish")
s = raw_input()
```

1.6.1 Structured Program Design

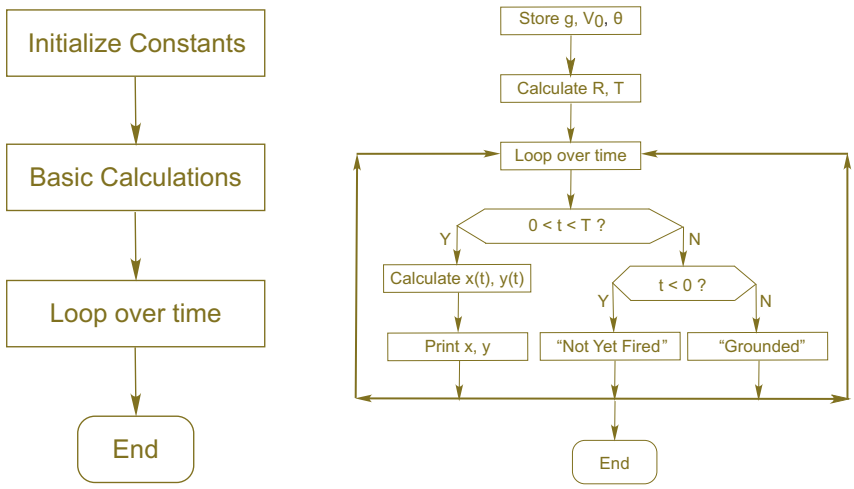
Programming is a written art that blends elements of science, mathematics, and computer science into a set of instructions that permit a computer to accomplish a desired task. Now that we are getting into the program-writing business, you will benefit from understanding the overall structures that you should be building into your programs, in addition to the grammar of a computer language. As with other arts, we suggest that until you know better, you follow some simple rules. A good program should

Give the correct answers.
Be clear and easy to read, with the action of each part easy to analyze.
Document itself for the sake of readers and the programmer.
Be easy to use.
Be easy to modify and robust enough to keep giving correct answers after modification.
Be passed on to others to use and develop further.

One attraction of object-oriented programming (Chapter 4; "Object-Oriented Programs: Impedance & Batons") is that it enforces these rules automatically. An elementary way to make any program clearer is to *structure* it 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 fact that compilers ignore these visual clues, human readers are aided by having a program that not only looks good but also has its different logical parts visually evident. Even though the space limitations of a printed page keep us from inserting as many blank lines as we would prefer, we recommend that you do as we say and not as we do!

In Figure 1.8 we present basic and detailed *flowcharts* that illustrate a possible program for computing projectile motion. A flowchart is not meant to be a detailed description of a

Figure 1.8 A flowchart illustrating a program to compute projectile motion. On the left are the basic components of the program, and on the right are some of its details. When writing a program, first map out the basic components, then decide upon the structures, and finally fill in the details. This is called *top-down programming*.



program but instead is a graphical aid to help visualize its logical flow. As such, it is independent of a specific computer language and is useful for developing and understanding the basic structure of a program. We recommend that you draw a flowchart or (second best) write a pseudocode before you write a program. *Pseudocode* is like a text version of a flowchart that leaves out details and instead focuses on the logic and structures:

```
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
```

1.6.2 Shells, Editors, and Execution

i) To gain some experience with your computer system, use an editor to enter the program **`Area.py`** that computes the area of a circle (yes, we know you can copy and past it, but you may need some exercise before getting to work). Then write your file to disk by saving it in your home (personal) directory (we advise having a separate subdirectory for each week). *Note:* For those who are familiar with Python, you may want to enter the program **`AreaFormatted.py`** instead (described in a later section) that uses commands that produce formatted output.

ii) Compile and execute the appropriate version of **`Area.py`**.

iii) Change the program so that it computes the volume $\frac{4}{3}\pi r^3$ of a sphere. Then write your file to disk by saving it in your home (personal) directory and giving it the name **`AreaMod.py`**.

iv) Compile and execute **`AreaMod`** (remember that the file name and class name must agree).

v) Check that your changes are correct by running a number of trial cases. A good input datum is $r = 1$ because then $A = \pi$. Then try $r = 10$.

vi) Experiment with your program. For example, see what happens if you leave out decimal points in the assignment statement for r, if you assign r equal to a blank, or if you assign a letter to r. Remember, it is unlikely that you will "break" anything by making a mistake, and it is good to see how the computer responds when under stress.

vii) Revise **`Area.py`** 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.

viii) See what happens when the data type used for output does not match the type of data in the file (e.g., data are **`double`**s, but read in as **`int`**s).

ix) Revise **`AreaMod`** so that it uses a main method (which does the input and output) and a separate method for the calculation. Check that you obtain the same answers as before.

Listing 1.3 **AreaFormatted.py** uses output statements to control the format of the outputted numbers.

```
# AreaFormatted: example use of formated output, keyboard inputm file I/O

from numpy import *
from sys import version
if int(version[0])>2:                    # raw_input deprecated in Python 3
    raw_input=input
name = raw_input( 'Key in your name: ')            # raw_input for strings
print("Hi ",name)
radius = eval(raw_input('Enter a radius: '))        # For numerical values
print('you entered radius= %8.5f'%radius)                # formatted output
name = raw_input('Key in another name: ')          # raw_input for strings
radius = eval(raw_input('Enter a radius: '))
print('Enter new name and r in file Name.dat')
inpfile = open('Name.dat','r')                    # Read from file Name.dat
for line in inpfile:
    line = line.split()                         # Splits components of line
    name = line[0]                                # First entry in the list
    print(" Hi  %10s" %(name))                     # print Hi + first entry
    r = float(line[1])                            # convert string to float
    print(" r = %13.5f" %(r))                    # convert to float & print
inpfile.close()
A = math.pi*r**2
print("Done, look in A.dat\n")
outfile = open('A.dat','w')
outfile.write( 'r=  %13.5f\n'%(r))
outfile.write('A =  %13.5f\n'%(A))
outfile.close()
print('r = %13.5f'%(r))                                    # Screen output
print('A = %13.5f'%(A))
print('Now example of integer input ')
age=int(eval(raw_input ('Now key in your age as an integer:  ')))
print("age: %4d  years old,  you don't look it!\n"%(age))
print("Enter and return a character to finish")
s = raw_input()
```

1.6.3 Formatted I/O in Python

The simplest I/O with Python is outputting to the screen with the **`print`** command (as seen in **`Area.py`**, Listing 1.2), and inputting from the keyboard with the **`input`** command (as seen in **`AreaFormatted.py`**, Listing 1.3). We also see in **`AreaFormatted.py`** that we can input strings (literal numbers and letters) by either enclosing the string in quotes (single or double), or using the **`raw_input`** command without quotes. To print a string with **`print`**, place the string in quotes.

The simplest output prints the value of a float just by giving its name:

```
print 'eps = ', eps
```

Output float in default format

This uses Python's default format, which tends to vary depending on the precision of the number being printed. As an alternative, you can control the format of your output. For floats you need to specify two things. First, how many digits (places) after the decimal point is desired, and second, how many spaces overall should be used for the number:

```
print("x=%6.3f, Pi=%9.6f, Age=%d \n") % (x, math.pi, age)
print "x=%6.3f, %(x), "Pi=%9.6f," %(math.pi), "Age=%d "%(age)," \n"
x = 12.345, Pi = 3.141593, Age=39
```

Output from either

Here the **%6.3f** formats a float (which is a double in Python) to be printed in fixed-point notation (the **f**) with 3 places after the decimal point and with 6 places overall (1 place for the decimal point, 1 for the sign, 1 for the digit before the decimal point, and 3 for the decimal). The directive **%9.6f** has 6 digits after the decimal place and 9 overall.

To print an integer we need specify only the total number of digits (there is no decimal part), and we do that with the **%d** (d for digits) format. The **%** symbol in these output formats indicates a conversion from the computer's internal format to that used for output. Notice in Listing 1.3 how we read from the keyboard, as well as from a file, and output to both screen and file. Beware that if you do not create the file **Name.dat**, the program will issue ("throw") an error message of the sort:

```
IOError: [Errno 2] No such file or directory: 'Name.dat'
```

Note that we have also use a **%n** directive here to indicate a new line. Other directives, some of which are demonstrated in **Directives.py** in Listing 1.4 (and some of which like backspace may not yet work right) are:

\"	double quote	\0NNN	octal NNN	\\	backslash
\a	alert (bell)	\b	backspace	\c	no more output
\f	form feed	\n	new line	\r	carriage ret
\t	horizontal tab	\v	vertical tab	%%	a single %

Listing 1.4 The program **Directives.py** illustrates formatting via directives and escape characters.

```
# Directives.py illustrates escape and formatting characters
import sys
print("hello \n")
print("\t it's me")                          # tabulator
b = 73
print("decimal 73 as integer b = %d "%(b))  # for integer
print("as octal b = %o"%(b))                # octal
print("as hexadecimal b = %x "%(b))         # works hexadecimal
print("learn \"Python\" ")                  # use of double quote symbol
print("shows a backslash \\")               # use of \\
print('use of single \' quotes \' ')        # print single quotes
```

1.6.4 I/O Redirection

Most programming environments assume that the standard (default) input is *from* the keyboard and the standard output is *to* the computer screen. But you can change that. A simple way to read from or write to a file is via *command-line redirection* from within the shell in which you are running your program. (This must be a Unix or Windows shell and not the Python shell since the former interpret system commands, while the latter interprets individual Python statements.) For example, here we run the program **Directives.py** from a system shell and redirect the output from the default screen to the file **outfile.dat**:

```
C:\PythonCodes> AreaFormatted.py > outfile.dat
```

Redirect standard output

Likewise, we can redirect input from the default keyboard to the file **`infile.dat`**:

```
C:PythonCodes> AreaFormatted.py < infile.dat
```
Redirect standard input

Or you can put them both together for complete redirection:

```
C:\PythonCodes> AreaFormatted.py < infile.dat > outfile.dat
```

In Listing 1.4 we show a clever way to redirect the standard output by temporarily redefining the operating system's standard output to a file. Run it and see!

Listing 1.5 The program **CommandLineArgs.py** demonstrates how arguments can be transferred to a main program via the command line.

```
# CommandLineArgs.py: Accepts 3 or 4 arguments from command line, e.g.:
#    java CommandLineArgs anInt aDouble [aString].
# [aString] = optional filename.
# Written by Zlatko Dimcovic in Java
import sys

intParam = 0                                          # Other values OK
doubleParam = 0.0;                              # Defaults, args optional
filename = "baseName"
print( 'there are %d arguments ' % len(sys.argv))
print( sys.argv)
if len(sys.argv) == 3 or len(sys.argv) == 4:          # Demand 2 or 3 args
    intParam = int(sys.argv[1])
    doubleParam =float(sys.argv[2])
    if len(sys.argv) == 4:
        filename = sys.argv[3]                     # 4th argument filename
    else:
        filename ="_i"+ sys.argv[1]+"_d" + sys.argv[2] +".dat"
    # print intParam, doubleParam, filename
else:
    print( "\n\t Usage: java CmdLineArgs intParam doubleParam [file]")
    print( "\t 1st arg must be int, 2nd double (or int),")

print ('Input arguments: intParam (1st) = ',intParam,
  'doubleParam (2nd) = ', doubleParam)
if len(sys.argv) == 4:
    print( "String input: ", filename)
elif len(sys.argv) == 3:
    print( "No file, use", filename)
else:
    print( "\n\t ERROR ! len(sys.argv) must be 3 or 4 \n")
```

1.6.5 Command-Line Input

Although we tend not to in our sample programs, you can also input data to your programs from the command line or shell by specifying values for the argument of your main program (or method). (Again we remind you that this would be the Unix or Windows shell which accepts systems commands, and not the Python shell, which accepts only individual Python statements.) Since main methods are still methods, they can take arguments (a *parameter list*) and return values. Normally one would execute a Python program from a shell just by entering the name of the file, for example, **`$ CommandLineArgs.py`**, or by double clicking on the file name. However you may add arguments that get sent to the main program by including them after the file name. As an example, the program **`CommandLineArgs.py`** in Listing 1.5 accepts and then uses arguments from the command line

```
D:\PythonCodes> CommandLineArgs.py 2 1.0 TempFile
```
Args: int, float, string

Here the main method is given an integer **`2`**, a float **`1.0`**, and a string **`TempFile`**, with the latter to be used as a file name. Note that this program is not shy about telling you what you should have done if you have forgotten to give it the arguments it expects. Further details are given within the program.

Listing 1.6 **FileCatchThrow.py** reads from the file and handles the I/O exception.

```
# FileCatchThrow.py:  throws and catches exception
#program with mistake to see action of exception
import sys
import math
r = 2
circum = 2.* math.pi* r                                  # Calculate circum
A = math.pi*r**2                                              # Calculate A
try:
    q = open("ThrowCatch.dat",'w')                  # Intentional error 'r'
                                                   # Replace r' to 'w', run
except  IOError:
    print( 'Cannot open file')
else:
     q.write("r = %9.6f, length = %9.6f, A= %9.6f "%(r, circum, A))
     q.close()
     print( 'Output in ThrowCatch.out')
     #catch(IOException ex){ex.printStackTrace(); }                # Catch
```

1.6.6 I/O Exceptions: FileCatchThrow.py

Exceptions occur when something goes wrong during the I/O process, such as not finding a file, trying to read past the end of a file, or interruptions of the I/O. Dealing with exceptions is important because it prevents the computer from freezing up and lets you know that there may be a problem that needs attention. If, for example, a file is not found, then Python will create an **`Exception`** object and pass it along ("throw exception") to the program that called the main method, and send the error message to you via the Python Shell. You gain control over these exceptions by including the phrase:

```
except IOError
```

In Listing 1.6 we give an example of the use of the **`except IOError`** construct and of how a program can deal with (*catches*) a thrown exception object. We see that the program has a **`try`** structure, within which a file is opened, and then a **`catch`** structure, where the program proceeds if an exception is thrown. The error is thus caught by the **`catch`** statement which prints a statement to that effect.

1.6.7 Automatic Code Documentation ⊙

There is a freely available package *epydoc* that automatically generates fancy-looking html or pdf documentation of your Python program (have we mentioned that documentation is very important for scientific computing and very ignored?). You can find *epydoc* at

```
http://epydoc.sourceforge.net/
```

where you need to pick the version appropriate for your operating system, download it, and then install it. To use *epydoc* you include comments in your code containing the necessary commands, and then process the comments with the separate epydoc application.

As an example, we use epydoc with the program **`OOPbeats.py`**. For organizations's sake, we create a new subdirectory **`Docs`** in the directory containing our Python codes, and place the output there. In this way we do not clutter our code directory with html files and such. To run epydoc, you must go to the directory on your computer where Python is stored and start up the graphical user interface (GUI) for epydoc. On our Windows machine this is

```
C:\Python25\Lib\site-packages\epydoc\gui.py
```

Go to the equivalent directory on your computer and execute **`gui.py`** by double clicking on it.

Figure 1.9 A part of the automatic code documentation produced by the Python utility *epydoc* when used on the program `OOPbeats.py`.

Home Trees Indices Help

Module OOPbeats :: Class OOPbeats [hide private] [frames] | no frames]

Class OOPbeats

source code

Instance Methods [hide private]

	`__init__(self, Ampl, freq1, freq2)`	source code
	`summaves(self, mytitle, myxtitle, myytitle, xma, xmi, yma, ymi)`	source code

Instance Variables [hide private]

	`k2` "Defines title of the window, x and y labels and max and min" values for the two axis

Home Trees Indices Help

Generated by Epydoc 3.0.1 on Tue Jan 13 15:24:42 2009 http://epydoc.sourceforge.net

(See actual code)

Now that you have a nice window, select File/New Project and then use the Browse menu to select the file you wish to document (**`OOPbeats.py`**). A typical output is shown in Figure 1.9.

1.7 COMPUTER NUMBER REPRESENTATIONS (THEORY)

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 arising from this representation. 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 binary integers (*bits*) 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 users. Consequently, binary strings are converted to *octal*, *decimal*, or *hexadecimal* numbers before the 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, the process leads to a decrease in precision.

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

$$1\,\text{byte} \equiv 1\,\text{B} \stackrel{\text{def}}{=} 8\,\text{bits}.$$

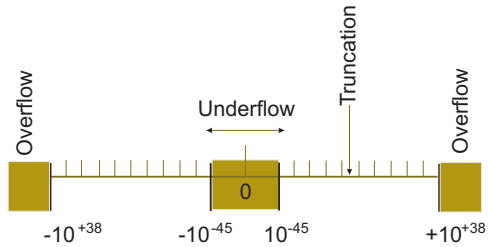
Memory and storage sizes are measured in bytes, kilobytes, megabytes, gigabytes, terabytes, and petabytes (10^{15}). Some care should be taken here by those who chose to compute sizes in detail because K does not always mean 1000:

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

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

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

Figure 1.10 The limits of single-precision floating-point numbers and the consequences of exceeding these limits. The hash marks represent the values of numbers that can be stored; storing a number in between these values leads to truncation error. The shaded areas correspond to over- and underflow.



Conveniently, 1 byte is also the amount of memory needed to store a single letter like "a", which adds up to a typical printed page requiring ∼3 kB.

The memory chips in some older personal computers used 8-bit words. This meant that the maximum integer was $2^7 = 128$ (7 because 1 bit is used for the sign). Trying to store a number larger than the hardware or software was designed for (*overflow*) was common on these machines; it was sometimes accompanied by an informative error message and sometimes not. Using 64 bits permits integers in the range 1–$2^{63} \simeq 10^{19}$. While at first this may seem like a large range, 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}.

1.7.1 IEEE Floating-Point Numbers

Real numbers are represented on computers in either *fixed-point* or *floating-point* notation. *Fixed-point notation* can be used for numbers with a fixed number of places beyond the decimal point (radix) or for integers. It has the advantages of being able to use *two's complement* arithmetic and being able to store integers exactly.[5] In the fixed-point representation with N bits and with a two's complement format, a number is represented as

$$N_{\text{fix}} = \text{sign} \times (\alpha_n 2^n + \alpha_{n-1} 2^{n-1} + \cdots + \alpha_0 2^0 + \cdots + \alpha_{-m} 2^{-m}), \tag{1.1}$$

where $n + m = N - 2$. That is, 1 bit is used to store the sign, with the remaining $(N - 1)$ bits used to store the α_i values (the powers of 2 are understood). The particular values for N, m, and n are machine-dependent. Integers are typically 4 bytes (32 bits) in length and in the range

$$-2147483648 \leq \text{4-B integer} \leq 2147483647.$$

An advantage of the representation (1.1) 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 (1.1)]. The corresponding disadvantage is that *small* numbers (those for which the first string of α 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 $(64\,\text{b} = 8\,\text{B})$. 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

[5] The *two's complement* of a binary number is the value obtained by subtracting the number from 2^N for an N-bit representation. Because this system represents negative numbers by the two's complement of the absolute value of the number, additions and subtractions can be made without the need to work with the sign of the number.

Table 1.3 The IEEE 754 Standard for Primitive Data Types.

Name	*Type*	*Bits*	*Bytes*	*Range*
`boolean`	Logical	1	$\frac{1}{8}$	**`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	$\pm 4.94065645841246544 \times 10^{-324} \leftrightarrow \pm 1.7976931348623157 \times 10^{+308}$

$c = +2.99792458 \times 10^{+8}$ m/s in scientific notation and $+0.299792458 \times 10^{+9}$ or 0.299795498 E09 m/s in engineering notation. In each of these cases, the number in front is called the *mantissa* and contains nine *significant figures*. The power to which 10 is raised is called the *exponent*, with the plus sign included as a reminder that these numbers 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* (the hash marks in Figure 1.10), is much smaller than the set of real numbers. In particular, machine numbers have a maximum and a minimum (the shading in Figure 1.10). 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. In the latter case, the software and hardware may be set up so that underflows are set to zero without your even being told. In contrast, overflows usually halt execution.

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. In fact, in the past each computer operating system and each computer language contained its own standards for floating-point numbers. Different standards meant that the same program running correctly on different computers could give different results. Even though 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 Table 1.3. In addition, when computers and software adhere to this standard, and most do now, 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 be strictly followed for your test cases. After you know that the code is okay, you may want to run with whatever gives the greatest speed and precision.

Table 1.4 Representation Scheme for Normal and Abnormal 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	$e = 0, \quad f \neq 0$	$(-1)^s \times 2^{-126} \times 0.f$
Signed zero (± 0)	$e = 0, \quad f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, \quad e = 255, \quad f = 0$	`+INF`
$-\infty$	$s = 1, \quad e = 255, \quad f = 0$	`-INF`
Not a number	$s = u, \quad e = 255, \quad f \neq 0$	`NaN`

There are actually a number of components in the IEEE standard, and different computer or chip manufacturers may adhere to only some of them. Furthermore, Python may not follow all as it develops, but probably will in time. Normally a floating-point number x is stored as

$$x_{\text{float}} = (-1)^s \times 1.f \times 2^{e-\text{bias}}, \tag{1.2}$$

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 a positive or a negative sign. Eight bits are used to stored 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 (Table 1.4). *Normal numbers* have $0 < e < 255$, and with them the convention is to assume that the mantissa's first bit is a 1, so only the fractional part f after the *binary point* is stored. The representations for *subnormal numbers* and the special cases are given in Table 1.4.

Note that the values $\pm$`INF` and `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 signals to the computer and to you that something has gone awry and that the calculation should probably stop until you straighten things out. In contrast, the value -0 can be used in a calculation with no harm. Some languages may set unassigned variables to -0 as a hint that they have yet to be assigned, though it is best not to count on that!

Because the uncertainty (error) is only in the mantissa and not the exponent, 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 is changed before the final number is stored. To repeat, for normal cases, the actual mantissa ($1.f$ in binary notation) contains an implied 1 preceding the binary point.

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

$$p = e - \text{bias}. \tag{1.3}$$

Examples of IEEE Representations

There are two basic, IEEE 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*. (In Python, however, floats are double precision.) Singles occupy 32 bits overall, with 1 bit 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 1 bit for the sign, 10 bits for the exponent, and 53 bits 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 Table 1.3. (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 the scheme in practise consider the 32-bit representation (1.2):

	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 8 bits are used to store the exponent e and since $2^8 = 256$, e has the range

$$0 \le e \le 255.$$

The values $e = 0$ and 255 are special cases. With $\text{bias} = 127_{10}$, the full exponent

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

and, as indicated in Table 1.3, for singles has the range

$$-126 \le p \le 127.$$

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 bit is

$$\text{Subnormal numbers} = (-1)^s \times 0.f \times 2^{e-126}. \tag{1.4}$$

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

$$\text{Mantissa} = 1.f = 1 + m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \cdots + m_0 \times 2^{-23}, \tag{1.5}$$

with $0.f$ used for subnormal numbers. The special $e = 0$ representations used to store ± 0 and $\pm\infty$ are given in Table 1.4.

To see how this works in practice (Figure 1.10), the largest positive normal floating-point number possible for a 32-bit machine has the maximum value $e = 254$ (the value 255 being reserved) and the maximum value for f:

$$\begin{aligned} X_{\max} &= 01111\ 1110\ 1111\ 1111\ 1111\ 1111\ 1111\ 111 \\ &= (0)(1111\ 1110)(1111\ 1111\ 1111\ 1111\ 1111\ 111), \end{aligned} \tag{1.6}$$

where we have grouped the bits for clarity. After putting all the pieces together, we obtain the value shown in Table 1.3:

$$
\begin{aligned}
&s = 0, \qquad e = 1111\ 1110 = 254, \qquad p = e - 127 = 127, \\
&f = 1.1111\ 1111\ 1111\ 1111\ 1111\ 111 = 1 + 0.5 + 0.25 + \cdots \simeq 2, \\
&\quad \Rightarrow (-1)^s \times 1.f \times 2^{p=e-127} \simeq 2 \times 2^{127} \simeq 3.4 \times 10^{38}.
\end{aligned} \tag{1.7}
$$

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

$$
\begin{aligned}
&s = 0, \qquad e = 0, \qquad p = e - 126 = -126 \\
&f = 0.0000\ 0000\ 0000\ 0000\ 0000\ 001 = 2^{-23} \\
&\quad \Rightarrow (-1)^s \times 0.f \times 2^{p=e-126} = 2^{-149} \simeq 1.4 \times 10^{-45}
\end{aligned} \tag{1.8}
$$

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

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

Doubles are stored as two 32-bit words, for a total of 64 bits (8 B). The sign occupies 1 bit, the exponent e, 11 bits, and the fractional mantissa, 52 bits: As we see here, the fields are stored

	s	e		f		f (cont.)	
Bit position	63	62	52	51	32	31	0

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,

$$\text{Bias} = 1111111111_2 = 1023_{10},$$

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

The bit patterns for doubles are given in Table 1.5, with the range and precision given in Table 1.3. To repeat, if you write a program with doubles, then 64 bits (8 bytes) will be used to store your floating-point numbers. Doubles have approximately 16 decimal places of precision (1 part in 2^{52}) and magnitudes in the range

$$4.9 \times 10^{-324} \ \leq \ \text{double precision} \ \leq \ 1.8 \times 10^{308}. \tag{1.9}$$

If a single-precision number x is larger than 2^{128}, a fault condition known as an *overflow* occurs (Figure 1.10). 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, not a number (NAN), 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 at least 64-bit (double-precision) floats*. And if you need double precision in one part of your calculation,

Table 1.5 Representation Scheme for IEEE Doubles

Number Name	*Values of s, e, and f*	*Value of Double*
Normal	$0 < e < 2047$	$(-1)^s \times 2^{e-1023} \times 1.f$
Subnormal	$e = 0, \quad f \neq 0$	$(-1)^s \times 2^{-1022} \times 0.f$
Signed zero	$e = 0, \quad f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, \quad e = 2047, \quad f = 0$	`+INF`
$-\infty$	$s = 1, \quad e = 2047, \quad f = 0$	`-INF`
Not a number	$s = u, \quad e = 2047, \quad f \neq 0$	`NaN`

you probably need it all over, which means double-precision library routines for methods and functions.

1.7.2 Python and the IEEE 754 Standard

Python is a relatively recent language with changes and extensions occurring as its use spreads and as its features mature. It should be no surprise then that Python does not at present adhere to all aspects, and especially the special cases, of the IEEE 754 standard. Probably the most relevant difference for us is that *Python does not support single (32 bit) precision floating point numbers*. So when we deal with a data type called a *float* in Python, it is the equivalent of a *double* in the IEEE standard. Since singles are inadequate for most scientific computing, this is not a loss. However be wary, if you switch over to Java or C you should declare your variables as `double`s and not as `float`s. While Python eliminates single-precision floats, it adds a new data type *complex* for dealing with complex numbers. Complex numbers are stored as pairs of doubles and are quite useful in science. We discuss complex numbers in Chapter 4, "Python Object-Oriented Programs", and Python's native implementation in §4.5.

The details of how closely Python adheres to the IEEE 754 standard depend upon the details of Python's use of the C or Java language to power the Python interpreter. In particular, with the recent 64 bit architectures for CPUs, the range may even be greater than the IEEE standard, and the abnormal numbers (±`INF`, `NaN`) may differ. Likewise, the exact conditions for overflows and underflows may also differ. That being the case, the exploratory exercises to follow become all that more interesting since we cannot say that we know what results you should obtain!

1.7.3 Over/Underflows Exercises

i) Consider the 32-bit single-precision floating-point number A:

	s	*e*	*f*
Bit position	31	30 23	22 0
Value	0	0000 1110	1010 0000 0000 0000 0000 000

a. What are the (binary) values for the sign s, the 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 1.625000.
d. Determine the full value of A.

ii) 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). Give your answer in decimal.
b. Check where under- and overflow occur for double-precision floating-point numbers (doubles in Java, floats in Python).
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 must observe your program's output as you explicitly pass through the limits. You accomplish this by continually adding and subtracting 1. (Because integer arithmetic uses *two's complement* arithmetic, you should expect some surprises.)

1.7.4 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 are good to 6–7 decimal places, while doubles are good to 15–16 places*. 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

$$7 = 0 \ \ 10000010 \ \ 1110\ 0000\ 0000\ 0000\ 0000\ 000, \tag{1.10}$$

$$10^{-7} = 0 \ \ 01100000 \ \ 1101\ 0110\ 1011\ 1111\ 1001\ 010, \tag{1.11}$$

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:

$$
\begin{aligned}
10^{-7} &= 0\ \ 01100001\ \ 0110\ 1011\ 0101\ 1111\ 1100101\ (0) \\
&= 0\ \ 01100010\ \ 0011\ 0101\ 1010\ 1111\ 1110010\ (10) \qquad (1.12)\\
&\cdots \\
&= 0\ \ 10000010\ \ 0000\ 0000\ 0000\ 0000\ 0000\ 000\ (0001101\cdots 0 \\
\Rightarrow \qquad & 7 + 1.0 \times 10^{-7} = 7. \qquad (1.13)
\end{aligned}
$$

Because there is no room left to store the last digits, they are lost, and after all this hard work the addition just gives 7 as the answer (truncation error in Figure 1.10). In other words, because a 32-bit computer stores only 6 or 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* ϵ_m as the maximum positive number that, on the computer, can be added to the number stored as 1 without changing that stored 1:

$$1_c + \epsilon_m \stackrel{\text{def}}{=} 1_c, \qquad (1.14)$$

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

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

where the actual value for ϵ is not known. In other words, except for powers of 2 that are represented exactly, we should assume that all single-precision numbers contain an error in the sixth decimal place and that all doubles have an error in the fifteenth place. And as is always the case with errors, we must assume that we do not know what the error is, for if we knew, then we would eliminate it! Consequently, the arguments we put forth regarding errors are always approximate, and that is the best we can do.

1.7.5 Determine Your Machine Precision

Write a program to determine the machine precision ϵ_m of your computer system within a factor of 2. A sample pseudocode is

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

A Python implementation is given in Listing 1.7, while a more precise one is **`ByteLimit.py`** on the instructor's guide.

Listing 1.7 The code **Limits.py** 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 key word (in *italics*).

```
# Limits.py: determines approximate machine precision

N = 3
eps = 1.0

for i in range(N):
```

```
    eps = eps/2
    one_Plus_eps = 1.0 + eps
    print('one + eps = ', one_Plus_eps)
    print('eps = ', eps)

print("Enter and return a character to finish")
s=raw_input()
```

i) Determine experimentally the precision of single-precision floats.

ii) Determine experimentally the precision of double-precision floats.

To print out a number in decimal format, the computer must convert from its internal binary representation. This not only takes time, but unless the number is a power of 2, leads to a loss of precision. So if you want a truly precise indication of the stored numbers, you should avoid conversion to decimals and instead print them out in octal or hexadecimal format (`\0NNN`).

1.8 PROBLEM: SUMMING SERIES

A classic numerical problem is the summation of a series to evaluate a function. As an example, consider the infinite series for $\sin x$:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \qquad \text{(exact)}.$$

Your **problem** is to use this series to calculate $\sin x$ for $x < 2\pi$ and $x > 2\pi$, with an absolute error in each case of less than 1 part in 10^8. While an infinite series is exact in a mathematical sense, it is not a good algorithm because we must stop summing at some point. An algorithm would be the finite sum

$$\sin x \simeq \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} \qquad \text{(algorithm)}. \tag{1.15}$$

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

1.8.1 Numerical Summation (Method)

Never mind that the algorithm (1.15) indicates that we should calculate $(-1)^{n-1}x^{2n-1}$ and then divide it by $(2n-1)!$ This is not a good way to compute. On the one hand, both $(2n-1)!$ and x^{2n-1} can get very large and cause overflows, even though their quotient may not. On the other hand, powers and factorials are very expensive (time-consuming) to evaluate on the computer. Consequently, a better approach is to use a single multiplication to relate the next term in the series to the previous one:

$$\frac{(-1)^{n-1}x^{2n-1}}{(2n-1)!} = \frac{-x^2}{(2n-1)(2n-2)} \frac{(-1)^{n-2}x^{2n-3}}{(2n-3)!}$$

$$\Rightarrow \qquad n\text{th term} = \frac{-x^2}{(2n-1)(2n-2)} \times (n-1)\text{th term}. \tag{1.16}$$

While we want to ensure definite accuracy for $\sin x$, 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 round-off error, a subject we talk about in Chapter 2, "Errors & Uncertainties in Computations"). To obtain an absolute error of 1 part in 10^8, we then stop the calculation when

$$\left|\frac{n\text{th term}}{\text{sum}}\right| < 10^{-8}, \tag{1.17}$$

where "term" is the last term kept in the series (1.15) and "sum" is the accumulated sum of all the terms. In general, you are free to pick any tolerance level you desire, although if it is too close to, or smaller than, machine precision, your calculation may not be able to attain it. A pseudocode for performing the summation is

```
term = x, sum = x, eps = 10^(-8)                    # Initialize do
  do term = -term*x*x/(2n+1)/(2*n-2);                 # New wrt old
  sum = sum + term                                      # Add term
  while abs(term/sum) > eps                        # Break iteration
end do
```

1.8.2 Implementation and Assessment

i) Write a program that implements this pseudocode for the indicated x values. Present the results as a table with headings x imax sum $|\text{sum} - \sin(x)|/\sin(x)$ where **`sin(x)`** is the value obtained from the built-in function. The last column here is the relative error in your computation. Modify the code that sums the series in a "good way" (no factorials) to one that calculates the sum in a "bad way" (explicit factorials).

ii) Produce a table as above.

iii) Start with a tolerance of 10^{-8} as in (1.17).

iv) 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.

v) Compare the number of decimal places of precision obtained with that expected from (1.17).

vi) Without using the identity $\sin(x+2n\pi) = \sin(x)$, 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.

vii) Show that as you keep increasing x, you will reach a regime where the algorithm does not even converge.

viii) Now make use of the identity $\sin(x + 2n\pi) = \sin(x)$ to compute $\sin x$ for large x values where the series otherwise would diverge.

ix) Repeat the calculation using the "bad" version of the algorithm (the one that calculates factorials) and compare the answers.

x) Set your tolerance level to a number smaller than machine precision and see how this affects your conclusions.

Beginnings are hard.

—*Chaim Potok*

Chapter Two

Errors & Uncertainties in Computations

To err is human, to forgive divine.

— Alexander Pope

Whether you are careful or not, errors and uncertainties are 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 algorithms or models can fail. Although it stifles creativity to keep thinking "error" when approaching a computation, it certainly is a waste of time, and may lead to harm, to work with results that are meaningless ("garbage") because of errors. In this chapter we examine some of the errors and uncertainties that may occur in computations. Even though we do not dwell on it, the lessons of this chapter apply to all other chapters as well.

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*
Errors	pdf	2.1

2.1 TYPES OF ERRORS (THEORY)

Let us say that you have a program of high complexity. To gauge why errors should be of concern, imagine a program with the logical flow

$$\text{start} \to U_1 \to U_2 \to \cdots \to U_n \to \text{end}, \tag{2.1}$$

where each unit U might be a statement or 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 medium-sized program with $n = 1000$ steps and that the probability of each step being correct is almost one, $p \simeq 0.9993$. This means that you end up with $P \simeq \frac{1}{2}$, that is, a final answer that is as likely wrong as right (not a good way to build a bridge). The problem is that, as a scientist, you want a result that is correct—or at least in which the uncertainty is small and of known size.

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 (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: imprecision caused by events such as fluctuations in electronics, 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-s calculation, a week-long calculation may have to be run several times to check reproducibility.

Approximation errors: imprecision arising from simplifying the mathematics so that a problem can be solved 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,

$$\begin{aligned}\sin(x) &= \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} \quad \text{(exact)} \\ &\simeq \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} \quad \text{(algorithm)} \\ &= \sin(x) + \mathcal{E}(x, N),\end{aligned} \tag{2.2}$$

where $\mathcal{E}(x, N)$ is the approximation error and where in this case $\mathcal{E}$ is the series from $N+1$ to ∞. Because approximation error arises from the algorithm we use to approximate the mathematics, it is also called *algorithmic error*. For every reasonable approximation, the approximation error should decrease as N increases and vanish in the $N \to \infty$ limit. Specifically for (2.3), because the scale for N is set by the value of x, a small approximation error requires $N \gg x$. So if x and N are close in value, the approximation error will be large.

Round-off errors: imprecision 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 round-off error accumulates as the computer handles more numbers, that is, as the number of steps in a computation increases, and may cause some algorithms to become *unstable* with a rapid increase in error. In some cases, round-off error may become the major component in your answer, leading to what computer experts call *garbage*. For example, if your computer kept four decimal places, then it will store $\frac{1}{3}$ as 0.3333 and $\frac{2}{3}$ as 0.6667, where the computer has "rounded off" the last digit in $\frac{2}{3}$. Accordingly, if we ask the computer to do as simple a calculation as $2(\frac{1}{3}) - \frac{2}{3}$, it produces

$$2\left(\frac{1}{3}\right) - \frac{2}{3} = 0.6666 - 0.6667 = -0.0001 \neq 0. \tag{2.3}$$

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

When considering the precision of calculations, it is good to recall our discussion in Chapter 1, "Computational Science Basics," of *significant figures* and of 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

$$a = 11223344556677889900 = 1.12233445566778899 \times 10^{19}. \tag{2.4}$$

Because the exponent is stored separately and is a small number, we can assume that it will be stored in full precision. In contrast, some of the digits of the mantissa may be truncated.

In double precision 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 are lost. As we see below, when we perform calculations with words of fixed length, it is inevitable that errors will be introduced (at least) into the least significant parts of the words.

2.1.1 Model for Disaster: Subtractive Cancellation

A calculation employing numbers that are stored only approximately on the computer can be expected to yield only an approximate answer. To demonstrate the effect of this type of uncertainty, we model the computer representation x_c of the exact number x as

$$x_c \simeq x(1+\epsilon_x). \tag{2.5}$$

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

$$\begin{aligned} a = b - c \quad &\Rightarrow \quad a_c \simeq b_c - c_c \simeq b(1+\epsilon_b) - c(1+\epsilon_c) \\ &\Rightarrow \quad \frac{a_c}{a} \simeq 1 + \epsilon_b \frac{b}{a} - \frac{c}{a}\epsilon_c. \end{aligned} \tag{2.6}$$

We see from (2.6) that the resulting error in a is essentially a weighted average of the errors in b and c, with no assurance that the last two terms will cancel. Of special importance here is to observe that the error in the answer a_c increases when we subtract two nearly equal numbers ($b \simeq c$) because then we are subtracting off the most significant parts of both numbers and leaving the error-prone least-significant parts:

$$\frac{a_c}{a} \stackrel{\text{def}}{=} 1 + \epsilon_a \simeq 1 + \frac{b}{a}(\epsilon_b - \epsilon_c) \simeq 1 + \frac{b}{a}\max(|\epsilon_b|, |\epsilon_c|). \tag{2.7}$$

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 (2.7)].

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

We have already seen an example of subtractive cancelation in the power series summation for $\sin x \simeq x - x^3/3! + \cdots$ for large x. A similar effect occurs for $e^{-x} \simeq 1 - x + x^2/2! - x^3/3! + \cdots$ for large x, where the first few terms are large but of alternating sign, leading to an almost total cancelation in order to yield the final small result. (Subtractive cancelation can be eliminated by using the identity $e^{-x} = 1/e^x$, although round-off error will still remain.)

2.1.2 Subtractive Cancelation Exercises

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

$$ax^2 + bx + c = 0 \tag{2.8}$$

has an analytic solution that can be written as either

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \text{or} \quad x'_{1,2} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}. \tag{2.9}$$

Inspection of (2.9) indicates that subtractive cancelation (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 cancelation increases and relate this to the known machine precision. (*Hint*: A good test case employs $a = 1$, $b = 1$, $c = 10^{-n}$, $n = 1, 2, 3, \ldots$.)
c. Extend your program so that it indicates the most precise solutions.

2. As we have seen, subtractive cancelation occurs when summing a series with alternating signs. As another example, consider the finite sum

$$S_N^{(1)} = \sum_{n=1}^{2N} (-1)^n \frac{n}{n+1}. \tag{2.10}$$

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

$$S_N^{(2)} = -\sum_{n=1}^{N} \frac{2n-1}{2n} + \sum_{n=1}^{N} \frac{2n}{2n+1}. \tag{2.11}$$

All terms are positive in this form with just a single subtraction at the end of the calculation. Yet even this one subtraction and its resulting cancelation can be avoided by combining the series analytically to obtain

$$S_N^{(3)} = \sum_{n=1}^{N} \frac{1}{2n(2n+1)}. \tag{2.12}$$

Even though all three summations $S^{(1)}$, $S^{(2)}$, and $S^{(3)}$ are mathematically equal, they may give different numerical results.

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* the 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 $N = 1,000,000$. (Recollect that $\log_{10} x = \ln x / \ln 10$.) The negative of the ordinate in this plot gives an approximate value for the number of significant figures.
c. See whether straight-line behavior for the error occurs in some region of your plot. This indicates that the error is proportional to a 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

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

Both series are finite as long as N is finite, and when summed analytically both give the same answer. Nonetheless, because of round-off error, the numerical value of $S^{(\text{up})}$ will not be precisely that of $S^{(\text{down})}$.

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

2.1.3 Round-off Error in a Single Step

Let's start by seeing how error arises from a single division of the computer representations of two numbers:

$$
\begin{aligned}
a = \frac{b}{c} \;\Rightarrow\; a_c = \frac{b_c}{c_c} &= \frac{b(1+\epsilon_b)}{c(1+\epsilon_c)}, \\
\Rightarrow\; \frac{a_c}{a} &= \frac{1+\epsilon_b}{1+\epsilon_c} \simeq (1+\epsilon_b)(1-\epsilon_c) \simeq 1+\epsilon_b-\epsilon_c, \\
\Rightarrow\; \frac{a_c}{a} &\simeq 1+|\epsilon_b|+|\epsilon_c|. \qquad (2.13)
\end{aligned}
$$

Here we ignore the very small ϵ^2 terms and add errors in absolute value since we cannot assume that we are fortunate enough to have unknown errors cancel each other. Because we add the errors in absolute value, this same rule holds for multiplication. Equation (2.13) is just the basic rule of error propagation from elementary laboratory work: You add the uncertainties in each quantity involved in an analysis to arrive at the overall uncertainty.

We can even generalize this model to estimate the error in the evaluation of a general function $f(x)$, that is, the difference in the value of the function evaluated at x and at x_c:

$$
\mathcal{E} = \frac{f(x)-f(x_c)}{f(x)} \simeq \frac{df(x)/dx}{f(x)}(x-x_c). \qquad (2.14)
$$

So, for

$$
f(x) = \sqrt{1+x}, \qquad \frac{df}{dx} = \frac{1}{2}\frac{1}{\sqrt{1+x}} \qquad (2.15)
$$

$$
\Rightarrow \quad \mathcal{E} \simeq \frac{1}{2}\frac{1}{\sqrt{1+x}}(x-x_c). \qquad (2.16)
$$

If we evaluate this expression for $x = \pi/4$ and assume an error in the fourth place of x, we obtain a similar relative error of 1.5×10^{-4} in $\sqrt{1+x}$.

2.1.4 Round-off Error Accumulation After Many Steps

There is a useful model for approximating how round-off 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 Chapter 5, "Monte Carlo Simulations," the total distance covered in N steps of length r, is, on the average,

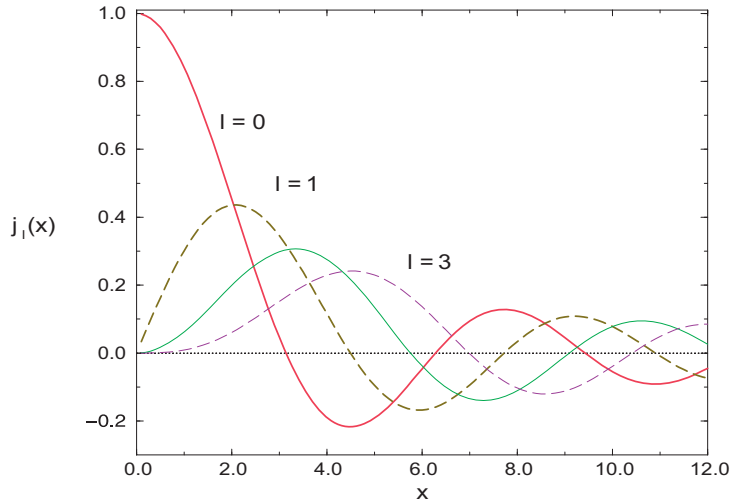
$$
R \simeq \sqrt{N}\, r. \qquad (2.17)
$$

By analogy, the total relative error ϵ_{ro} arising after N calculational steps each with machine precision error ϵ_m is, on the average,

$$
\epsilon_{\text{ro}} \simeq \sqrt{N}\, \epsilon_m. \qquad (2.18)
$$

If the round-off 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 cancelation, and the error may increase as $N\epsilon_m$. Even worse, in some recursive algorithms, where the error generation is coherent, such as the upward recursion for Bessel functions, the error increases as $N!$.

Figure 2.1 The first four spherical Bessel functions $j_l(x)$ as functions of x. Notice that for small x, the values for increasing l become progressively smaller.



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 that a program running for 3 h performs approximately 10^{14} operations. Therefore, if round-off 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 requires double precision and a good algorithm. If we want a higher-precision answer, then we will need a very good algorithm.

2.2 ERRORS IN SPHERICAL BESSEL FUNCTIONS (PROBLEM)

Accumulating round-off errors often limits the ability of a program to calculate accurately. Your **problem** is to compute the spherical Bessel and Neumann functions $j_l(x)$ and $n_l(x)$. These function are, respectively, the regular/irregular (nonsingular/singular at the origin) solutions of the differential equation

$$x^2 f''(x) + 2x f'(x) + \left[x^2 - l(l+1)\right] f(x) = 0. \tag{2.19}$$

The spherical Bessel functions are related to the Bessel function of the first kind by $j_l(x) = \sqrt{\pi/2x}J_{n+1/2}(x)$. They occur in many physical problems, such as the expansion of a plane wave into spherical partial waves,

$$e^{i\mathbf{k}\cdot\mathbf{r}} = \sum_{l=0}^{\infty} i^l\,(2l+1) j_l(kr)\,P_l(\cos\,\theta). \tag{2.20}$$

Figure 2.1 shows what the first few j_l look like, and Table 2.1 gives some explicit values. For the first two l values, explicit forms are

$$j_0(x) = +\frac{\sin\,x}{x}, \qquad j_1(x) = +\frac{\sin\,x}{x^2} - \frac{\cos\,x}{x} \tag{2.21}$$

$$n_0(x) = -\frac{\cos\,x}{x}. \qquad n_1(x) = -\frac{\cos\,x}{x^2} - \frac{\sin\,x}{x}. \tag{2.22}$$

Table 2.1 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	$+9.518519719\,10^{-6}$	$+9.616310231\,10^{-10}$	$+2.901200102\,10^{-16}$
1	$+9.006581118\,10^{-3}$	$+9.256115862\,10^{-05}$	$+2.826498802\,10^{-08}$
10	$-3.949584498\,10^{-2}$	$-5.553451162\,10^{-02}$	$+1.255780236\,10^{-01}$

2.2.1 Numerical 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 summing its asymptotic expansion for large x values. The approach we adopt is based on the *recursion relations*

$$j_{l+1}(x) = \frac{2l+1}{x} j_l(x) - j_{l-1}(x), \quad \text{(up)}, \tag{2.23}$$

$$j_{l-1}(x) = \frac{2l+1}{x} j_l(x) - j_{l+1}(x), \quad \text{(down)}. \tag{2.24}$$

Equations (2.23) and (2.24) are the same relation, one written for upward recurrence from small to large l values, and the other for downward recurrence to small l values. With just a few additions and multiplications, recurrence relations permit rapid, simple computation of the entire set of j_l values for fixed x and all l.

To recur upward in l for fixed x, we start with the known forms for j_0 and j_1 (2.21) and use (2.23). As you will prove for yourself, this upward recurrence usually seems to work at first but then fails. The reason for the failure can be seen from the plots of $j_l(x)$ and $n_l(x)$ *versus* x (Figure 2.1). If we start at $x \simeq 2$ and $l = 0$, we will see that as we recur j_l up to larger l values with (2.23), we are essentially taking the difference of two "large" functions to produce a "small" value for j_l. This process suffers from subtractive cancelation and always reduces the precision. As we continue recurring, we take the difference of two small functions with large errors and produce a smaller function with yet a larger error. After a while, we are left with only round-off 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 $n_l(x)$:

$$j_l^{(c)} = j_l(x) + \epsilon n_l(x). \tag{2.25}$$

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 when the numerical value of $n_l(x)$ is much larger than that of $j_l(x)$ because even a minuscule amount of a very large number may be large. In contrast, if we use the upward recurrence relation (2.23) to produce the spherical Neumann function n_l, there will be no problem because we are combining small functions to produce larger ones (Figure 2.1), a process that does not contain subtractive cancelation.

The simple solution to this problem (*Miller's device*) is to use (2.24) for downward recursion of the j_l values starting at a large value $l = L$. This avoids subtractive cancelation 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 $j_L^{(c)}$, after a short while we will 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 $j_L^{(c)}$, the relative values will be accurate. The absolute values are fixed from the know value (2.21), $j_0(x) = \sin x/x$. Because the recurrence relation is a linear relation between the j_l values, we need only normalize all the computed values via

$$j_l^{\text{normalized}}(x) = j_l^{\text{compute}}(x) \times \frac{j_0^{\text{analytic}}(x)}{j_0^{\text{compute}}(x)}. \tag{2.26}$$

Accordingly, after you have finished the downward recurrence, you obtain the final answer by normalizing all $j_l^{(c)}$ values based on the known value for j_0.

2.2.2 Implementation and Assessment: Recursion Relations

A program implementing recurrence relations is most easily written using subscripts. If you need to polish up on your skills with subscripts, you may want to study our program **`Bessel.py`** in Listing 2.1 before writing your own.

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 (meaning a relative error $\simeq 10^{-10}$). See Table 2.1 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.

Listing 2.1 **Bessel.py** determines spherical Bessel functions by downward recursion (you should modify this to also work by upward recursion).

```
# Bessel.py

from visual.graph import *

Xmax = 40.
Xmin = 0.25
step = 0.1                                          # Global class variables
order = 10; start = 50          # Plot j_order
graph1 = gdisplay(width = 500, height = 500, title = 'Sperical Bessel, \
       L = 1 (red), 10',xtitle = 'x', ytitle = 'j(x)',\
       xmin=Xmin,xmax=Xmax,ymin=-0.2,ymax=0.5)
funct1 = gcurve(color=color.red)
funct2 = gcurve(color=color.green)
def down (x, n, m):                          # Method down, recurs downward
     j = zeros( (start + 2), float)
     j[m + 1] = j[m] = 1.                            # Start with anything
     for k in range(m, 0, -1):
         j[k - 1] = ( (2.*k + 1.)/x)*j[k] - j[k + 1]
     scale = (sin(x)/x)/j[0]                 # Scale solution to known j[0]
     return j[n] * scale

for x in arange(Xmin, Xmax, step):
    funct1.plot(pos = (x, down(x, order, start)))

for x in arange(Xmin, Xmax, step):
    funct2.plot(pos = (x, down(x,1,start)))
```

2.3 EXPERIMENTAL ERROR INVESTIGATION (PROBLEM)

Numerical algorithms play a vital role in computational physics. 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?

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 use more terms." Well, some algorithms may be asymptotic expansions that just approximate a function in certain regions of parameter space and converge only 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 round-off errors. And because round-off errors eventually diverge to infinity, the best we can hope for is a "best" approximation. Good algorithms are good not only because they are fast but also because being fast means that round-off error does not have much time to grow.

Let us assume that an algorithm takes N steps to find 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:

$$\epsilon_{\text{approx}} \simeq \frac{\alpha}{N^\beta}. \tag{2.27}$$

Here α and β are empirical constants that change for different algorithms and may be only approximately constant, and even then only as $N \to \infty$. 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, round-off error tends to grow slowly and somewhat randomly with N. If the round-off errors in each step of the algorithm are not correlated, then we know from previous discussion that we can model the accumulation of error as a random walk with step size equal to the machine precision ϵ_m:

$$\epsilon_{\text{ro}} \simeq \sqrt{N}\epsilon_m. \tag{2.28}$$

This is the slow growth with N that we expect from round-off error. The total error in a computation is the sum of the two types of errors:

$$\epsilon_{\text{tot}} = \epsilon_{\text{approx}} + \epsilon_{\text{ro}} \tag{2.29}$$

$$\epsilon_{\text{tot}} \simeq \frac{\alpha}{N^\beta} + \sqrt{N}\epsilon_m. \tag{2.30}$$

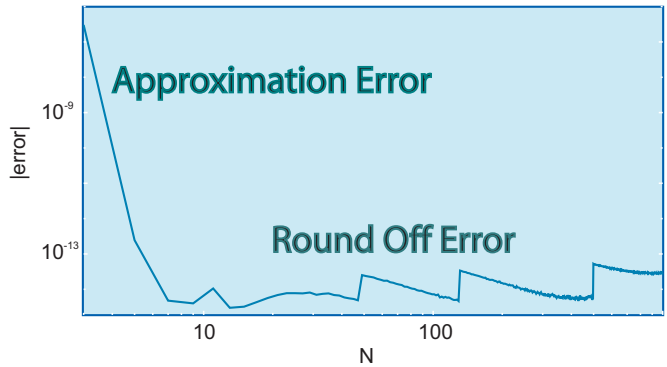
For small N we expect the first term to be the larger of the two but ultimately to be overcome by the slowly growing round-off error.

As an example, in Figure 2.2 we present a log-log plot of the relative error in numerical integration using the Simpson integration rule (Chapter 6, "Integration"). We use the $\log_{10}$ of the relative error because its negative tells us the number of decimal places of precision obtained.[1] As a case in point, let us assume $\mathcal{A}$ is the exact answer and $A(N)$ the computed answer. If

$$\frac{\mathcal{A} - A(N)}{\mathcal{A}} \simeq 10^{-9}, \quad \text{then} \quad \log_{10}\left|\frac{\mathcal{A} - A(N)}{\mathcal{A}}\right| \simeq -9. \tag{2.31}$$

[1] Most computer languages use $\ln x = \log_e x$. Yet since $x = a^{\log_a x}$, we have $\log_{10} x = \ln x / \ln 10$.

Figure 2.2 A log-log plot of relative error *versus* the number of points used for a numerical integration. The ordinate value of $\sim 10^{-14}$ at the minimum indicates that $\sim$14 decimal places of precision are obtained before round-off error begins to build up. Notice that while the round-off error does fluctuate indicating a statistical aspect of error accumulation, on the average it is increasing but more slowly than did the algorithm's error decrease. slowly.



We see in Figure 2.2 that the error does show a rapid decrease for small N, consistent with an inverse power law (2.27). In this region the algorithm is converging. As N is increased, the error starts to look somewhat erratic, with a slow increase on the average. In accordance with (2.29), in this region round-off error has grown larger than the approximation error and will continue to grow for increasing N. Clearly then, the smallest total error will be obtained if we can stop the calculation at the minimum near 10^{-14}, that is, when $\epsilon_{\text{approx}} \simeq \epsilon_{ro}$.

In realistic calculations 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, a graph (Figure 2.2) of your answer and deduce the manner in which your algorithm is converging. Specifically, at some point you should see that the mantissa of the answer changes only in the less significant digits, with that place moving further to the right of the decimal point as the calculation executes more steps. Eventually, however, as the number of steps becomes even larger, round-off error leads to a fluctuation in the less significant digits, with a gradual increase on the average. It is best to quit the calculation before this occurs.

Based upon this understanding, an approach to obtaining the best approximation is to deduce when your answer behaves like (2.29). To do that, we call $\mathcal{A}$ the exact answer and $A(N)$ the computed answer after N steps. We assume that for large enough values of N, the approximation converges as

$$A(N) \simeq \mathcal{A} + \frac{\alpha}{N^{\beta}}, \tag{2.32}$$

that is, that the round-off error term in (2.29) is still small. We then run our computer program with $2N$ steps, which should give a better answer, and use that answer to eliminate the unknown $\mathcal{A}$:

$$A(N) - A(2N) \simeq \frac{\alpha}{N^{\beta}}. \tag{2.33}$$

To see if these assumptions are correct and determine what level of precision is possible for the best choice of N, plot $\log_{10} |[A(N) - A(2N)]/A(2N)|$ *versus* $\log_{10} N$, similar to what we have done in Figure 2.2. 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 β from the slope. As N gets larger, you should see the graph change from a straight-line decrease to a slow increase as round-off error

begins to dominate. A good place to quit is before this. In any case, now you understand the error in your computation and therefore have a chance to control it.

As an example of how different kinds of errors enter into a computation, we assume we know the analytic form for the approximation and round-off errors:

$$\epsilon_{\text{approx}} \simeq \frac{1}{N^2}, \quad \epsilon_{\text{ro}} \simeq \sqrt{N}\epsilon_m, \tag{2.34}$$

$$\Rightarrow \quad \epsilon_{\text{tot}} = \epsilon_{\text{approx}} + \epsilon_{\text{ro}} \simeq \frac{1}{N^2} + \sqrt{N}\epsilon_m. \tag{2.35}$$

The total error is then a minimum when

$$\frac{d\epsilon_{\text{tot}}}{dN} = \frac{-2}{N^3} + \frac{1}{2}\frac{\epsilon_m}{\sqrt{N}} = 0, \tag{2.36}$$

$$\Rightarrow \quad N^{5/2} = \frac{4}{\epsilon_m}. \tag{2.37}$$

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

$$N^{5/2} \simeq \frac{4}{10^{-7}} \quad \Rightarrow \quad N \simeq 1099, \quad \Rightarrow \quad \epsilon_{\text{tot}} \simeq 4 \times 10^{-6}. \tag{2.38}$$

In this case most of the error is due to round-off and is not approximation error. Observe, too, that even though this is the minimum total error, the best we can do is about 40 times machine precision (in double precision the results are better).

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

$$\epsilon_{\text{approx}} \simeq \frac{2}{N^4}. \tag{2.39}$$

The total error is now

$$\epsilon_{\text{tot}} = \epsilon_{\text{ro}} + \epsilon_{\text{approx}} \simeq \frac{2}{N^4} + \sqrt{N}\epsilon_m. \tag{2.40}$$

The number of points for minimum error is found as before:

$$\frac{d\epsilon_{\text{tot}}}{dN} = 0 \quad \Rightarrow \quad N^{9/2} \quad \Rightarrow \quad N \simeq 67 \quad \Rightarrow \quad \epsilon_{\text{tot}} \simeq 9 \times 10^{-7}. \tag{2.41}$$

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 the better algorithm is quicker and, by using fewer steps, produces less round-off error.

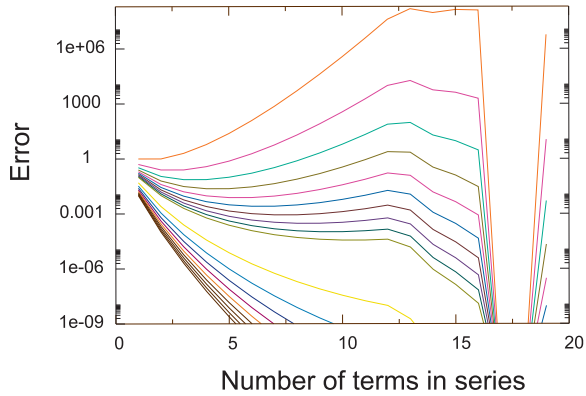
Exercise: Estimate the error now for a double-precision calculation. ▌

2.3.1 Error Assessment

We have already discussed the Taylor expansion of $\sin x$:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}x^{2n-1}}{(2n-1)!}. \tag{2.42}$$

Figure 2.3 The error in the summation of the series for e^{-x} *versus* N for various x values. The values of x increase vertically for each curve. Note that a negative initial slope corresponds to decreasing error with N, and that the dip corresponds to a rapid convergence followed by a rapid increase in error. (Courtesy of J. Wiren.)



This series converges in the mathematical sense for all values of x. Accordingly, a reasonable algorithm to compute the $\sin(x)$ might be

$$\sin(x) \simeq \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!}. \tag{2.43}$$

While in principle it should be faster to see the effects of error accumulation in this algorithm by using single-precision numbers, C and Python 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 $\sin(x)$ as the finite sum (2.43). (If you already did this in Chapter 2, "Computational Science Basics," then you may reuse that program and its results here.)
2. Calculate your series for $x \leq 1$ and compare it to the built-in function `Math.sin(x)` (you may assume that the built-in function is exact). Stop your summation at an N value for which the next term in the series will be no more than 10^{-7} of the sum up to that point,

$$\frac{|(-1)^N x^{2N+1}|}{(2N-1)!} \leq 10^{-7} \left| \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} \right|. \tag{2.44}$$

3. Examine the terms in the series for $x \simeq 3\pi$ and observe the significant subtractive cancelations that occur when large terms add together to give small answers. [Do not use the identity $\sin(x + 2\pi) = \sin x$ to reduce the value of x in the series.] In particular, print out the near-perfect cancelation around $n \simeq x/2$.
4. See if better precision is obtained by using trigonometric identities to keep $0 \leq x \leq \pi$.
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 it no longer converges.
6. Make a series of graphs of the error *versus* N for different values of x. (See Chapter 3, "Visualization Tools.") You should get curves similar to those in Figure 2.3.

Because this series summation is such a simple, correlated process, the round-off error does not accumulate randomly as it might for a more complicated computation, and we do not obtain the error behavior (2.32). We will see the predicted error behavior when we examine integration rules in Chapter 6, “Integration.”

Chapter Three

Visualization Tools

If I can't picture it, I can't understand it.

— Albert Einstein

In this chapter we discuss the tools needed to visualize data produced by simulations and measurements. Whereas other books may choose to relegate this discussion to an appendix, or not to include it at all, we believe that visualization is such an integral part of computational science, and so useful for your work in the rest of this book, that we have placed it right here, up front.

ALL THE VISUALIZATION TOOLS WE discuss are powerful enough for professional scientific work and are free or open source. Commercial packages such as Matlab, AVS, Amira, and Noesys produce excellent scientific visualization but are less widely available. Mathematica and Maple have excellent visualization packages as well, but we have not found them convenient when dealing with large numerical data sets.[1] The tools we discuss, and have used in preparing the visualizations for the text, are

Matplotlib: Matplotlib is a very powerful library of plotting functions callable from within Python that is capable of producing publication quality figures in a number of output formats. It is, by design, similar to the plotting packages with *MATLAB*, and is made more powerful by its use of the *numpy* package for numerical work. In addition to 2-D plots, Matplotlib can also create interactive, 3-D visualizations of data.

Visual (VPython): The programming language "Python" is so often employed with the *Visual* graphics module and the IDLE interface that the combination is often referred to as *Vpython*[2]. Much of the use of the Visual extension has been to create 3-D demonstrations and animations for education, which are surprisingly easy to make and useful. We will use Visual at times for 2-D plots of numerical data and animations.

Gnuplot: 2-D and 3-D plotting, traditionally stand-alone (after you have finished running your program). Originally for Unix operating systems, with an excellent windows port also available. Recently, a Python Package **Gnuplot.py** has been developed and it allows you to call gnuplot from within Python programs and thus plot data as they are computed. We have traditionally used gnuplot to create simple animations by plotting slightly different graphs one after another.

Ace/gr (Grace): Traditionally, a stand-alone, menu-driven, publication-quality 2-D plotting for Unix systems; can run under MS Windows with Cygwin. Recently, the **pygrace** package has become available and it permits access to the Grace plotting tool.

[1] Visualization with Maple and Mathematica is discussed in [L 05].

[2] The Visual package was written by David Scherer in 2000 as a sophomore at Carnegie University and is being updated by Bruce Sherwood.

OPenDX: Formerly IBM DataExplorer. A stand-alone, multidimensional data tool for Unix or for Windows under Cygwin (tutorial in [LP&B 08]).

Tkinter: Although beyond the level of this text, Python contains a graphical user interface (GUI) programming module called *Tkinter* or *Tk*. It is employed by the packages we use and you may see them refer to it.

3.1 DATA VISUALIZATION (GENERAL)

One of the most rewarding uses of computers is visualizing the results of calculations. While in the past this was done with 2-D plots, in modern times it is regular practice to use 3-D (surface) plots, volume rendering (dicing and slicing), and animation, as well as virtual reality (gaming) tools. These types of visualizations are often breathtakingly beautiful and may provide deep insights into problems by letting us see and "handle" the functions with which we are working. Visualization also assists in the debugging process, the development of physical and mathematical intuition, and the all-around enjoyment of your work. One of the reasons for visualization's effectiveness may arise from the large fraction ($\sim$10% direct and $\sim$50% with coupling) of our brain involved in visual processing and the advantage gained by being able to use this brainpower as a supplement to our logical powers.

In thinking about ways to present your results, keep in mind that the point of visualization is to make the science clearer and to communicate your work to others. It follows then that you should make all figures as clear, informative, and self-explanatory as possible, especially if you will be using them in presentations without captions. This means labels for all curves and data points, a title, and labels on the axes.[3] After this, you should look at your visualization and ask whether there are better choices of units, ranges of axes, colors, style, and so on, that might get the message across better and provide better insight. Considering the complexity of human perception and cognition, there may not be a single best way to visualize a particular data set, and so some trial and error may be necessary to see what looks best. Although all this may seem like a lot of work, the more often you do it the quicker and better you get at it and the better you will be able to communicate your work to others.

3.2 PYTHON MATPLOTLIB, VISUAL (VPYTHON) PACKAGES

In this chapter we describe the use of the Matplotlib and the Visual packages for plotting graphs within your Python programs. (The Visual package combined with Python is often called VPython.) Both packages make nice 2-D plots, while only Matplotlib makes 3-D/surface plots. However, although VPython cannot make surface plots,it can create 3-D geometric figures like spheres and cubes.

As an alternative, either by calling *Gnuplot.py* package or by using the stand-alone program, Gnuplot does very nice 2-D and 3-D plotting, and in §3.7.3 we discuss how to do that. The approaches to Matplotlib and Gnuplot for 3-D plotting have some differences: Matplotlib requires a function $z(x, y)$, while Gnuplot works with just the stored array of z values, using the row and column numbers as the x and y values.

[3] Although this may not need saying, place the independent variable x along the abscissa (horizontal), and the dependent variable $y = f(x)$ along the ordinate.

3.2.1 Installing Python's Matplotlib and Visual Packages

We recommend that you install Vpython first and then Matplotlib as follows (we describe the Windows installation, with the Linux and Mac OSX being similar):

1. *Make sure to install packages in the prescribed order!* Download and install Python first, before the other packages.
2. To install Python go to the VPython Web site:
   `vpython.org/`
3. The Python language is being updated continually and there may be several versions available for download. Since not all packages (for example Matplotlib) work with the latest version, we recommend you download the latest version that is consistent with Matplotlib (currently **`python-2.7.msi`**).
4. After Python is installed, install the VPython package. As described on the VPython site, ensure that the Python and VPython you download are compatible with each other.
5. Numpy should now have installed automatically. This is a Python library for numerical work and especially for matrix operations. Its use is similar to MATLAB with both being based on the LAPACK linear algebra subroutine library[LAP 00]. Numpy is also used by the plotting package Matplotlib.
6. Next download and install Matplotlib for plotting math functions. We got it from
   `sourceforge.net/projects/matplotlib/`
   and downloaded **`matplotlib-0.98.53.win32-py2.5.exe`**.

There should be good documentation in the **`Doc`** directory of your Python installation, with further examples and documentation found on Web sites, for example:

`matplotlib.sourceforge.net/contents.html`

3.3 USING PACKAGES IN PYTHON PROGRAMS

A package or module is a collection of related methods or classes of methods that are assembled together into a subroutine library. Incorporating these packages permits you to vastly extend the power of your Python programs[4]. You use a package by placing an appropriate **`import`** or **`from`** statement at the beginning of your program. These statements find, compile, and run the methods contained in the package. The **`import`** statement loads the entire module package, which is efficient, but may require you to include the package name as a prefix to the method you want. For example,

```
>>> import visual.graph
>>> y1 = visual.graph.gcurve(color=visual.graph.color.blue)        Prefixes!
```

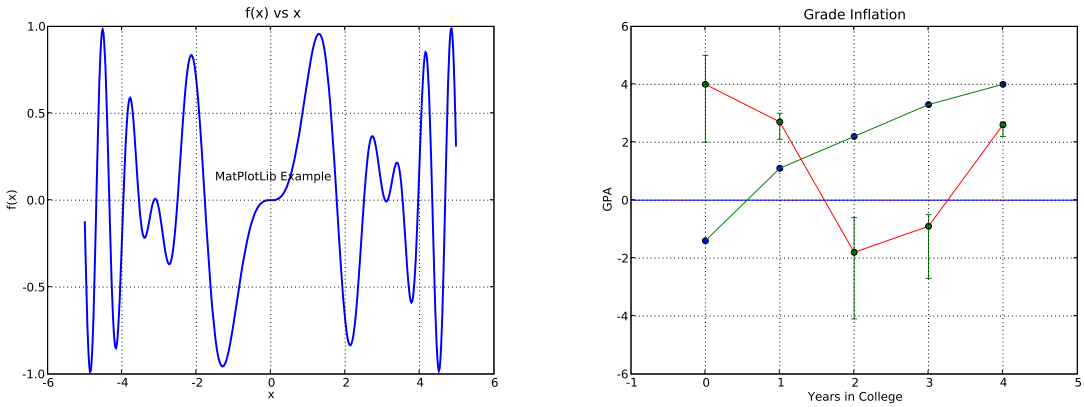
where **`>>>`** is the prompt for a Python shell. Some of the typing can be avoided by assigning a symbol to the package name:

```
>>> import visual.graph as p                                  Import graph method from visual
>>> from mpl_tools.mplot3d import Axes3D                      Just for 3D
>>> y1 = p.gcurve(color=vg.color.blue, delta=3)               How used
```

Technically, **`plot`** works here without a prefix because the imported name is copied into the *scope* from where the **`from`** statement appear, and because the individual method has been copied (**`import`** loads the package, but individual methods still need to be copied). There is a starred version of the **`from`** command that copies *all* of the methods of a package and

[4]The Python Package Index at `pypi.python.org/pypi` is a repository of 5835 software packages, and it is all free.

Figure 3.1 *Left:* Output from `EasyMatPlot.py` of a simple, x-y plot using the `matplotlib` package. *Right:* Output from `GradesMatPlot.py` that places two sets of data points, two curves, and unequal upper and lower error bars, all on one plot.



thereby extends the namespace to include all of the methods. Now you can leave out even more prefixes:

```
>>> from pylab import *          Import all methods from pylab package
>>> plot(x, y, '-', lw=2)        A pylab method without prefix
>>> polar(...)                   A different method
```

Don't be confused with the names of the modules being imported, it's all matplotlib. The **`pylab`** package provides an assortment of plotting commands as well as non-plotting functions from numpy and matplotlib.mlab.

3.4 MATPLOTLIB FOR 2-D GRAPHS IN PYTHON

3.4.1 Using Arrays in Python

One way Matplotlib does its plotting so well is by creating *numpy* arrays of the data to be plotted. Computing with numpy arrays is fast and simple in Python. In part, this is because numpy is optimized for arrays, and in part because arrays are handled and processed much as if they were simple, scalar variables[5]. We start with an example of *slicing*, a technique that is also used in ordinary Python with lists and tuples, in which two indices separated by a colon indicate a range:

```
from visual import *
stuff = zeros(10, float)
t = arange(4)
stuff[3:7] = sqrt(t+1)
```

Here we start by creating the numpy array **`stuff`** of floats, all of whose 10 elements are initialized to zero. Then we create the array **`t`** containing the four elements [0, 1, 2, 3] by assigning 4 variables uniformly in the range 0–4 (the "a" in **`arange`** creates floating-point variables, **`range`** creates integers). Next we use a slice to assign [sqrt(0+1), sqrt(1+1), sqrt(2+1), sqrt(3+1)] = [1, 1.414, 1.732, 2] to the middle elements of the **`stuff`** array.

[5] We thank Bruce Sherwood for helpful comments on these points.

Note that the numpy version of the **sqrt** function, one of many universal function (*ufunctions*) supported by numpy[6], has the amazing property of automatically outputting an array whose length is that of its argument, in this case, the array **t**. In general, major power in numpy comes from its *broadcasting* operation, an operation in which values are assigned to multiple elements via a single assignment statement. Broadcasting permits Python to *vectorize* array operations, which means that the same operation can be performed on different array elements in parallel (or nearly so). Broadcasting also speeds up processing since array operations occur C instead of Python, and with a minimum of array copies being made. Here is a simple aspect of broadcasting:

```
w = zeros(100, float)
w = 23.7
```

The first line creates the numpy array **w**, and the second line "broadcasts" the value 23.7 to all elements in the array. There are many possible array operations in numpy and various rules pertaining to them; we recommend that the serious user explore the extensive numpy documentation for additional information.

3.4.2 Matplotlib Examples

Matplotlib is a powerful plotting package that lets you make 2-D and 3-D graphs, histograms, power spectra, bar charts, errorcharts, scatterplots, and so forth, directly from your Python program. It is free, uses the sophisticated numerics of numpy and LAPACK[LAP 00], and, believe it or not, it is easy to use. Matplot commands are by design similar to the plotting commands of MATLAB, a commercial problem-solving environment that is particularly popular with engineers. As true in MATLAB, Matplotlib assumes that you have placed the x and y values that you wish to plot into arrays. Because Matplotlib is not built into Python you must import individual methods or the entire package into your program. On line 3 of **EasyMatPlot.py** in Listing 3.1 we import Matplotlib as the **pylab** library:

```
>>> from pylab import *
```
Load Matplotlib commands

As already noted, Matplotlib requires *you* to calculate and input arrays of the x and y values of the points you want plotted. We do this on lines 9 and 10 with

```
x = arange(Xmin, Xmax, DelX)
```
Form x array in range with increment

```
y = -sin(x)*cos(x)
```
Form y array using x array

As you see, creating arrays for plotting is easy with the numpy method **arange** as it constructs an array of floating point numbers between **Xmax** and **Xmin** in steps of **DelX**. For integer arrays use the **range** method. And since **x** is an array, **y = sin(x)** is automatically one too! Note next in Listing 3.1 how the **plot** command places the points on the plot. A '-' is used to indicate a line and (**lw**=2) sets it width. The **xlabel** and **ylabel** commands label the axis, and the **text** and **title** commands place a title on top of the plot and one along with x axis. The **show** command produces a graph on your desktop. Other commands are given in the table below.

[6] A ufunction is a function that operates on N-D arrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. In other words, a ufunc is a vectorized wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.

Matplotlib Command	Effect	Matplotlib Command	Effect
plot(x, y, '-', lw=2)	x-y curve, line width 2pts	myPlot.setYRange(-8., 8.)	Set y range
show()	Show output graph	myPlot.setSize(500, 400)	Plot size in pixels
xlabel('x')	x axis label	pyplot.semilogx	Semilog x plot
ylabel('f(x)')	y axis label	pyplot.semilogy	Semilog y plot
title('f vs x')	Add title	grid(True)	Draw grid
text(x, y, s)	Add text s at (x, y)	myPlot.setColor(false)	Black & White
myPlot.addPoint(0,x,y,true)	Add (x, y) to set 0, connect	myPlot.setButtons(true)	For zoom button
myPlot.addPoint(1,x,y, false)	Add (x, y) to set 1, no connect	myPlot.fillPlot( )	Fir ranges to data
pyplot.errorbar	Point + error bar	myPlot.setImpulses(true,0)	Vert lines, set 0
pyplot.clf()	Clear current figure	pyplot.contour	Contour lines
pyplot.scatter	Scatter plot	pyplot.bar	Bar charts
pyplot.polar	Polar plot	pyplot.gca	For current axis
myPlot.setXRange(-1., 1.)	Set x range	pyplot.acorr	autocorrelation plot

Listing 3.1 The program **EasyMatPlot.py** produces a , 2-D x-y plot using the `matplotlib` package (which includes the numpy package).

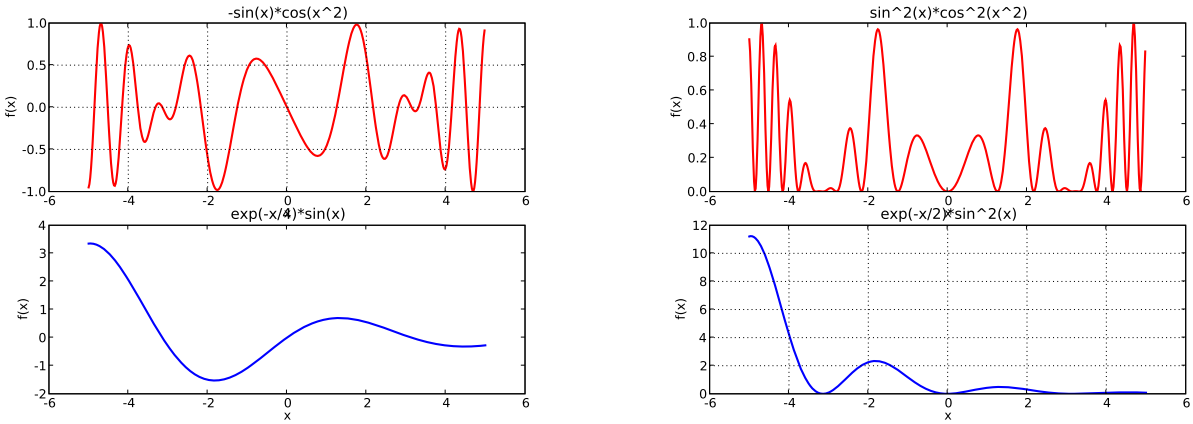
```
# EasyMatPlot.py:    Simple use of matplotlib's plot command

from pylab import *                                    # Load matplotlib

Xmin = -5.0;        Xmax = +5.0;      Npoints= 500
DelX= (Xmax-Xmin)/Npoints                                    # Delta x
x = arange(Xmin, Xmax, DelX)                        # x range + increment
y =  sin(x)*sin(x*x)                                  # Function to plot

print ('arange => x[0], x[1],x[499]=%8.2f %8.2f %8.2f' %(x[0],x[1],x[499]) )
print ('arange => y[0], y[1],y[499]=%8.2f %8.2f %8.2f' %(y[0],y[1],y[499]) )
print ("\n Now doing the plotting thing, look for Figure 1 on desktop"       )

xlabel('x');      ylabel('f(x)');     title(' f(x) vs x')        # labels
text(-1.5,  0.1, 'MatPlotLib Example')                      # Text on plot
plot(x, y, '-', lw=2)
grid(True)                                                     # Form grid

show()                                                 # Make screen plot
```

Listing 3.2 The program **GradesMatPlot.py** produces a , 2-D x-y plot using the `matplotlib` package.

```
# GradesMatPlot.py: Using Matplotlib's plot command with multi data sets & curves

import pylab as p                    # Matplotlib, pymode for power users
from numpy import *

p.title('Grade Inflation')                             # Title and labels
p.xlabel('Years in College')
p.ylabel('GPA')

xa = array([-1, 5])                                 # For horizontal line
ya = array([0, 0])                                       # Array of zeros
p.plot(xa, ya)                                     # Draw horizontal line

x0 = array([0, 1, 2, 3, 4])                            # Data set 0 points
y0 = array([-1.4, +1.1, 2.2, 3.3, 4.0])
p.plot(x0, y0, 'bo')                          # Data set 0 = blue circles
p.plot(x0, y0, 'g')                                    # Data set 0 = line

y1 = array([4.0, 2.7, -1.8, -0.9, 2.6])                # Data set 1 points
t = arange(0, 5, 1)
p.plot(t, y1, 'r')

err1sup = array([1.0, 0.3, 1.2, 0.4, 0.1])        # Asymmetric error bars
err1inf = array([2.0, 0.6, 2.3, 1.8, 0.4])
p.errorbar(t, y1, [err1inf, err1sup], fmt = 'o')         # Plot error bars

p.grid(True)                                                   # Grid line
p.show()                                           # Create plot on screen
```

As we see in **`GradesMatplot.py`** Listing 3.2 and Figure 3.1 Right, the **`plot`** command can also plot a number of data sets on one graph as well as different curves. We start on line 3 of by

Figure 3.2 *Left* and *Right* columns show two separate outputs, each of two figures, produced by `MatPlot2figs.py`. (We used the slider button to add some space between the red and blue plots.)



importing the Visual package so we can use the **`array`** command, while on line 4 we import Matplotlib (**`pylab`**) for our plotting routines. Since we have imported two packages, we add the **`pylab`** prefix to the plot commands so that Python knows which package to use. In order to place a horizontal line along $y = 0$, on line 10 and 11 we use the **`array`** command to set up an array of x values with $-1 \leq x \leq 5$, a corresponding array of y values $y \equiv 0$, and then plot these arrays (line 12). Next we go about placing four more curves in this figure. First on lines 14–16 we create a data set 0 as blue points and then connect them with a green line. Second on lines 19–21 we plot up a different data set as a red line. Finally, on lines 23–25 we define unequal lower and upper error bars and have the **`plot`** command add them to data set 1 and plot them. We then add grid lines and show the plot on the screen.

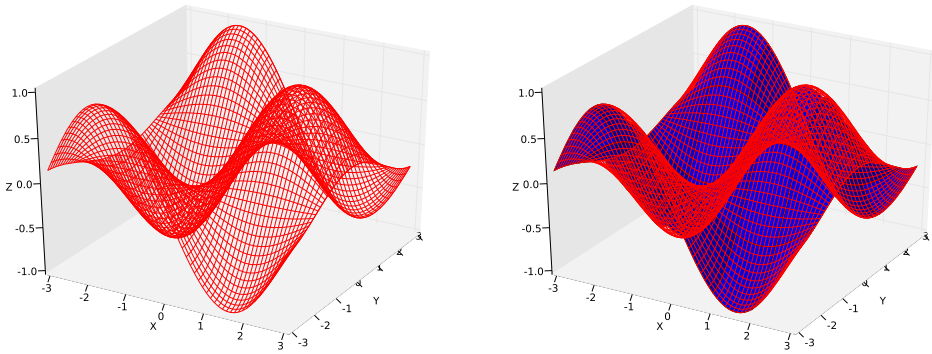
Often the science is clearer if there are several curves in one plot and several plots in one figures. Matplotlib lets you do this with the **`plot`** and the **`subplot`** commands. For example, in **`MatPlot2figs.py`** in Listing 3.3 and Figure 3.2 we have placed two curves in one plot, and then output two different figures, each containing two plots. The key here is repetition of the **`subplot`** command:

`figure(1)`	The 1st figure
`subplot(2,1,1)`	2 rows, 1 column, 1st subplot
`subplot(2,1,2)`	2 rows, 1 column, 2nd subplot

Listing 3.3 The program **MatPlot2figs.py** produces the two figures shown in Figure 3.2, with each having two plots created by `matplotlib`.

```
# MatPlot2figs.py: plot of 2 subplots on 1 fig & 2 separate figs

from pylab import *                                    # Load matplotlib

Xmin = -5.0;      Xmax =  5.0;        Npoints= 500
DelX= (Xmax-Xmin)/Npoints                                  # Delta x
x1 = arange(Xmin, Xmax, DelX)                             # x1 range
x2 = arange(Xmin, Xmax, DelX/20)                   # Different x2 range
y1 =  -sin(x1)*cos(x1*x1)                                # Function 1
y2 =   exp(-x2/4.)*sin(x2)                               # Function 2
print("\n Now plotting, look for Figures 1 & 2 on desktop")
#        Figure 1
figure(1)
subplot(2,1,1)                              # 1st subplot in first figure
plot(x1, y1, 'r', lw=2)
xlabel('x');      ylabel( 'f(x)' );      title( '-sin(x)*cos(x^2)' )
grid(True)                                                # Form grid
subplot(2,1,2)                              # 2nd subplot in first figure
plot(x2, y2, '-', lw=2)
```

Figure 3.3 A 3-D wire frame *Left* and a surface plot with wire frame *Right*. Both are produced by `SurfacePlot.py` using Matplotlib.




```
xlabel('x')                                                    # Axes labels
ylabel( 'f(x)' )
title( 'exp(-x/4)*sin(x)' )

#        Figure 2
figure(2)
subplot(2,1,1)                                        # 1st subplot in 2nd figure
plot(x1, y1*y1, 'r', lw=2)
xlabel('x');        ylabel( 'f(x)' );     title( 'sin^2(x)*cos^2(x^2)' )

     # form grid
subplot(2,1,2)                                        # 2nd subplot in 2nd figure
plot(x2, y2*y2, '-', lw=2)
xlabel('x');        ylabel( 'f(x)' );      title( 'exp(-x/2)*sin^2(x)' )
grid(True)

show()                                                         # Show graphs
```

3.4.3 Matplotlib for 3-D Surface Plots

A 2-D plot of the potential $V(r) = 1/r$ *versus* r is fine for visualizing the potential field surrounding a single charge. However, to visualize the same potential as a function of the Cartesian coordinates x and y, $V(x, y) = 1/\sqrt{x^2 + y^2}$, we need a 3-D visualization. We get that by creating a world in which the z dimension (mountain height) is the value of the potential, and x and y lie on a flat plane below the mountain. Because the surface we are creating is a 3-D object, it is not possible to draw it on a flat screen, and so different techniques are used to give the impression of three dimensions to our brains. We do that by rotating the object, shading it, employing parallax, and other tricks.

Listing 3.4 The program **SurfacePlot.py** produces the 3-D/surface plots in Figure 3.3 by using Matplotlib commands. SurfacePlot.py

```
# SurfacePlot.py: matplotlib 3D plot you can rotate and scale via mouse

import matplotlib.pylab  as p
from mpl_toolkits.mplot3d import Axes3D
from visual import *    #for arange

print("Please be patient, I have packages to import & points to plot")
delta = 0.1
x = arange( -3., 3., delta )
y = arange( -3., 3., delta )
X, Y = p.meshgrid(x, y)
Z = sin(X)*cos(Y)                                    #  surface height
```

```
fig = p.figure()                                        # create figure
ax = Axes3D(fig)                                          # plots axes
# ax.plot_surface(X, Y, Z)                                   # surface
ax.plot_wireframe(X, Y, Z, color = 'r')                    # wireframe
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

p.show()                                                  # show figure
```

In Figure 3.3 we show a wire-frame plot (left) and a surface-plus-wire-frame plot (right). These are obtained from the program **`SurfacePlot.py`** in Listing **??**. Lines 8 and 9 are the standard creation of x and y arrays of floats using **`arange`**. Line 10 uses Matplotlib's **`meshgrid`** method to set up the grid of x and y values needed for the surface plot. The rest of the program is self-explanatory with **`fig`** being the plot object, **`ax`** being the 3-D axes object (the library **`axes3d`** is part of the Mmatplotlib extension for 3-D), and **`plot_wireframe`** and **`plot_surface`** creating wire frame and surface plots respectively.

3.4.4 Matplotlib Exercises

We encourage you to make your own plots and personalize them to your own likings by trying out other commands and by including further options in the commands. The Matplotlib documentation is extensive and available on the Web. As an exercise, explore:

1. how to zoom in and zoom out on sections of a plot,
2. how to save your plots in various formats,
3. how to print up your graphs,
4. how to pan through your graphs,
5. the effects obtained by adjusting the plot parameters available from the pull-down menu,
6. how to increase the space between subplots,
7. and for surfaces, explore how to rotate and scale the surfaces.

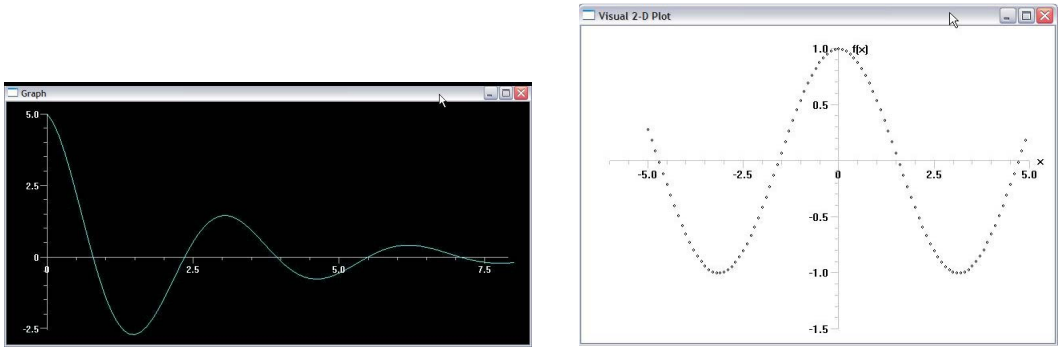
Listing 3.5 The program **EasyVisual.py** produces a 2-D $x - y$ plot using the *Visual* package.

```
# EasyVisual.py:        Simple graph object using Visual

from visual.graph import *                                # Import Visual

funct1 = gcurve(color = color.cyan)                # Connected curve object

for x in arange(0., 8.1, 0.1):                                  # x range
    funct1.plot( pos = (x, 5.*cos(2.*x)*exp(-0.4*x)) )       # Plot points

graph1 =  gdisplay(x = 0, y = 0, width = 600, height = 450, # Second plot
          title = 'Visual 2-D Plot', xtitle = 'x', ytitle = '  f(x)',
          xmax = 5., xmin = - 6., ymax = 1, ymin = - 1.5,
          foreground = color.black, background = color.white)

plotObj = gdots(color = color.black)                              # Dots

for x in arange( -5.,  +5, 0.1 ):
    plotObj.plot(pos = (x, cos(x)))
```

3.5 2-D PLOTS WITH VISUAL (VPYTHON)

As we have already indicated, the Python Visual package is often used to create 3-D demonstrations and animations that are useful for realistic visualizations. In VPython there are several ways for making nice two dimensional graphs and animations. In Figure 3.4 we present two

Figure 3.4 Screen dumps of x-y plots produced by the Visual package in Listing `EasyVisual.py`. The *left* plot uses default parameters while the *right* plot uses user-supplied options.



separate plots produced by the program **`EasyVisual.py`** in Listing 3.5. This uses the command **`gdisplay`** to input information about screen coordinates (in pixels) and the world coordinates (in the math units). The **`x=0, y=0`** arguments places the graph window at the upper left corner of the screen, and the **`width=600, height =300`** arguments creates a graph window that is 600 pixels wide and 300 pixels high. Actually, VPython is so user friendly that you can omit **`xmin, xmax, ymin`** and **`ymax`** and let VPthon figure this out for you from the input data.

As you peruse Listing 3.5 observe that the technique for plotting with Visual is quite different from that used by Matplotlib. While the latter first creates arrays of x and y values and then does all the plotting in one fell swoop, this Visual program creates plot objects and then adds the points to them, one-by-one. We see in Listing 3.5 that the first plot in Figure 3.4 is created using default plotting parameters to produce a connected curve (**`gcurve`**), while in the second plot we graph just the points (**`gdots`**), but with axes labels and a title.

Listing 3.6 The program **3GraphVisual.py** produces a , 2-D x-y plot using the `matplotlib` and `numpy`packages.

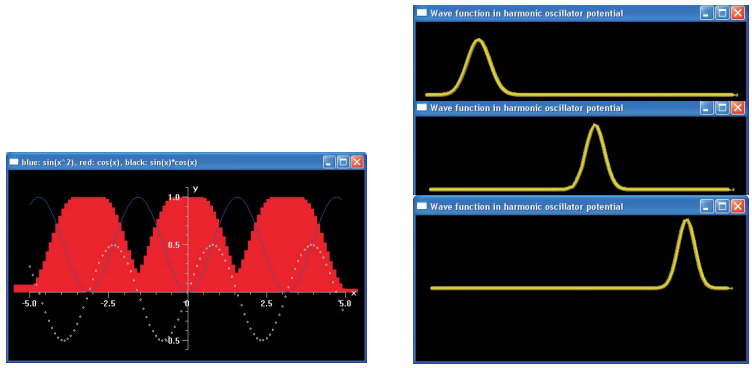
```
# 3GraphVisual.py: 3 plots in the same figure, with bars, dots and curve

import visual.graph as vg

string = "blue: sin(x^2), red: cos(x), black: sin(x)*cos(x)"
graph1 = vg.gdisplay(x=0, y=0, width=600, height=350, title=string,
            xtitle='x', ytitle='y', xmax=5.,xmin=-5., ymax=1.1, ymin=-0.6)
#    curve, vertical bars, dots
y1 = vg.gcurve(color=vg.color.blue, delta=3)
y2 = vg.gvbars(color=vg.color.red)
y3 = vg.gdots(color=vg.color.white, delta=3)

for x in vg.arange(-5, 5, 0.1):                    # arange for floats
    y1.plot( pos=(x, vg.sin(x)*vg.sin(x)) )
    y2.plot( pos=(x, vg.cos(x)*vg.cos(x)) )
    y3.plot( pos=(x, vg.sin(x)*vg.cos(x)) )
```

As we already discussed when describing Matplotlib, it is often a good idea to place several plots in the same figure. We give an example of this in **`3GraphVisual.py`** (Listing 3.6) and Figure 3.5 left. We see here red vertical bars (**`gvbars`**), dots (**`gdots`**), and a curve (**`gcurve`**). Also note in **`3GraphVisual.py`** that we avoid long prefixes to the commands by starting the program with

```
import visual.graph as vg
```

This both imports Visual's graphing package and assigns the symbol **`vg`** to **`visual.graph`**.

Figure 3.5 *Left:* Output graph from the program `3GraphVisual.py` that produces a , 2-D x-y plot using the `matplotlib` package. *Right* Three frames from the animation of a quantum mechanical wave packet bound within a harmonic oscillator potential that was produced with the program `HarmosAnimate.py`.



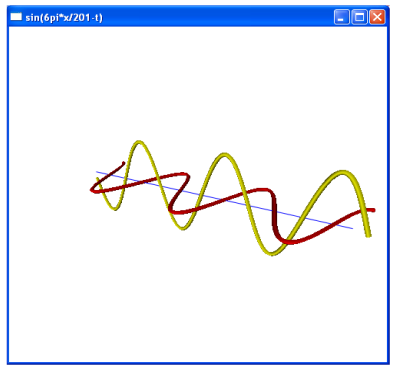
Listing 3.7 The program **HarmosAnimate.py** solves the time-dependent Schrödinger equation for a probability wave packet and creates an animation with Visual.

```
#  HarmosAnimate: Soltn of Sch Eqt for HO with animation

from visual import *

# initialize wave function, probability, potential
dx = 0.04;    dx2 = dx*dx;  k0 = 5.5*pi;  dt = dx2/20.0;  xmax = 6.0
xs = arange(-xmax,xmax+dx/2,dx)              # array of x positions

g=display(width=500,height=250,title='Wave Packet in HO Potential')
PlotObj= curve(x=xs, color=color.yellow, radius=0.1)
g.center = (0,2,0)                                    # center of scene
                                          # initial condition; wave packet
psr = exp(-0.5*(xs/0.5)**2) * cos(k0*xs)            # Re wave function Psi
psi = exp(-0.5*(xs/0.5)**2) * sin(k0*xs)            # Im wave function Psi
v = 15.0*xs**2

# t = 0
# while t < 0.85:                                  # Propage while in well
while True:
   rate(500)
   psr[1:-1] = psr[1:-1] - (dt/dx2)*(psi[2:]+psi[:-2]-2*psi[1:-1]) +dt*v[1:-1]*psi[1:-1]
   psi[1:-1] = psi[1:-1] + (dt/dx2)*(psr[2:]+psr[:-2]-2*psr[1:-1]) -dt*v[1:-1]*psr[1:-1]
   PlotObj.y = 4*(psr**2 + psi**2)
   # t += dt
```

Listing 3.8 The program **3Danimate.py** creates a 3D animation of an E & M wave.

```
# 3Danimate.py:  3-D animation of EM Wave
# Plots sin and cos waves perpendicular to each other

from visual import *                          # import graphics routines

xmax = 201                                       # number of x,y points
scene = display(x=0, y=0, width= 500, height= 500,
        title= 'sin(6pi*x/201-t)', background=(1.,1.,1.0),
        forward=(-0.6,-0.5,-1))            # White background, tilted view
sinWave = curve(x=range(0,xmax), color=color.yellow, radius=4.5)
cosWave = curve(x=range(0,xmax), color=color.red, radius=4.5)
Xaxis   = curve(pos=[(-300,0,0),(300,0,0)], color=color.blue)
incr = math.pi/xmax                                          # x increment
for t in arange(0, 10, 0.02):                                  # time loop
    for i in range(0, xmax):                          # loop through x values
        x = i*incr                                     # x world coordinates
        f  = math.sin(6.0*x-t)                                   # sin wave
        zz = math.cos(6.0*x-t)                                   # cos wave
        yp = 100*f                                 #  f to y axis screen coords
        xp = 200*x-300                                 # TF x to screen coords
        zp = 100*zz                                   # TF z to screen coords
        sinWave.x[i] = xp                                  # plot x component
        sinWave.y[i] = yp                                 # plot  y component
        cosWave.x[i] = xp                             # x coord of z function
```

Figure 3.6 A screen shot of a 3-D animation produced with the program 3Danimate.py.




```
        cosWave.z[i] = zp                          # z coord of z function
```

3.6 ANIMATIONS WITH VISUAL (VPYTHON)

Creating animations with Visual is essentially just making the same 2-D plot over and over again at slightly different times, and plotting them on top of each other. If done properly this gives the impression of motion. We give a number of sample codes that produce animation in the **`PythonCodes`** directory and an example in Listing 3.7 and with the three frames in Figure 3.5. (The code **`HarmosAnimate.py`** in Listing 3.7 comes from Chapter 18 in which we describe methods for solving the time-dependent Schrödinger equation.) Notice how lines 10 and 11 set up the plot, with **`PlotObj`** representing (what else?) a plot object. Lines 29 and 30 are within a loop that updates the points, with line 33 (still within the loop) plotting the points. Since this is done continually (every 10 **`counts`**), the replotting appears as an animation. Note that being able to plot points individually without having to store them all in an array for all times keep the memory demand of the program quite reasonable.

As a somewhat more complex, but striking, example in listing 3.8 we present a short program that creates a 3-D animation of an electromagnetic wave. It does this by plotting $\sin(6\pi x/201 - t)$ along the y axis and $\cos(6\pi x/201 - t)$ along the z axis, both as functions of time. (In §18.9 we run a real E&M simulation.) Figure **??** shows a screen shot of the the program's output which hardly does justice to the animation. The animation is produced by plotting both functions at different instances of time and then applying the **`display`** and **`curve`** methods.

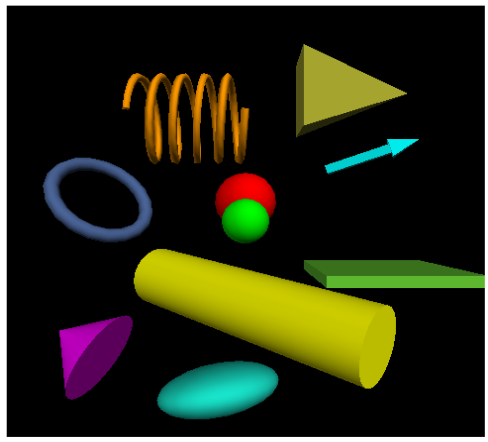
There are two features worth noting in the program. The first is the conversion from the actual x and y values to screen coordinates that will fit into the box we have created on the screen for the figure. The second is the **`forward=(-0.6,-0.5,-1)`** command which sets the viewing angle so that we get a view of all three axes.

3.6.1 3-D Objects in Vpython

Listing 3.9 The program **3Dshapes.py** that produces a sample of 3-D shapes available with VPython.

```
# Some 3-D Shapes of VPython

from visual import *
```

Figure 3.7 Some of the 3-D shapes that can be created with single commands in VPython.



```
graph1 = display(width=500, height=500, title='VPython 3-D Shapes', range=10)
sphere(pos=(0,0,0), radius=1, color=color.green)
sphere(pos= (0,1,-3), radius=1.5, color=color.red)
arrow(pos=(3,2,2), axis=(3,1,1), color=color.cyan)
cylinder(pos=(-3,-2,3), axis=(6,-1,5), color=color.yellow)
cone(pos=(-6,-6,0), axis=(-2,1,-0.5), radius=2, color=color.magenta)
helix(pos=(-5,5,-2), axis=(5,0,0), radius=2, thickness=0.4, color=color.orange)
ring(pos=(-6,1,0), axis=(1,1,1), radius=2, thickness=0.3, color=(0.3,0.4,0.6))
box(pos=(5,-2,2), length=5, width=5, height=0.4, color=(0.4,0.8,0.2))
pyramid(pos=(2,5,2), size=(4,3,2), color=(0.7,0.7,0.2))
ellipsoid(pos=(-1,-7,1), axis=(2,1,3), length=4, height=2, width=5, color=(0.1,0.9,0.8))
```

One way to make simulations appear more realistic is to use 3-D shapes, for example, a sphere for a bouncing ball. VPython can produce a variety of 3-D shapes with one-line commands, as shown in Fig. 3.7 and produced by the code in Listing 3.9. To make the ball bounce you would need to vary the **`position`** variable according to some kinematic equations.

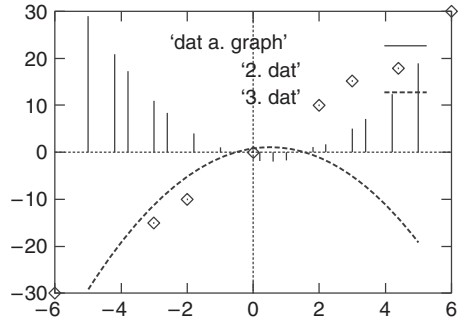
3.7 GNUPLOT: RELIABLE 2-D AND 3-D PLOTS

Gnuplot is a versatile 2-D and 3-D graphing package that makes Cartesian, polar, surface, and contour plots. Gnuplot is classic open software, available free on the Web, and supports many output formats.

Begin Gnuplot with a file of $(x,\ y)$ data points, say, **`graph.dat`**. Next issue the **`gnuplot`** command from a shell or from the Start menu. A new window with the Gnuplot prompt **`gnuplot>`** should appear. Construct your graph by entering Gnuplot commands at the Gnuplot prompt or by using the pull-down menus in the Gnuplot window:

`> gnuplot`	Start Gnuplot program
`Terminal type set to 'x11'`	Type of terminal for Unix
`gnuplot>`	The Gnuplot prompt

Figure 3.8 A Gnuplot graph for three data sets distinguished by the use of impulses and lines.




```
gnuplot>  plot "graph.dat"
```
Plot data file graph.dat

Plot a number of graphs on the same plot using several data files (Figure 3.8):

```
gnuplot>  plot 'graph.dat' with impulses, '2.dat', '3.dat' with lines
```

The general form of the 2-D plotting command and its options are

plot {**ranges**} ***function*** {**title**} {**style**} {, ***function*** . . . } — Command

with points	Default. Plot a symbol at each point.
with lines	Plot lines connecting the points.
with linespoint	Plot lines and symbols.
with impulses	Plot vertical lines from the x axis to points.
with dots	Plot a small dot at each point (scatterplots).

For Gnuplot to accept the name of an external file, that name must be placed in 'single' or "double" quotes. If there are multiple file names, the names must be separated by commas. Explicit values for the x and y ranges are set with options:

```
gnuplot> plot [xmin:xmax] [ymin:ymax] "file"
```
Generic

```
gnuplot> plot [--10:10] [--5:30] "graph.dat"
```
Explicit

3.7.1 Gnuplot Input Data Format ⊙

The format of the data file that Gnuplot can read is not confined to (x, y) values. You may also read in a data file as a C language *scanf* format string xy by invoking the **using** option in the **plot** command. (Seeing that it is common for Linux/ Unix programs to use this format for reading files, you may want to read more about it.)

plot '**datafile**' { **using** { **xy** | **yx** | **y** } {"**scanf string**"} }

This format explicitly reads selected rows into x or y values while skipping past text or unwanted numbers:

```
gnuplot>  plot "data" using "%f%f"
```
Default, 1st x, 2nd y.

```
gnuplot>  plot "data" using yx "%f %f"
```
Reverse, 1st y, 2nd x.

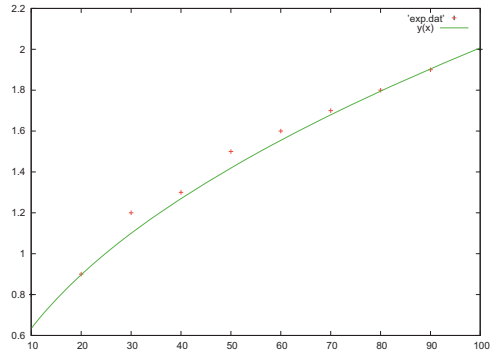
```
gnuplot>  plot "data" xy using "%*f %f %*f %f"
```
Use row 2,4 for x, y.

```
gnuplot>  plot "data" using xy "%*6c %f%*7c%f"
```

This last command skips past the first six characters, reads one x, skips the next seven characters, and then reads one y. It works for reading in x and y from files such as

```
theta: −20.000000 Energy: −3.041676 theta: −19.000000 Energy:
```

Figure 3.9 A Gnuplot plot of data from a file along with the plot of an analytic function.




```
−3.036427 theta: −18.000000 Energy: −3.030596 theta: −17.000000
Energy: −3.024081 theta: −16.000000 Energy: −3.016755
```

Observe that because the data read by Gnuplot are converted to floating-point numbers, you use `%f` to read in the values you want.

Besides reading data from files, Gnuplot can also generate data from user-defined and library functions. In these cases the default independent variable is x for 2-D plots and (x, y) for 3-D ones. Here we plot the acceleration of a nonharmonic oscillator:

`gnuplot> k = 10`	Set value for k
`gnuplot> a(x) = .5*k*x**2`	Analytic expression
`gnuplot> plot [--10:10] a(x)`	Plot analytic function

A useful feature of Gnuplot is its ability to plot analytic functions along with numerical data. For example, Figure 3.9 compares the theoretical expression for the period of a simple pendulum to experimental data of the form

```
# length(cm)        period (sec)
10  0.8
20  0.9
30  1.2
40  1.3
50  1.5
60  1.6
70  1.7
80  1.8
90  1.9
100 2.0
```

Note that the first line of text is ignored since it begins with a # . We plot with

`gnuplot> g = 980`	Set value for g
`gnuplot> y(x) = 2*3.1416*sqrt(x/g)`	Period $T = y$, length $L = x$
`gnuplot> plot "exp.data", y(x)`	Plot both data and function

3.7.2 Printing Plots

Gnuplot supports a number of printer types including PostScript. The safest way to print a plot is to save it to a file and then print the file:

1. Set the “terminal type” for your printer.

2. Send the plot output to a file.
3. Replot the figure for a new output device.
4. Quit Gnuplot (or get out of the Gnuplot window).
5. Print the file with a standard print command.

For a more finished product, you can import Gnuplot's output `.ps` file into a drawing program, such as CorelDraw or Illustrator, and fix it up just right. To see what types of printers and other output devices are supported by Gnuplot, enter the **`set terminal`** command without any options in a **`gnuplot`** shell. Here is an example of creating and printing a PostScript figure:

```
gnuplot> set terminal postscript          Choose local printer type
Terminal type set to 'postscript'         Gnuplot response
gnuplot> set term postscript eps          Another option
gnuplot> set output "plt.ps"              Send figure to file
gnuplot> replot                           Plot again so file is sent
gnuplot> quit                             Or get out of gnu window
% lp plt.ps                               Unix print command
```

3.7.3 Gnuplot Surface (3-D) Plots

A 2-D plot is fine for visualizing the potential field $V(r) = 1/r$ surrounding a single charge. However, when the same potential is expressed as a function of Cartesian coordinates, $V(x, y) = 1/\sqrt{x^2 + y^2}$, we need a 3-D visualization. We get that by creating a world in which the z dimension (mountain height) is the value of the potential and x and y lie on a flat plane below the mountain. Because the surface we are creating is a 3-D object, it is not possible to draw it on a flat screen, and so different techniques are used to give the impression of three dimensions to our brains. We do that by rotating the object, shading it, employing parallax, and other tricks.

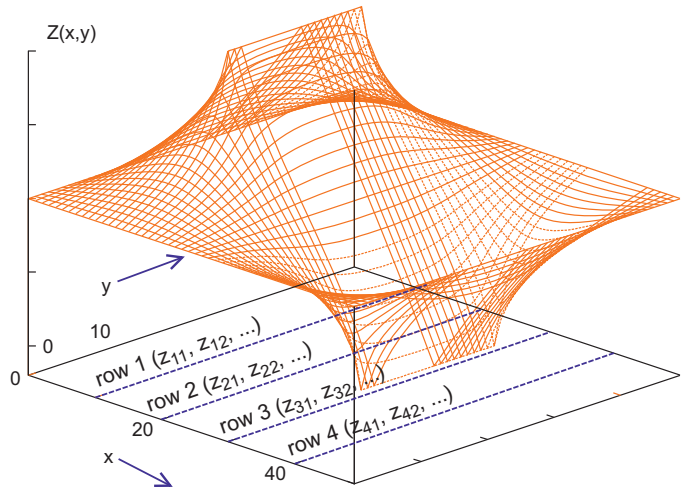
In Gnuplot, the surface (3-D) plot command is **`splot`**, and it is used in the same manner as **`plot`**—with the obvious extension to (x, y, z). A surface (Figure 3.10) is specified by placing the $z(x, y)$ values in a matrix but without ever giving the x and y values explicitly. The x values are the row numbers in the matrix and the y values are the column values (Figure 3.10). This means that only the z values are read in and that they are read in row by row, with different rows separated by blank lines:

row 1 (blank line) row 2 (blank line) row 3 ... row *N*.

Here each row is input as a column of numbers, with just one number per line. For example, 13 columns each with 25 z values would be input as a sequence of 25 data elements, followed by a blank line, and then another sequence followed by a blank line, and so on:

```
0.0
0.695084369397148
1.355305208363503
1.9461146066793003
...
-1.0605832625347442
-0.380140746321537
[blank line]
0.0
0.6403868757235301
1.2556172093991282
...
2.3059977070286473
2.685151549102467
  [blank line]
2.9987593603912095
...
```

Figure 3.10 A surface plot $z(x, y) = z$ (row, column) showing the format of the input data. Only z values are input, with the values of row and column indices used as x and y values. The input data has successive rows separated by blank lines with the column values repeating for each row.



Although there are no explicit x and y values given, Gnuplot plots the data with the x and y assumed to be the row and column numbers.

Versions 4.0 and above of Gnuplot have the ability to rotate 3-D plots interactively. You may also adjust your plot with the command

```
gnuplot>  set view rotx, rotz, scale, scalez
```

where $0 \leq$ **rotx** $\leq 180°$ and $0 \leq$ **rotz** $\leq 360°$ are angles in degrees and the scale factors control the size. Any changes made to a plot are made when you redraw the plot using the **replot** command.

To see how this all works, here we give a sample Gnuplot session that we will use in Chapter 17, "PDEs for Electrostatics & Heat Flow," to plot a 3-D surface from numerical data. The program **Laplace.java** contains the actual code used to output data in the form for a Gnuplot surface plot.[7]

```
> gnuplot                                      Start Gnuplot system from a shell
gnuplot> set hidden3d                          Hide surface whose view is blocked
gnuplot> set nohidden3d                        Show surface though hidden from view
gnuplot> splot 'Laplace.dat' with lines        Surface plot of Laplace.dat with lines
gnuplot> set view 65,45                        Set x and y rotation viewing angles
gnuplot> replot                                See effect of your change
gnuplot> set contour                           Project contours onto xy plane
gnuplot> set cntrparam levels 10               10 contour levels
gnuplot> set terminal epslatex                 Output in Encapsulated PostScript for LaTeX
gnuplot> set terminal PostScript               Output in PostScript format for printing
gnuplot> set output "Laplace.ps"               Plot output to be sent to file Laplace.ps
gnuplot> splot 'Laplace.dat' w l               Plot again, output to file
gnuplot> set terminal x11                      To see output on screen again
```

[7] Under Windows, there is a graphical interface that is friendlier than the Gnuplot subcommands. The subcommand approach we indicate here is reliable and universal.

```
gnuplot> set title 'Potential V(x,y) vs x,y'        Title graph
gnuplot> set xlabel 'x Position'                    Label x axis
gnuplot> set ylabel 'y Position'                    Label y axis
gnuplot> set zlabel 'V(x,y)'; replot                Label z axis and replot
gnuplot> help                                       Tell me more
gnuplot> set nosurface                              Do not draw surface; leave contours
gnuplot> set view 0, 0, 1                           Look down directly onto base
gnuplot> replot                                     Draw plot again; may want to write to file
gnuplot> quit                                       Get out of Gnuplot
```

3.7.4 Gnuplot Vector Fields

Even though it is simpler to compute a scalar potential than a vector field, vector fields often occur in nature. In Chapter 17, "PDEs for Electrostatics & Heat Flow," we show how to compute the electrostatic potential $U(x, y)$ on an $x + y$ grid of spacing Δ. Since the field is the negative gradient of the potential, $\mathbf{E} = -\vec{\nabla} U(x, y)$, and since we solve for the potential on a grid, it is simple to use the central-difference approximation for the derivative (Chapter 7 "Differentiation & Searching") to determine $\mathbf{E}$:

$$E_x \simeq \frac{U(x+\Delta, y) - U(x-\Delta, y)}{2\Delta} = \frac{U_{i+1,j} - U_{i-1,j}}{2\Delta}, \tag{3.1}$$

$$E_y \simeq \frac{U(x, y+\Delta) - U(x, y-\Delta)}{2\Delta} = \frac{U_{i,j+1} - U_{i,j-1}}{2\Delta}. \tag{3.2}$$

Gnuplot contains the **vectors** style for plotting vector fields as arrows of varying lengths and directions (Figure 3.11).

```
> plot 'Laplace_field.dat' using 1:2:3:4 with vectors        Vector plot
```

Here **Laplace_field.data** is the data file of **(x, y, Ex, Ey)** values, the explicit columns to plot are indicated, and additional information can be provided to control arrow types. What Gnuplot actually plots are vectors from (x, y) to $(x + \Delta x, y + \Delta y)$, where you input a data file with each line containing the $(x, y, \Delta x, \Delta y)$ values. Thousands of tiny arrows are not very illuminating (Figure 3.11 left), nor are overlapping arrows. The solution is to plot fewer points and larger arrows. On the right in Figure 3.11 we plot every fifth point normalized to unit length via

$$\Delta x = \frac{E_x}{N}, \quad \Delta y = \frac{E_y}{N}, \quad N = \sqrt{E_x^2 + E_y^2}. \tag{3.3}$$

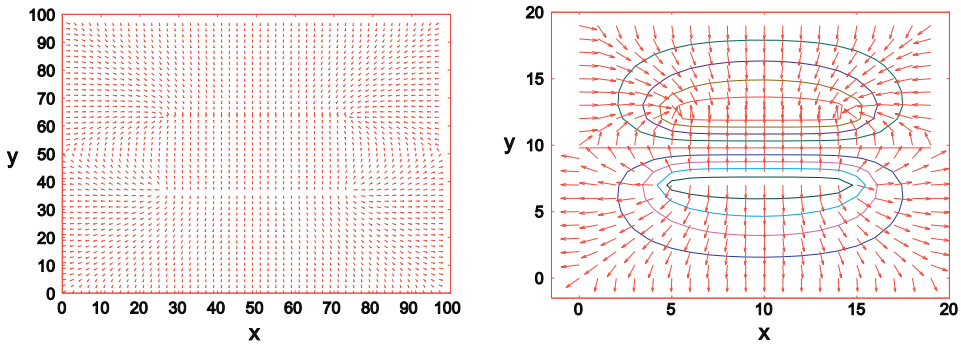
The data file was produced just by adding the lines

```
Ex = -( U[i+1,j] - U[i-1,j] )          // Compute field components
Ey = -( U[i,j+1] - U[i,j-1] )
```

And here are some commands that add contour lines to the field plots:

```
gnuplot> unset key
gnuplot> set nosurface
gnuplot> set contour base
gnuplot> set cntrparam levels 10
gnuplot> set view 0,0,1,1
gnuplot> splot 'Laplace_pot1.dat' with lines
gnuplot> set terminal push
gnuplot> set terminal table
```

Figure 3.11 Two different visualizations by Gnuplot of the same electric field of a parallel plate capacitor. The figure on the right includes equipotential surfaces and uses one-fifth as many field points, and longer vectors, but of constant length. The x and y values are the column and row indices.




```
gnuplot> set out 'equipot.dat'
gnuplot> replot
gnuplot> set out
gnuplot> set terminal pop
gnuplot> reset
gnuplot> plot 'equipot.dat' with lines
gnuplot> unset key
gnuplot> plot 'equipot.dat' with lines, 'Laplace_field1.dat' with vectors
```

By setting **terminal** to **table** and setting **out** to **equipot.dat**, the numerical data for equipotential lines are saved in the file **equipot.dat**. This file can then be plotted together with the vector field lines.

3.7.5 Animations from Gnuplot⊙

An *animation* is a collection of images called *frames* that when viewed in sequence convey the sensation of continuous motion. It is an excellent way to visualize the time behavior of a simulation or of a function $f(\mathbf{x}, t)$, as may occur in wave motion or heat flow. We recommend that you examine several of the animation links at the beginning of many of our chapters and judge for yourself how well they work.

Gnuplot itself does not create animations. However, you can use it to create a sequence of frames and then concatenate the frames into a movie. Here we create an animated *gif* that, when opened with a Web browser, automatically displays the frames in a rapid enough sequence for your mind's eye to see them as a continuous event. Although Gnuplot does not output **.gif** files, we outputted *pixmap* files and converted them to gif's. Because a number of commands are needed for each frame and because hundreds or thousands of frames may be needed for a single movie, we wrote the script **MakeGifs.script** to automate the process. (A *script* is a file containing a series of commands that might normally be entered by hand to execute within a shell. When placed in a script, they are executed as a single command.) The script in Listing 3.10, along with the file **samp_color**, should be placed in the directory containing the data files (**run.lmn** in this case).

The #! line at the beginning tells the computer that the subsequent commands are in the korn shell. The symbol **$i** indicates that **i** is an argument to the script. In the present case, the script is run by giving the name of the script with three arguments, (1) the beginning time, (2)

the maximum number of times, and (3) the name of the file where you wish your gifs to be stored:

```
% MakeGifs.script 1 101 OutFile
```
Make gif from times 1 to 101 in `OutFile`

The > symbol in the script indicates that the output is directed to the file following the >. The **ppmquant** command takes a pixmap and maps an existing set of colors to new ones. For this to work, you must have the map file **samp_colormap** in the working directory. Note that the increments for the counter, such as **i=i+99**, should be adjusted by the user to coincide with the number of files to be read in, as well as their names. Depending on the number of frames, it may take some time to run this script. Upon completion, there should be a new set of files of the form **OutfileTime.gif**, where **Outfile** is your chosen name and **Time** is a multiple of the time step used. Note that you can examine any or all of these gif files with a Web browser.

The final act of movie making is to merge the individual gif files into an animated gif with a program such as **gifmerge**:

```
> gifmerge --10 *.gif > movie
```
Merge all .gif files into movie

Listing 3.10 **MakeGifs.script**, a script for creating animated gifs.

```
#! /bin/ksh

unalias rm
integer i=$1
while test i -lt $2
do

 if test i -lt 10
  then
  print "set terminal pbm small color; set output\"$3t=0000$i.ppm\"; set noxtics; set noytics;
  set size 1.0, 1.0; set yrange [0:.1];
  plot 'run.0000$i' using 1:2 w lines, 'run.0000$i' using 1:3 w lines;
  " >data0000$i.gnu
  gnuplot data0000$i.gnu
  ppmquant -map samp_colormap $3t=0000$i.ppm>$3at=0000$i.ppm
  ppmtogif -map samp_colormap $3at=0000$i.ppm > $30000$i.gif
  rm $3t=0000$i.ppm $3at=0000$i.ppm data0000$i.gnu
  i=i+99
  fi

if test i -gt 9 -a i -lt 1000
then
print "set terminal pbm small color; set output\"$3t=00$i.ppm\"; set noxtics; set noytics;
set size 1.0, 1.0; set yrange [0:.1];
plot 'run.00$i' using 1:2 w lines, 'run.00$i' using 1:3 w lines;
" >data00$i.gnu
gnuplot data00$i.gnu
ppmquant -map samp_colormap $3t=00$i.ppm>$3at=00$i.ppm
ppmtogif -map samp_colormap $3at=00$i.ppm > $300$i.gif
rm $3t=00$i.ppm $3at=00$i.ppm data00$i.gnu
i=i+100
fi

if test i -gt 999 -a i -lt 10000
then
print "set terminal pbm small color; set output\"$3t=0$i.ppm\"; set noxtics; set noytics;
set size 1.0, 1.0; set yrange [0:.1];
plot 'run.0$i' using 1:2 w lines, 'run.0$i' using 1:3 w lines;
" >data0$i.gnu
gnuplot data0$i.gnu
ppmquant -map samp_colormap $3t=0$i.ppm>$3at=0$i.ppm
ppmtogif -map samp_colormap $3at=0$i.ppm > $30$i.gif
rm $3t=0$i.ppm $3at=0$i.ppm data0$i.gnu
i=i+100
fi

done
```

Here the `--10` separates the frames by 0.1 s in real time, and the * is a wildcard that will include all .gif files in the present directory. Because we constructed the .gif files with sequential numbering, **gifmerge** pastes them together in the proper sequence and places them in the file

movie, which can be viewed with a browser.

3.8 TEXTURING AND 3-D IMAGING

In §13.10 we give a brief explanation of how the inclusion of *textures* (Perlin noise) in a visualization can add an enhanced degree of realism. While it is a useful graphical technique, it incorporates the type of correlations and coherence also present in fractals and thus is in Chapter 13, "Fractals & Statistical Growth." In a related vein, in §13.10.1 we discuss the graphical technique of ray tracing and how it, especially when combined with Perlin noise, can produce strikingly realistic visualizations.

Stereographic imaging creates a virtual reality in which your brain and eye see objects as if they actually existed in our 3-D world. There are a number of techniques for doing this, such as virtual reality caves in which the viewer is immersed in an environment with images all around, and projection systems that project multiple images, slightly displaced, such that the binocular vision system in your brain (possibly aided by appropriate glasses) creates a 3-D image in your mind's eye.

Stereographics is often an effective way to let the viewer see structures that might otherwise be lost in the visualization of complex geometries, such as in molecular studies or star creation. But as effective as it may be, stereo viewing is not widely used in visualization because of the difficulty and expense of creating and viewing images. Here we indicate how the low-end, inexpensive viewing technique known as *ChromaDepth* [Chrom] can produce many of the same effects as high-end stereo vision without the use of special equipment. Not only is the technique easy to view and easy to publish, it is also easy to create [Bai 05]. Indeed, the OpenDX color images work well with ChromaDepth.

ChromaDepth consists of two pieces: a simple pair of glasses and a display methodology. The glasses contain very thin, diffractive gratings. One grating is blazed so that it shifts colors on the red end of the spectrum more than on the blue end, and this makes the red elements in the 3-D scene appear to be closer to the viewer. This often works fine with the same color scheme used for coding topographic maps or for scientific visualizations and so requires little or no extra work. You just write your computer program so that it color-codes the output in a linear rainbow spectrum based on depth. If you do not wear the glasses, you still see the visualization, yet with the glasses on, the image appears to jump out at you.

3.9 GRACE/ACE: SUPERB 2-D GRAPHS FOR UNIX/LINUX

Our favorite package for producing publication-quality 2-D graphs from numerical data is *Grace/xmgrace*. It is free, easy to work with, incredibly powerful, and has been used for many of the (nicer) figures in our research papers and books. Grace is a WYSIWYG tool that also contains powerful scripting languages and data analysis tools, although we will not get into that. We will illustrate multiple-line plots, color plots, placing error bars on data, multiple-axis options and such. Grace is derived from *Xmgr*, aka *ACE/gr*, originally developed by the Oregon Graduate Institute and now developed and maintained by the Weizmann Institute [Grace]. Grace is designed to run under the Unix/Linux operating systems, although we have had success using in on an MS Windows system from within the Cygwin [CYG] Linux emulation.[8]

[8] If you do this, make sure to have the Cygwin download include the `xorg-x11-base` package in the `X11` category (or a later version), as well as `xmgrace`.

Table 3.1 *The text file `Grace.dat` contains one x value and one y value per line. The file `Grace2.dat` contains one x value and two y values per line.

Text Files `Grace.dat` and `Grace2.dat`*

`Grace.dat`		`Grace2.dat`		
x	y	x	y	z
1	2	1	2	50
2	4	2	4	29
3	5	3	5	23
4	7	4	7	20
5	10	5	10	11
6	11	6	11	10
7	20	7	20	7
8	23	8	23	5
9	29	9	29	4
10	50	10	50	2

3.9.1 Grace Basics

To learn about Grace properly, we recommend that you work though some of the tutorials available under its Help menu, study the user's guide, and use the Web tutorial [Grace]. We present enough here to get you started and to provide a quick reference. The first step in creating a graph is to have the data you wish to plot in a text (ASCII) file. The data should be broken up into columns, with the first column the abscissa (x values) and the second column the ordinate (y values). The columns may be separated by spaces or tabs but *not* by commas. For example, the file **`Grace.dat`** in the left column of Table 3.1 contains one abscissa and one ordinate per line, while the file **`Grace2.dat`** in the right column of Table 3.1 contains one abscissa and two ordinates (y values) per line.

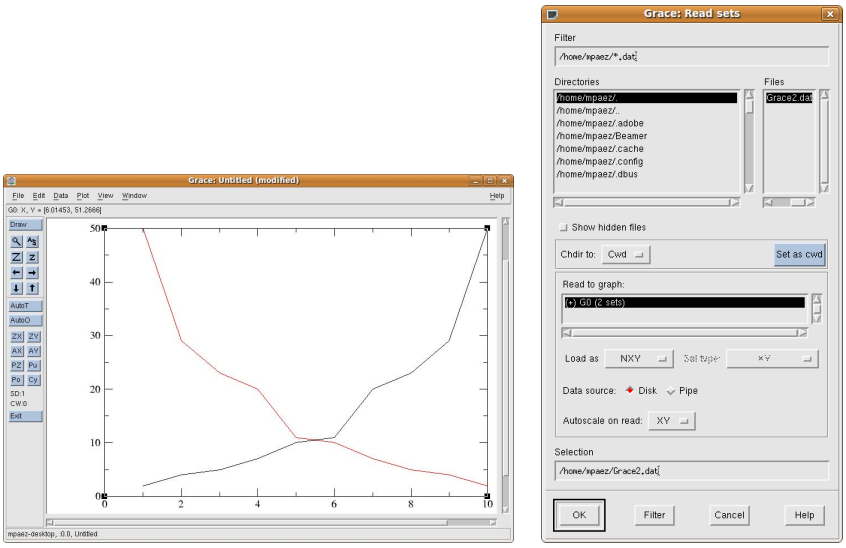
1. **Open Grace** by issuing the **`grace`** or **`xmgrace`** command from a Unix/ Linux/Cygwin command line (prompt):

   > **`grace`** Start Grace from Unix shell

   In any case, make sure that your command brings up the user-friendly graphical interface shown on the left in Figure 3.12 and not the pure-text command one.
2. **Plot a single data set** by starting from the menu bar on top. Then
   a. Select progressively from the submenus that appear, Data/Import/ASCII.
   b. The **`Read sets`** window shown on the right in Figure 3.12 appears.
   c. Select the directory (folder) and file in which you have the data; in the present case select **`Grace.dat`**.
   d. Select Load as/Single set, Set type/XY and Autoscale on read/XY.
   e. Select OK (to create the plot) and then Cancel to close the window.
3. **Plot multiple data sets** (Figure 3.12) by following similar procedure as in step 2, only now
   a. Select **`Grace2.dat`**, which contains two y values as shown in Table 3.1.
   b. Change Load as to NXY to indicate multiple data sets and then plot.

   Note: We suggest that you start your graphs off with Autoscale on read in order to see all the data sets plotted. You may then change the scale if you want, or eliminate some points and replot.

Figure 3.12 *Left:* The main Grace window with the file `Grace2.dat` plotted with the title, subtitle, and labels. *Right:* The data input window.



4. **Label and modify the axis properties** by going back to the main window. Most of the basic utilities are under the Plot menu.
   a. Select Plot/Axis properties.
   b. Within the Axis Property window that appears, select Edit/X axis or Edit/Y axis as appropriate.
   c. Select Scale/Linear for a linear plot, or Scale/Logarithmic for a logarithmic or semilog plot.
   d. Enter your choice for Axis label in the window.
   e. Customize the ticking and the numbers' format to your heart's desire.
   f. Choose Apply to see your changes and then Accept to close the window.
5. **Title the graph,** as shown on the left in Figure 3.12, by starting from the main window and again going to the Plot menu.
   a. Select Plot/Graph appearance.
   b. From the Graph Appearance window that appears, select the Main tab and from there enter the title and subtitle.
6. **Label the data sets,** as shown by the box on the left in Figure 3.12, by starting from the main window and again going to the Plot menu.
   a. Select Plot/Set Appearance.
   b. From the Set Appearance window that appears, highlight each set from the Select set window.
   c. Enter the desired text in the Legend box.
   d. Choose Apply for each set and then Accept.
   e. You can adjust the location, and other properties, of the legend box from Plot/Graph Appearance/Leg. box.
7. **Plotting points as symbols** (Figure 3.13 left) is accomplished by starting at the main menu. Then
   a. Select Plot/Set Appearance.
   b. Select one of the data sets being plotted.
   c. Under the Main/Symbol properties, select the symbol Type and Color.
   d. Choose Apply, and if you are done with all the data sets, Accept.
8. **Including error bars** (Figure 3.13 left) is accomplished by placing them in the data file read into Grace along with the data points. Un-

Figure 3.13 *Left:* A plot of points as symbols with no line. *Right:* A plot of the same points as symbols with lines connecting the points and with error bars read from the file.

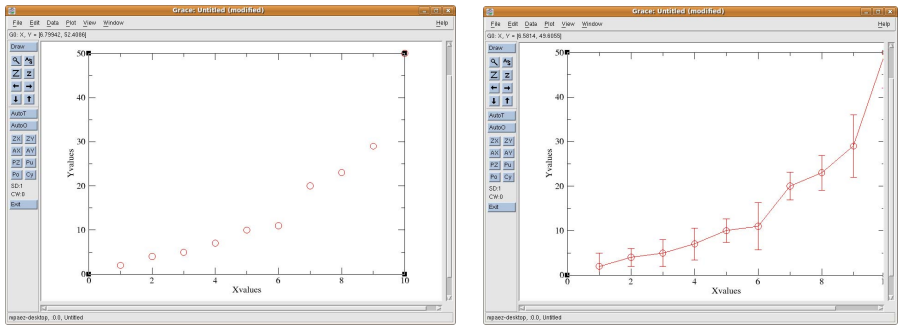
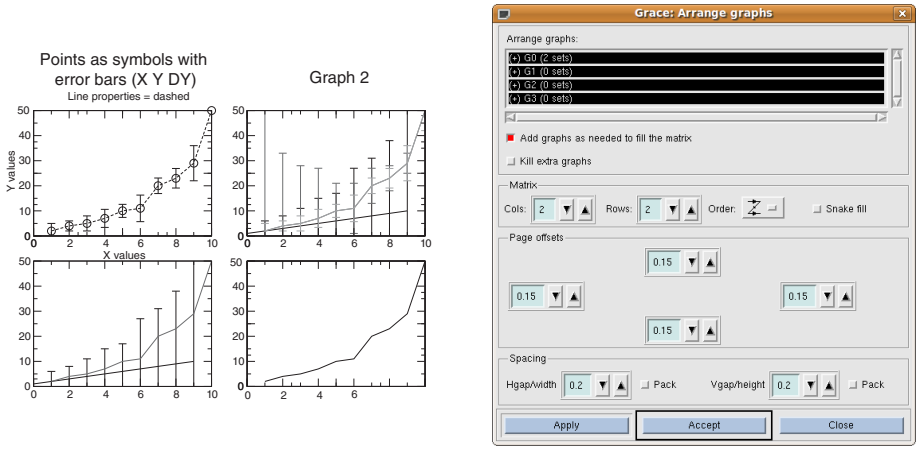


Figure 3.14 *Left:* Grace's placement of four graphs in a 2×2 matrix. *Right:* The Grace window that opens under Edit/Arrange Graphs and is used to set up the matrix into which multiple graphs are placed.



der Data/Read Sets are, among others, these possible formats for Set type:

(X Y DX), (X Y DY), (X Y DX DX), (X Y DY DY), (X Y DX DX DY DY)

Here DX is the error in the x value, DY is the error in the y value, and repeated values for DX or DY are used when the upper and lower error bars differ; for instance, if there is only one DY, then the data point is Y $\pm$ DY, but if there are two error bars given, then the data point is Y $+$ DY$_1$, $-$DY$_2$. As a case in point, here is the data file for (X Y DY):

X	1	2	3	4	5	6	7	8	9	10
Y	2	4	5	7	10	11	20	23	29	50
DY	3	2	3	3.6	2.6	5.3	3.1	3.9	7	8

9. **Multiple plots on one page** (Figure 3.14 left) are created by starting at the main window. Then
   a. Select Edit/Arrange Graphs.
   b. An Arrange Graphs window (Figure 3.14 right) opens and provides the options for setting up a matrix into which your graphs are placed.
   c. Once the matrix is set up (Figure 3.14 left), select each space in sequence and then create the graph in the usual way.
   d. To prevent the graphs from getting too close to each other, go back to the Arrange Graphs window and adjust the spacing between graphs.
10. **Printing and saving plots**
   a. To save your plot as a complete Grace project that can be opened again and edited, from the main menu select File/Save As and enter a **`filename.agr`** as the file name. It

is a good idea to do this as a backup before printing your plot (communication with a piece of external hardware is subject to a number of difficulties, some of which may cause a program to "freeze up" and for you to lose your work).

b. To print the plot, select File/Print Setup from the main window
c. If you want to save your plot to a file, select Print to file and then enter the file name. If you want to print directly to a printer, make sure that Print to file is not selected and that the selected printer is the one to which you want your output to go (some people may not take kindly to your stuff appearing on their desk, and you may not want some of your stuff to appear on someone else's desk).
d. From Device, select the file format that will be sent to the printer or saved.
e. Apply your settings when done and then close the Print/Device Setup window by selecting Accept.
f. If now, from the main window, you select File/Print, the plot will be sent to the printer *or* to the file. Yes, this means that you must "Print" the plot in order to send it to a file.

If you have worked through the steps above, you should have a good idea of how Grace works. Basically, you just need to find the command you desire under a menu item. To help you in your search, in Table 3.9.1 we list the Grace menu and submenu items.

Grace Menu and Submenu Items

Edit	*Data*
Data sets	Data set operations
Set operations	sort, reverse, join, split, drop points
copy, move, swap	Transformations
Arrange graphs matrix, offset, spacing	expressions, histograms, transforms, convolutions, statistical ops, interpolations
Overlay graphs	Feature extraction
Autoscale graphs	min/max, average, deviations, frequency,
Regions	COM, rise/fall time, zeros
Hot links	Import
Set/Clear local/fixed point	Export
Preferences	
Plot	*View*
Plot appearance	Show locator bar (default)
background, time stamp, font, color	Show status bar (default)
Graph appearance	Show tool bar (default)
style, title, labels, frame, legends	Page setup
Set appearance	Redraw
style, symbol properties, error bars	Update all
Axis properties	
labels, ticks, placement	
Load/Save parameters	
Window	
Command	Font tool
Point explorer	Console
Drawing objects	

Chapter Four

Python Object-Oriented Programs: Impedance & Batons

This chapter contains two units dealing with object-oriented programming *(OOP) at increasing levels of sophistication. In most of the codes in this book we try to keep our programming transparent to a wide class of users and to keep our Python examples similar to those in C and Fortran. Accordingly, we have deliberately shied away from the use of advanced OOP techniques. Nevertheless, OOP is a key element in modern programming and so it is essential that all readers have some understanding of it. We recommend that you read Unit I (the example is easy) so that you are comfortable declaring, creating, and manipulating both static and dynamic objects. Unit II deals with more advanced aspects of OOP and, while recommended, may be put off for later reading, especially for those who are object-challenged at this stage in their computing careers.*

VIDEO LECTURES, APPLETS AND ANIMATIONS FOR THIS CHAPTER

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*
Object Oriented Programming I	pdf	4.1–4.5

Applets

Name	*Sections*
Translate + Rotate	4.10.1

4.1 UNIT I. BASIC OBJECTS; COMPLEX IMPEDANCE

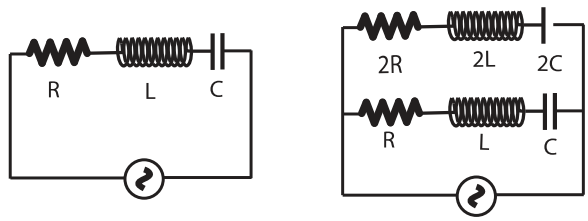
Problem: We are given a circuit containing a resistor of resistance R, an inductor of inductance L, and a capacitor of capacitance C (Figure 4.1 left). All three elements are connected in series to an alternating voltage source $V(t) = V_0 \cos \omega t$. Determine the magnitude and time dependence of the current in this circuit as a function of the frequency ω.

We solve this RLC circuit for you and assign as *your particular problem* that you repeat the calculation for a circuit in which there are two RLC circuits in parallel (Figure 4.1 right). Assume a single value for inductance and capacitance, and three values for resistance:

$$L = 1000\,\mathrm{H}, \qquad C = \frac{1}{1000}\,\mathrm{F}, \qquad R = \frac{1000}{1.5}, \ \frac{1000}{2.1}, \ \frac{1000}{5.2}\ \Omega. \tag{4.1}$$

Consider frequencies of applied voltage in the range $0 < \omega < 2/\sqrt{LC} = 2/\mathrm{s}$.

Figure 4.1 *Left:* An *RLC* circuit connected to an alternating voltage source. *Right:* Two *RLC* circuits connected in parallel to an alternating voltage. Observe that one of the parallel circuits has double the values of *R*, *L*, and *C* as does the other.



4.2 COMPLEX NUMBERS (MATH)

Complex numbers are useful because they let us double our work output with only the slightest increase in effort. This is accomplished by manipulating them as if they were real numbers and then separating the real and imaginary parts at the end of the calculation. We define the symbol z to represent a number with both real and imaginary parts (Figure 4.2 left):

$$z = x + iy, \quad \text{Re } z = x, \quad \text{Im } z = y. \tag{4.2}$$

Here $i \stackrel{\text{def}}{=} \sqrt{-1}$ is the imaginary number, and the combination of real plus imaginary numbers is called a *complex* number. In analogy to a *vector* in an imaginary 2-D space, we also use *polar coordinates* to represent the same complex number:

$$r = \sqrt{x^2 + y^2}, \qquad \theta = \tan^{-1}(y/x), \tag{4.3}$$

$$x = r\cos\theta, \qquad y = r\sin\theta. \tag{4.4}$$

The essence of the computing aspect of our problem is the programming of the rules of arithmetic for complex numbers. This is an interesting chore because while most computer languages contain all the rules for real numbers, you must educate them as to the rules for complex numbers (Fortran and Python being the well-educated exceptions). So although complex numbers are a *primitive data type* in Python, for pedagogic purposes, and to match the Java version of this text, we here construct complex numbers as *objects*. We start with two complex numbers, which we distinguish with subscripts:

$$z_1 = x_1 + i\,y_1, \quad z_2 = x_2 + i\,y_2. \tag{4.5}$$

Complex arithmetic rules derive from applying algebra to z's Re and Im parts:

Addition:
$$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2), \tag{4.6}$$

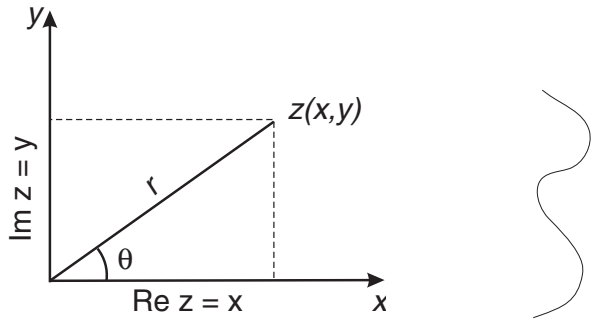
Subtraction:
$$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2), \tag{4.7}$$

Multiplication:
$$\begin{aligned} z_1 \times z_2 &= (x_1 + iy_1) \times (x_2 + iy_2) \\ &= (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1) \end{aligned} \tag{4.8}$$

Division:
$$\begin{aligned} \frac{z_1}{z_2} &= \frac{x_1 + iy_1}{x_2 + iy_2} \times \frac{x_2 - iy_2}{x_2 - iy_2} \\ &= \frac{(x_1x_2 + y_1y_2) + i(y_1x_2 - x_1y_2)}{x_2^2 + y_2^2}. \end{aligned} \tag{4.9}$$

An amazing theorem by Euler relates the base of the natural logarithm system, complex

Figure 4.2 *Left:* Representation of a complex number as a vector in space. *Right:* An abstract drawing; some people see a face, others a body silhouette.



numbers, and trigonometry:

$$e^{i\theta} = \cos\theta + i\sin\theta \quad \text{(Euler's theorem).} \tag{4.10}$$

This leads to the polar representation of complex numbers (Figure 4.2 left),

$$z \equiv x + iy = re^{i\theta} = r\cos\theta + ir\sin\theta. \tag{4.11}$$

Likewise, Euler's theorem can be applied with a complex argument to obtain

$$e^z = e^{x+iy} = e^x e^{iy} = e^x(\cos y + i\sin y). \tag{4.12}$$

4.3 RESISTANCE BECOMES IMPEDANCE (THEORY)

We apply Kirchhoff's laws to the RLC circuit in Figure 4.1 left by summing voltage drops as we work our way around the circuit. This gives the differential equation for the current $I(t)$ in the circuit:

$$\frac{dV(t)}{dt} = R\,\frac{dI}{dt} + L\,\frac{d^2I}{dt^2} + \frac{I}{C}, \tag{4.13}$$

where we have taken an extra time derivative to eliminate an integral over the current. The analytic solution follows by assuming that the voltage has the form $V(t) = V_0\,\cos\,\omega t$ and by guessing that the resulting current $I(t) = I_0 e^{-i\omega t}$ will also be complex, with its real part the physical current. Because (4.13) is linear in I, the law of linear superposition holds, and so we can solve for the complex I and then extract its real and imaginary parts:

$$I(t) = \frac{1}{Z}V_0 e^{-i\omega t}, \quad Z = R + i\left(\frac{1}{\omega C} - \omega L\right), \tag{4.14}$$

$$\Rightarrow \quad I(t) = \frac{V_0}{|Z|}e^{-i(\omega t+\theta)} = \frac{V_0}{|Z|}\left[\cos(\omega t+\theta) - i\sin(\omega t+\theta)\right], \tag{4.15}$$

$$|Z| = \sqrt{R^2 + \left(\frac{1}{\omega C} - \omega L\right)^2}, \quad \theta = \tan^{-1}\left(\frac{1/\omega C - \omega L}{R}\right).$$

We see that the amplitude of the current equals the amplitude of the voltage divided by the magnitude of the complex impedance, and that the phase of the current relative to that of the voltage is given by θ.

The solution for the two *RLC* circuits in parallel (Figure 4.1 right) is analogous to that with ordinary resistors. Two impedances in series have the same current passing through them, and so we add voltages. Two impedances in parallel have the same voltage across them, and so

we add currents:

$$Z_{\text{ser}} = Z_1 + Z_2, \qquad \frac{1}{Z_{\text{par}}} = \frac{1}{Z_1} + \frac{1}{Z_2}. \tag{4.16}$$

4.4 ABSTRACT DATA STRUCTURES, OBJECTS (CS)

What do you see when you look at the *abstract* object on the right of Figure 4.2? Some readers may see a face in profile, others may see some parts of human anatomy, and others may see a total absence of artistic ability. This figure is abstract in the sense that it does not try to present a true or realistic picture of the object but rather uses a symbol to suggest more than meets the eye. Abstract or formal concepts pervade mathematics and science because they make it easier to describe nature. For example, we may define $v(t)$ as the velocity of an object as a function of time. This is an abstract concept in the sense that we cannot see $v(t)$ but rather just infer it from the changes in the observable position. In computer science we create an abstract object by using a symbol to describe a collection of items. In Python we have built- in or *primitive data types* such as integers, floating-point numbers, Booleans, and strings. In addition, we may define *abstract data structures* of our own creation by combining primitive data types into more complicated structures called *objects*. These objects are abstract in the sense that they are named with a single symbol yet they contain multiple parts.

To distinguish between the general structure of objects we create and the set of data that its parts contain, the general object is called a *class*, while the object with specific values for its parts is called an *instance* of the class, or just an *object* . In this unit our objects will be complex numbers, while at other times they may be plots, vectors, or matrices. The classes that we form will not only contain objects (data structures) but also the associated methods for modifying the objects, with the entire class thought of as an object.

In computer science, abstract data structures must possess three properties:

1. **Typename:** Procedure to construct new data types from elementary pieces.
2. **Set values:** Mechanism for assigning values to the defined data type.
3. **Set operations:** Rules that permit operations on the new data type (you would not have gone to all the trouble of declaring a new data type unless you were interested in doing something with it).

In terms of these properties, when we declare a variable to have two (real and imaginary) parts, we satisfy property 1. When we assign floats to the parts, we satisfy property 2. And when we define addition and subtraction, we satisfy property 3. The actual process of creating objects in your Python program uses *nonstatic, dynamic* or *active* variables and methods that modify or interact with objects. It is these dynamic (nonstatic) method that utilize the power of object-oriented programming.

4.4.1 Object Declaration and Construction

Though we may assign a name like `x` to an object, because objects have multiple components you cannot assign one explicit *value* to an object. It follows then, that when Python deals with objects, it does so by *reference*; that is, the name of the variable *refers to the location in memory* where the object is stored and not to the explicit values of the object's parts. To

see what this means in practice, the class file **Complex.py** in Listing 4.1 adds and multiplies complex numbers, with the complex numbers represented as objects.

Listing 4.1 **ComplexDummy.py** defines the `Complex` class that permits the use of complex data objects. The constructor method `__init__` uses `z` as an intermediary or dummy variable.

```
# ComplexDummy.py: Performs complex algebra with dummy intermediary

class Complex:
    def __init__(z, x, y):                          # class constructor
        z.re = x                            # assign real part of complex
        z.im = y                            # assign imag part of complex

    def addt(z1, z2):                                       # adds z1 + z2
       return Complex(z1.re + z2.re, z1.im + z2.im)

    def subt(z1, z2):                                  # subtracts z1-z2
        return Complex(z1.re - z2.re, z1.im- z2.im)

    def mult(z1, z2):                                 # multiplies z1*z2
        return Complex(z1.re*z2.re -z1.im*z2.im,
                       z1.re*z2.im+z1.im*z2.re)

    def __repr__ (z):                  # convert z to string for printing
        return '(%f, %f) ' %(z.re, z.im)

print('Operations with two complex numbers\n')

z1 = Complex(2.0, 3.0)                          # the first complex number
print ( 'z1 =', z1)
z2 = Complex(4.0, 6.0)                                # other complex one
print("z2 =" , z2)
z3 = Complex.addt(z1,z2)                                   # add operation
print("z1 + z2= ",z3)
print('z1 - z2=', Complex.subt(z1,z2))             # prints both re and im
print('z1 * z2=', Complex.mult(z1,z2))                   # multiplication
print("Enter and return any character to quit")
s = raw_input()
```

4.4.2 Implementation in Python

In the program **ComplexDummy.py** we define and test the class **Complex** that adds, subtracts and multiplies complex numbers (division is left as an exercise).

1. Enter the program **ComplexDummy.py** by hand, trying to understand it as best you are able. (Yes, we know that you can just copy it, but then you do not become familiar with the constructs.)
2. Compile and execute this program and check that the output agrees with the results you obtain via a hand calculation. The results we obtained were:

```
Operations with two complex numbers
z1 = (2.000000 , 3.000000)
z2 = (4.000000 , 6.000000)
z1 + z2=  (6.000000 , 9.000000)
z1 - z2= (-2.000000 , -3.000000)
z1 * z2= (-10.000000 , 24.000000)
```

Observe the following in **ComplexDummy.py**:

- The class **Complex** is declared on line 3 with the statement
  class Complex

- The class methods are defined with the **`def`** statement.
- The first method in the program has the funny-looking name **`__init__`**. The double underscores on each side of a method indicate a reserved name in Python, in this case, for the *class constructor*. Here **`__init__`** *is special because it constructs the* initial *object that will be given the same name as the class*. Yes, this means that even though this methods's name is **`__init__`**, it produces a **`Complex`** object because **`Complex`** is the name of the class.
- The word **`Complex`** is a number of things in this program. It is the name of the *class* on line 3, as well as the name of the method called to create complex objects on lines 6 and 10 (we remind the reader that calling **`Complex`** really calls **`__init__`**). Although neophytes may view this multiple and indirect uses of the name **`Complex`** as confusing, more experienced users may view it as elegant and efficient.
- The constructor has three arguments, **`z`**, **`x`** and **`y`**. *By convention, the first argument of a method always refers to the object on which the method acts.* Since in this case the first argument is **`z`**, it represents the **`Complex`** object and so can be used as a **`Complex`** object without being further assigned.
- The second and third arguments of the constructor **`__init__`** are **`x`** and **`y`**, and represent the real and the imaginary parts of the complex object to be constructed. Since only the first argument is special and assumed to be a complex number, **`x`** and **`y`** are just ordinary doubles.
- The actual object returned by the constructor is a *tuple* containing two doubles (real and imaginary parts). A tuple is like a list, but is enclosed in round brackets (...) rather than in the square brackets [...] of a list. Operationally, a tuple differs from a list in being immutable (elements can't be changed) and this makes them safer and more efficient to use.
- The *dot operator* is used to refer to the *attributes* of an object (as in Java). In this case we extract the *re and* im parts of a complex object much like we would extract the components of a physical vector, that is, by taking dot products of it with unit vectors along each axis:

`z1.re`	real part of object **`z1`**	**`z1.im`**	imaginary part of object **`z1`**
`z2.re`	real part of object **`z2`**	**`z2.im`**	imaginary part of object **`z2`**

  The dot notation is also used to apply methods to objects, as we shall see.
- The constructor **`__init__`** is seen to use an intermediary or dummy variable **`z`** to construct the complex object. It sets **`self`** equal to **`z`** and returns **`self`** as the complex object. A more elegant method will follow soon.
- The variable **`self`** is always used to name the object associated with the class (also true in Java), in this case **`Complex`**.
- Our version of complex arithmetic uses the methods **`addt, subt`** and **`mult`**, each of which returns a Complex object consisting of a tuple of real and imaginary parts. (Names like "add" and "sum" might seem more natural, yet they are have already been reserved.)
- Notice that **`ComplexDummy.py`** also contains the method **`__str__`**, with the underscores again indicating a method native to Python. We use it here to redefine for this class the built-in Python method for converting an argument into a string. Specifically, when called by the **`print`** command it permits both parts of a complex number to be printed in one fell swoop.
- As is standard in Python, indentation is used to define program structure.
- There is no explicit **`main`** method in this program where execution begins, but instead it just begins on line 22 with the first statement that is not contained in a method.

Let us now stock of what we have up to this point. On line 4 we declare that the variables **`z.re`** and **`z.im`** will be the two separate parts of the created object. As each instance of each object created will have different values for the object's parts, these variables are referred to as *instance variables*. Because the name of the class and the names of the objects it creates are all the same, it is sometimes useful to use yet another word to distinguish one from the other.

Hence the phrase *instance of a class* is used to refer to the created objects (in our example, **z1**, **z2**, and **z3**). This distinguishes them from the abstract data type (**Complex**). Look now at the part of the program that calls the methods and the statement:

```
z1 = Complex(2.0, 3.0)                                                    24
```

Although the class file does *not* contain a method **Complex**, because this is the name of the class, Python substitutes the default constructor **__init__** for **Complex** and creates the object **z1** with two parts to it. Likewise for object **z2**.

Exercise: Change line 24 so that **z1** has real and imaginary parts of 0. Then, set the real and imaginary parts to 2.0 and 3.0 explicitly with statements of the form **z1.re = 2.0**, and check that you get the same result as before. ▌

Once our complex-number objects have been created, it is easy to do arithmetic with them. We know the rules of complex arithmetic and complex trigonometry and so can write Python methods to implement them. It makes sense to place these methods in the same class file that defines the data type since these associated methods are needed to manipulate objects of that data type. In contrast to **z1** and **z2**, the complex object **z3** is not created with the constructor directly, but instead is created by being set equal to the sum of two objects:

```
z3 = Complex.addt(z1, z2)                                                 30
```

This says to add the complex number **z1** to the complex number **z2**, and then store the result "as" (in the memory location reserved for) the complex number **z3**.

Lines 9–10 contain the method **addt**. As indicated before, the *dot operator* convention means that **z1.re** will contain the **re** part of **z1** and that **z1.im** will contain the imaginary part. Thus the operation on line 10 extracts the real parts of **z1** and **z2**, adds them together, and then does the same for the imaginary parts. We also notice on line 10 that the method **addt** returns an object of the type **Complex** with two parts, and so **z3** becomes a Complex object. Also notice on that line how the **return** statement does not directly return a **Complex** object, such as **z1**, but rather seems to return a call to the method **Complex**; the method call returns a **Complex** object that then gets returned via the method **addt**.

Notice how the methods defined in this class have the class name preceding them and separated from the class name by a dot (for example, **Complex.addt**). One way of understanding line 30 above is that we are performing the operation **addt** on the object **Complex**, with **z1** and **z2** being parameter objects (arguments for the methods). The general form is

```
<object>.<method>(<arg1>, <arg2>, ...)
```

where it is not necessary for a method to take arguments. Another way of viewing this statement is that **Complex** is the name of the class in which the method is to be found. Go ahead, try running the program without the **Complex.** in front of a method name and see if Python will accept it!

4.4.3 Without Dummies and z's

The program **ComplexSelf.py** in Listing 4.2 also adds and multiplies complex numbers as objects. However, it avoids the use of the dummy variable **z** in the constructor and **z1** and **z2** in the other methods by, instead, using the special names **self** and **other** as variables. The reserved word **self** refers to the object associated with the class, which in this case is **Complex**,

while the reserved word **other** refers to *another* object *of the same class*, but different from **self**. This is similar to Java. Other than eliminating one line in the **addt** method, the program is the same as before but with standard names in all the methods.

Listing 4.2 **ComplexSelf.py** defines the class `Complex` using the special names `self` and `other`.

```
# ComplexSelf.py: creates Complex class using self & other, no dummy, z's
from sys import version
if int(version[0])>2:   # raw_input deprecated in Python 3
    raw_input=input
class Complex:

    def __init__(self, x, y):                          # class constructor
        self.re = x                                     # assign real part
        self.im = y                                     # assign imag part

    def addt(self, other):                            # adds self to other
       return Complex(self.re + other.re, self.im + other.im)

    def subt(self, other):                           # subtract self-other
        return Complex(self.re - other.re, self.im- other.im)

    def mult(self, other):                       # multiplies self * other
        return Complex(self.re*other.re -self.im*other.im,
                       self.re*other.im+self.im*other.re)
    def __repr__ (self):                   # convert z to string for print
        return '(%f, %f) ' %(self.re, self.im)

print('Operations with two complex numbers\n')

z1 = Complex(2.0, 3.0)                               # first complex number
print('z1 =', z1)
z2 = Complex(4.0, 6.0)                                  # other complex one
print("z2 =",z2)
z3 = Complex.addt(z1,z2)                                       # add z1 + z2
print("z1 + z2= ",z3)
print('z1 - z2=', Complex.subt(z1,z2))
print('z1 * z2=', Complex.mult(z1,z2))
print("Enter and return any character to quit")
s = raw_input()
```

Exercise

1. Enter the **ComplexSelf.py** class file by hand, trying to understand it in the process. If you have entered **ComplexDummy.py** by hand, you may modify that program to save some time (but be careful!).
2. Compile and execute **ComplexSelf.py** and check that the output agrees with the results you obtained previously. ▌

4.4.4 The Elegance of Overload

A more elegant way of writing our class for complex arithmetic is to use a technique known as *operator overloading* in which we extend Python's built-in definitions of addition, subtraction and multiplication so that they also apply to complex numbers. In this way we can use our usual symbols and let Python decide which method to use based on the arguments given to the methods. Some of the special methods defined in Python for the overloading are:

Symbol	Method	Symbol	Method
+	**__add__**	-	**__sub__**
*	**__mul__**	/	**__div__**

with more methods defined for overloading other symbols.

Listing 4.3 **ComplexOverLoad.py** defines a `Complex` class that overloads the +, - and / operators.

```
# ComplexOverload.py: performs complex arithmetic via operator overload

class Complex1:

    "Another class to add subtract and multiply complex numbers"
    "Uses overload of operators + - and * "

    def __init__(self,x,y):                          # class constructor
        self.re=x                                     # assing real part
        self.im=y                                     # assing imag part
        # print " en init ", self.re, "  ", self.im

    def __add__(self,other):                      # extend predefined add
       return Complex1(self.re + other.re, self.im + other.im)

    def __sub__(self,other):                  # extend predefined subtract
        return Complex1(self.re - other.re, self.im - other.im)

    def __mul__(self, other):                     # extend predefined mult
        return Complex1(self.re*other.re -self.im*other.im,
                        self.re*other.im + self.im*other.re)

    def __repr__ (self):
        return '(%f , %f) ' %(self.re, self.im)

print('\n Operations with two complex numbers via operator overload\n')
z1 = Complex1(2.0,3.0)                             # first complex number
print('z1=', z1)
z2= Complex1(4.0,6.0)                             # other complex number
print("z2 = ",z2, "\n")
print("z1 + z2=", z1+z2)                              # use __add__ for +
print("z1 * z2=", z1*z2)                              # use __mul__ for *
print('z1 - z2=', z1-z2)                              # use __sub__ for -
print('z1 * z2=', z1*z2)
print("Enter and return any character to quit")
s = raw_input()
```

The program **`ComplexOverLoad.py`** in Listing 4.3 does the same calculation as the previous programs but with overloading. Its executable part is seen to use the **`Complex`** constructor to create the complex numbers, but other than that, the arithmetic operations look like regular arithmetic operations. However, the definitions of **`__add__`**, **`__sub__`** and **`__mul__`** for complex numbers in the top part of the program are new, and will be used only when acting on **`Complex`** objects. Otherwise, the regular definitions are used.

4.4.5 Python OOP Differs from Java and C++*

(If you are not a Java or C++ programmer, it makes sense to skip this section.)

1. Python classes are by default *public* and so do not have to be declared as such.
2. There are no static methods for objects in Python, that is, they are all dynamic, active or nonstatic.
3. For advanced programming that integrates Python with C, there are procedures to deal with C's static methods in Python.
4. Python does not require the user to declare a variable's *type*, for example, as we might in Java with **`public static double x;`**.
5. Python does not require the name of a file to match the name of the class it contains, and so we have the class **`complex`** in the file named **`ComplexDummy.py`**.

4.5 PYTHON'S BUILT-IN COMPLEX NUMBER TYPE

Finally, we should indicate that in addition to built-in numeric data types of integers, long integers and floating point numbers (doubles), Python also has built-in complex objects, which are composed of doubles. Similar to what we have created here, Python's complex numbers are declared as `complex(real, imag)`, with `z.real` and `z.imag` the real and imaginary parts respectively, and with the parentheses mandatory. Note that the built-in type name uses lowercase `complex` and `z.real` and `z.imag` instead of our `z.re` and `z.im`. The first argument in the `complex` declaration can also be a complex number represented via a string, in which case the second argument should not be given. To see how this is done, from within an interactive Python shell we have:

```
>>> z1 = complex(2.0, 3.0)          Define complex z1
>>> z2 = complex(4.0, 6.0)          Define complex z2
>>> print z1, z2
  (2+3j) (4+6j)                      The returned output
```

Note here that the suffix of `j` or `J` is used for the imaginary number (normally called i in math and physics, but j in engineering). For this to work, there cannot be any spaces before the `j`, and the pure imaginary number i needs to have a 1 before the `j`, that is `1j`. This also means that we can write complex numbers as `(real + imag j)` and even use that form with methods:

```
>>> z1 = 2.0 + 3.0j                 Define complex z1
>>> z2 = 4.0 + 6.0J                 Define complex z2
>>> print z1+z2
>>> print z1 + z2
 (6+9j)
>>> abs(1+1j)
 1.4142135623730951                 The answer
>>> z1.conjugate()                  Note object.method
 (2-3j)                             The answer
```

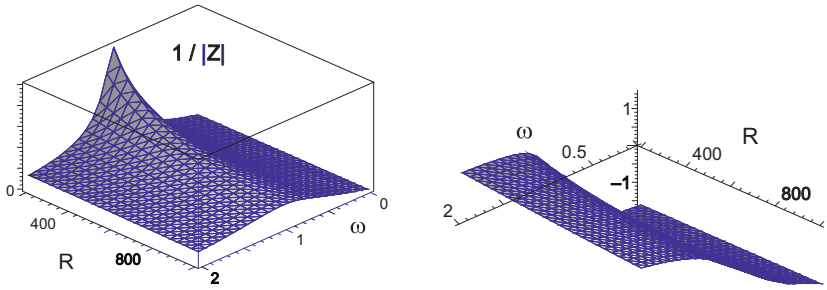
4.6 COMPLEX CURRENTS (SOLUTION)

1. Extend the class `Complex` by adding new methods to take the modulus, take the complex conjugate, and determine the phase of complex numbers.
2. Test your methods by checking that the following identities hold for a variety of complex numbers:

$$\begin{aligned} z + z &= 2z, & z + z* &= 2\,\mathrm{Re}\,z \\ z - z &= 0, & z - z* &= 2\,\mathrm{Im}\,z \\ zz* &= |z|^2, & zz* &= r^2 \quad \text{(which is real)} \end{aligned} \tag{4.17}$$

   Hint: Compare your output to some cases of pure real, pure imaginary, and simple complex numbers that you are able to evaluate by hand.
3. Equation (4.14) gives the magnitude and phase of the current in a single *RLC* circuit. Modify the given complex arithmetic program so that it performs the required complex arithmetic.
4. Compute and then make a plot of the magnitude and phase of the current in the circuit as a function of frequency $0 \le \omega \le 2$.
5. Construct a $z(x, y)$ surface plot of the magnitude and phase of the current as functions of *both* the frequency of the external voltage ω and of the resistance R. Observe how the magnitude has a maximum when the external frequency $\omega = 1/\sqrt{LC}$. This is the *resonance* frequency.

Figure 4.3 *Left:* A plot of the inverse impedance $1/|Z|$ *versus* resistance R and frequency ω. This behavior of the impedance leads to the magnitude of the current having a maximum at $\omega = 1$. *Right:* A plot of the current's phase *versus* resistance R and frequency ω showing that below resonance, $\omega < 1$, the current lags the voltage, while above resonance the current leads the voltage.



6. Another approach is to make a 3-D visualization of the complex Z as a function of a complex argument (Figure 4.3). Do this by treating the frequency $\omega = x + iy$ as a complex number. You should find a sharp peak at $x = \text{Re}(\omega) = 1$. Adjust the plot so that you tell where $\text{Im}(1/Z)$ changes sign. If you look closely at the graph, you should also see that there is a maximum for a negative imaginary value of ω. This is related to the length of the lifetime of the resonance.
7. **Assessment:** You should notice a resonance peak in the magnitude at the same frequency for which the phase vanishes. The smaller the resistance R, the more sharply the circuit should pass through resonance. These types of circuits were used in the early days of radio to tune to a specific frequency. The sharper the peak, the better the quality of reception.
8. The second part of the problem dealing with the two circuits in parallel is very similar to the first part. You need to change only the value of the impedance Z used. To do that, explicitly perform the complex arithmetic implied by (4.16), deduce a new value for the impedance, and then repeat the calculation of the current.

4.7 OOP WORKED EXAMPLES

Creating object-oriented programs requires a transition from a procedural programming mindset, in which functions take arguments as input and produce answers as output, to one in which objects are created, probed, transferred, and modified. To assist you in the transition, we present here two sample procedural programs and their OOP counterparts. In both cases the OOP examples are longer but presumably easier to modify and extend.

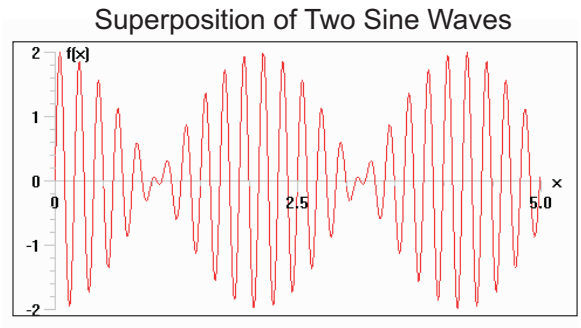
4.7.1 OOP Beats

You obtain beats if you add together two sine functions y_1 and y_2 with nearly identical frequencies,

$$y_3(t) = y_1(t) + y_2(t) = A\,\sin(30\,t) + A\,\sin(33\,t). \tag{4.18}$$

Beats look like a single sine wave with a slowly varying amplitude (Figure 4.4). In Listing 4.4 we give **`Beats.py`**, a simple program that plots beats. You see here that all the computation is done in the main program, with the only method called is that for plotting. The two sine functions are added together on line 15, which is within the for loop that runs over time. Contrast this with the object-oriented program **`OOPbeats.py`** in Listing 4.5 that produces the same graph.

Figure 4.4 Superposition of two sine waves of similar wave number that produce beats.



Listing 4.4 **Beats.py** plots beats using procedural programming. Contrast this with the object-oriented program `Beats.py` in Listing 4.5.

```
#Beats: plots sin(30*x)+sin(33*x), 0<=x <=5.0

from visual.graph import *                    # graphics and math classes
# Plot setup  (init x, width, height, title, axes names, font, color)
graph = gdisplay(x = 0,y = 0, width = 500, height = 300,    # set up plot
                title = 'Beats: f(x)=sin(30*x)+sin(33*x)',
                xtitle = 'x', ytitle='f(x)', xmax=5.0, xmin=0.0, ymax=2,
                ymin=-2, foreground=color.black ,background=color.white)
function = gcurve(color = color.red)
for x in arange(0., 5.0, 0.01):                          # min, max, step
    rate(40)
    y = sin(30*x) + sin(33*x)                          # function to plot
    function.plot( pos=(x, y))                            # plot function
```

In the OOP version, the main program is the two lines at the end. It is short because all it does is create an **`OOPbeats`** object named **`sumsines`**, and then sums the two waves by having **`sumwaves`** modify the object. The constructor for an **`OOPbeats`** object is given on line 6, followed by the **`sumwaves`** method. The **`sumwaves`** method takes the arguments needed for plotting and does the plotting.

Listing 4.5 **Beats.py** plots beats using OOP, in contrast to the procedural program `Beats.py` in Listing 4.4.

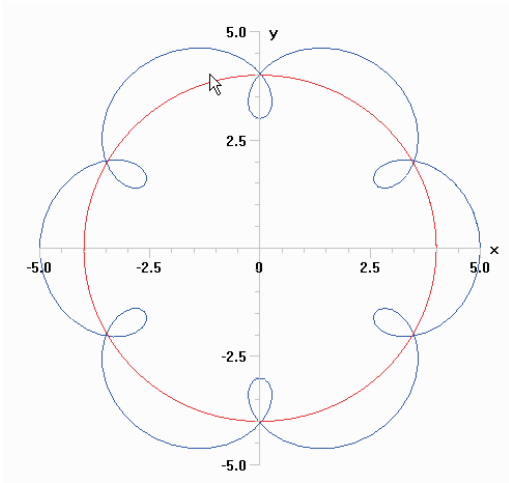
```
#Beats: plots sin(30*x)+sin(33*x), 0<=x <=5.0

from visual.graph import *                    # graphics and math classes
# Plot setup  (init x, width, height, title, axes names, font, color)
graph = gdisplay(x = 0,y = 0, width = 500, height = 300,    # set up plot
                title = 'Beats: f(x)=sin(30*x)+sin(33*x)',
                xtitle = 'x', ytitle='f(x)', xmax=5.0, xmin=0.0, ymax=2,
                ymin=-2, foreground=color.black ,background=color.white)
function = gcurve(color = color.red)
for x in arange(0., 5.0, 0.01):                          # min, max, step
    rate(40)
    y = sin(30*x) + sin(33*x)                          # function to plot
    function.plot( pos=(x, y))                            # plot function
```

4.7.2 OOP Planet

In our second example we add together two periodic functions representing positions *versus* time. One set describes the position of the moon as it revolves around a planet, and the other the position of the planet as it revolves about the sun. The position of the planet at time t relative to the sun is described by

$$x_p = R\,\cos(\omega_p\,t), \quad y_p = R\,\sin(\omega_p\,t). \tag{4.19}$$

Figure 4.5 The trajectory of a satellite in a circular orbit about a planet as seen from a planet in a circular orbit around the sun.



The position of the satellite, relative to the sun, is given by the sum of its position relative to the planet and the position of the planet relative to the sun:

$$x_s = x_p + r\,\cos(\omega_s\,t) = R\,\cos(\omega_p\,t) + r\,\cos(\omega_s\,t),$$
$$y_s = y_p + r\,\sin(\omega_s\,t) = R\,\sin(\omega_p\,t) + r\,\sin(\omega_s\,t).$$

If $\omega_s \simeq \omega_p$ this looks like beating, yet we will make a parametric plot of $x(t)$ *versus* $y(t)$ to visualize the orbit. The procedural program **`Moon.py`** in Listing 4.6 produces Figure 4.5.

Listing 4.6 The procedural program **Moon.py** computes trajectory of a satellite seen from a planet.

```
#    Moon.py: moon orbiting a planet via OOP

from visual.graph import *                    # graphics and math classes
graph = gdisplay(x = 0, y = 0, width = 500, height = 500,
                title = 'Motion of a satellite around a planet',
                xtitle='x', ytitle='y', xmax=5.,xmin=-5., ymax=5, ymin=-5,
                foreground = color.black, background = color.white)
moonfun = gcurve(color = color.red)          # for a 2D graph, curve in red
Radius = 4.0                                          # Planet Orbit radius
wplanet = 2.0                                       # planet angular velocity
radius = 1.0                                 # moon Orbit radius around planet
wmoon = 14.0                                   # moon ang. vel. around planet
for time in arange(0., 3.2, 0.02):                    # time min, max, step
    rate(20)
    x = Radius*cos(wplanet*time) + radius*cos(wmoon*time)        # moon x
    y = Radius*sin(wplanet*time) + radius*sin(wmoon*time)        # moon y
    moonfun.plot(pos = (x, y) )                       # plots moon position
```

Exercise: Rewrite the program using OOP:

1. Define a mother class **`OOPPlanet`** containing:

`Radius`	Planet's orbit radius
`wplanet`	Planet's orbit ω
(**`xp, yp`**)	Planet's coordinates
`getX(double t), getY(double t)`	Planet coordinates methods
`trajectory()`	Method for planet's orbit

2. Define a daughter class **`OOPMoon`** containing:

radius	Radius of moon's orbit
wmoon	Frequency of moon in orbit
(**xm, ym**)	Moon's coordinates
trajectory()	Method for moon's orbit relative to sun

3. The main program must contain one instance of the class **planet** and another instance of the class **Moon**, that is, one **planet** object and one **Moon** object.
4. Have each instance call its own **trajectory** method to plot the appropriate orbit. For the planet this should be a circle, while for the moon it should be a circle with retrogrades (Figure 4.5).

One solution, which produces the same results as the previous program, is the program **OOPPlanet.py** in Listing 4.7. As with **Beats.py**, the main program for **OOPPlanet.py** is at the end and is short; it creates an **OOPMoon** object and then plots the moon's orbit by applying the **trajectory** method to the object. The class contains:

1. A constructor **__init__** used to initialize the values of the planet radius and its angular velocity.
2. Methods **getX** and **getY** that obtain the position of the planet at a given time from a planet object. Methods such as these that extract information from objects are called *accessors*.
3. A method **scenario** that prepares a window in which a planet's or moon's orbit (or both) will be plotted.
4. A method called **position** that plots the planet's orbit around the sun.

Listing 4.7 **OOPPlanet.py** creates an `OOPMoon` object and then plots the moon's orbit by applying the `trajectory` method to the object.

```
# OOPPlanet.py: Planet orbiting Sun, or moon orbiting planet

from visual.graph import *                    # graphics and math classes

class OOPPlanet:
    def __init__(self, Rad, pomg):            # Planet class constructor
        self.Radius = Rad                                  # Planet Radius
        self.wplanet = pomg                    # Planet angular velocity

    def getX(self, time):
       return self.Radius*cos(self.wplanet*time)

    def getY(self, time):
       return self.Radius*sin(self.wplanet*time)
    #Defines window parmas, can be used by planet or moon

    def scenario(self, mytitle, myxtitle, myytitle, xma, xmi, yma, ymi):
        graph = gdisplay(x = 0, y = 0, width = 500, height = 500,
                title=mytitle, xtitle=myxtitle, ytitle=myytitle, xmax=xma,
                xmin = xmi, ymax=yma, ymin=ymi, foreground = color.black,
                background = color.white)

    def position(self):                                  # for plane orbit
        planetfun = gcurve(color = color.red)
        for time in arange(0.,  + 3.3, 0.02):             # min, max, step
            xp = self.getX(time)
            yp = self.getY(time)
            planetfun.plot(pos = (xp, yp) )                # plot function

class OOPMoon(OOPPlanet):              # OOPMoon is  subclass of OOPPlanet

    def __init__(self, Rad, pomg, rad, momg):   # constructor planet&Moon

        OOPPlanet.__init__(self, Rad, pomg)
        self.radius = rad                                  # Moon radius
        self.wmoon = momg                           # Moon angular velocity

    def position(self):                         # moon orbit around planet
        moonfun = gcurve(color = color.blue)          # trace orbit in blue
        for time in arange(0.,  + 3.3, 0.02):             # min, max, step
            xm = self.getX(time) + self.radius*cos(self.wmoon*time)
```

```
            ym = self.getY(time) + self.radius*sin(self.wmoon*time)
            moonfun.plot(pos = (xm, ym) )          # plots moon's position
            rate(10)
moon = OOPMoon(4.0, 2.0, 1.0, 14.0)                    # init planet & moon
moon.scenario('Satelite orbit around planet', 'x', 'y', 5,  -5, 5,  -5)
moon.position()                                  # overrrides planet position
planet = OOPPlanet(4.0, 2.0)                           # initializes planet
planet.position()
#uncomment next two lines for window with the motion of the planet
# planet.scenario('Planet orbit around Sun', 'x', 'y', 5,  - 5, 5,  - 5)
# planet.position()
```

4.8 SUBCLASSES AND CLASS INHERITANCE

What is new about the program in Listing 4.7 is that it contains two classes, **OOPPlanet** on line 6 and **OOPMoon** on line 30, with **OOPMoon** within **OOPPlanet**. That being the case, **OOPMoon** is a *subclass* or *daughter class* of the *mother class* **OOPPlanet**. The daughter class inherits the properties of the mother class as well as having properties of its own. Thus, on lines 40 and 41, **OOPMoon** uses the **getX(time)** and **getY(time)** methods from the **OOPPlanet** class without having to say **OOPPlanet.getX(time)** to specify the class name.

Specifically, note on line 30 how the class **OOPMoon** is defined a as a child of the class **OOPPlanet** with the statement containing **OOPPlanet** as an argument:

```
class OOPMoon(OOPPlanet):
```
Defines subclass OOPMoon

Because **OOPMoon** is a subclass of **OOPPlanet**, it inherits the **scenario, getX** and **getY** from **OOPPlanet**. But since the **OOPMoon** class defines its own **position** method, this overrides the one in the mother class.

To plot the motion of the moon around a planet as seen from the sun, it is necessary to specify the radius of the planet's orbit around the sun as well as its angular velocity. This is the reason why the parameter list of the constructor for **OOPMoon**,

```
def __init__( self, Rad, pomg, rad, momg ):
```
32

not only contains **self**, which passes the properties of **OOPMoon**, but also contains properties of the planet and the angular velocity of the moon around the planet. In turn, we see that on line 33,

```
OOPPlanet.__init__(self, Rad, pomg)
```
33

two of these arguments are transmitted to the **OOPPlanet** class through its member **init**. Because this initialization does not actually create any instances of the two classes, it is called an *abstract initialization*. As a consequence of inheritance, the methods **getX** and **getY** defined in the **OOPPlanet**, are also owned by the **OOPMoon** and are used there to add the coordinates of the moon and planet together:

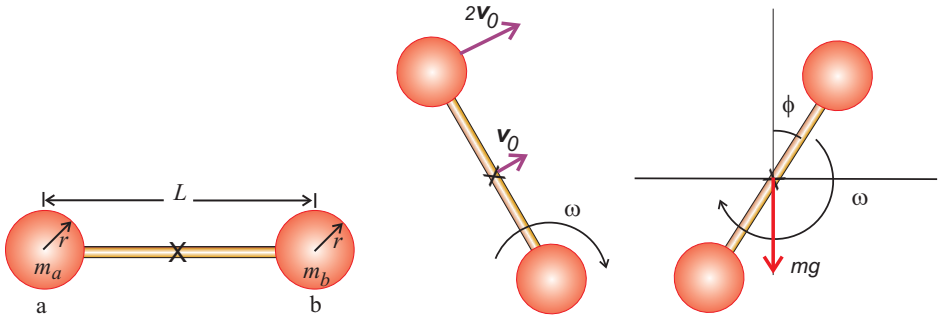
```
xm=self.getX(time) + self.radius*math.cos(self.wmoon*time) # x
```
40

```
ym=self.getY(time) + self.radius*math.sin(self.wmoon*time) # y
```
41

4.9 UNIT II. ADVANCED OBJECTS; BATON PROJECTILES ⊙

In this unit we look at more advanced aspects of OOP. These aspects are designed to help make programming more efficient by making the reuse of already written components easier and more reliable. The ideal is to permit this even for entirely different future projects for which you will have no memory or knowledge of the internal workings of the already written components that you want to reuse. OOP concepts can be particularly helpful in complicated projects in which you need to add new features without "breaking" the old ones and in which you may be modifying code that you did not write.

Figure 4.6 *Left:* The baton before it is thrown. "X" marks the COM. *Center:* The initial conditions for the baton as it is thrown. *Right:* The baton spinning in the air under the action of gravity.



4.10 TRAJECTORY OF A THROWN BATON (PROBLEM)

We wish to describe the trajectory of a baton that spins as it travels through the air. On the left in Figure 4.6 the baton is shown as as two identical spheres joined by a massless bar. Each sphere has mass m and radius r, with the centers of the spheres separated by a distance L. The baton is thrown with the initial velocity (Figure 4.6 center) corresponding to a rotation about the center of the lower sphere.

Problem: Write an OOP program that computes the position and velocity of the baton as a function of time. The program should

1. plot the position of each end of the baton as a function of time;
2. plot the translational kinetic energy, the rotational kinetic energy, and the potential energy of the baton, all as functions of time;
3. use several classes as building blocks so that you may change one building block without affecting the rest of the program;
4. (optional) then be extended to solve for the motion of a baton with an additional lead weight at its center.

4.10.1 Combined Translation and Rotation (Theory)

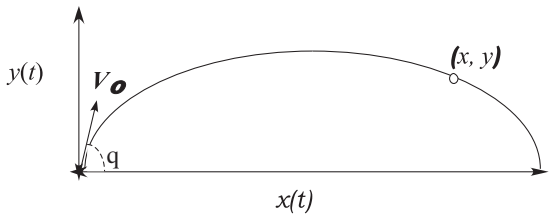
Applet Classical dynamics describes the motion of the baton as the motion of its center of mass (COM) (marked with an "X" in Figure 4.6), plus a rotation about the COM (Figure 4.6 right). Because the translational and rotational motions are independent, each may be determined separately, and because we ignore air resistance, the angular velocity ω about the COM is constant.

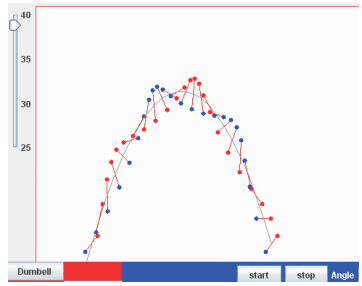
The baton is thrown with an initial velocity (Figure 4.6 center). The simplest way to view this is as a translation of the entire baton with a velocity $\mathbf{V}_0$ and a rotation of angular velocity ω about the COM. To determine ω, we note that the tangential velocity due to rotation is

$$v_t = \frac{1}{2}\omega L. \tag{4.20}$$

For the direction of rotation as indicated in Figure 4.6, this tangential velocity is added to the COM velocity at the top of the baton and is subtracted from the COM velocity at the bottom.

Figure 4.7 *Left:* The trajectory $[x(t),\ y(t)]$ followed by the baton's COM. *Right:* The applet **JParabola.java** showing the entire baton as its COM follows a parabola. Your eye tends to notice the spinning and not the parabola.





Because the total velocity equals 0 at the bottom and $2\mathbf{V}_0$ at the top, we are able to solve for ω:

$$\frac{1}{2}\omega L - V_0 = 0 \quad \Rightarrow \quad V_0 = \frac{1}{2}\omega L, \quad \Rightarrow \quad \omega = \frac{2V_0}{L}. \tag{4.21}$$

If we ignore air resistance, the only force acing on the baton is gravity, and it acts at the COM of the baton (Figure 4.6 right). Figure 4.7 shows a plot of the trajectory $[x(t), y(t)]$ of the COM:

$$(x_{\rm cm},\ y_{\rm cm}) = \left(V_{0x}t,\ V_{0y}t - \frac{1}{2}gt^2\right), \quad (v_{x,\rm cm},\ v_{y,\rm cm}) = (\,V_{0x},\ V_{0y} - gt\,),$$

where the horizontal and vertical components of the initial velocity are

$$V_{0x} = V_0\ \cos\,\theta, \quad V_{0y} = V_0\ \sin\,\theta.$$

Even though ω = constant, it is a constant about the COM, which itself travels along a parabolic trajectory. Consequently, the motion of the baton's ends may appear complicated to an observer on the ground (Figure 4.7 right). To describe the motion of the ends, we label one end of the baton a and the other end b (Figure 4.6 left). Then, for an angular orientation ϕ of the baton,

$$\phi(t) = \omega t + \phi_0 = \omega t, \tag{4.22}$$

where we have taken the initial $\phi = \phi_0 = 0$. Relative to the COM, the ends of the baton are described by the polar coordinates

$$(r_a, \phi_a) = \left(\frac{L}{2}, \phi(t)\right), \quad (r_b, \phi_b) = \left(\frac{L}{2}, \phi(t) + \pi\right). \tag{4.23}$$

The ends of the baton are also described by the Cartesian coordinates

$$(x'_a, y'_a) = \frac{L}{2}\,[\cos\omega t,\ \sin\omega t]\,, \quad (x'_b, y'_b) = \frac{L}{2}\,[\cos(\omega t + \pi),\ \sin(\omega t + \pi)]\,.$$

The baton's ends, as seen by a stationary observer, have the vector sum of the position of the COM plus the position relative to the COM:

$$(x_a,\ y_a) = \left[V_{0x}t + \frac{L}{2}\cos(\omega t),\ V_{0y}t - \frac{1}{2}gt^2 + \frac{L}{2}\sin(\omega t)\right], \tag{4.24}$$

$$(x_b,\ y_b) = \left[V_{0x}t + \frac{L}{2}\cos(\omega t + \pi),\ V_{0y}t - \frac{1}{2}gt^2 + \frac{L}{2}\sin(\omega t + \pi)\right].$$

If L_a and L_b are the distances of m_a and m_b from COM, then

$$L_a = \frac{m_b}{m_a + m_b}, \quad L_b = \frac{m_a}{m_a + m_b}, \quad \Rightarrow \qquad m_a L_a = m_b L_b. \tag{4.25}$$

The moment of inertia of the masses (ignoring the bar connecting them) is

$$I_{\text{masses}} = m_a L_a^2 + m_b L_b^2. \tag{4.26}$$

If the bar connecting the masses is uniform with mass m and length L, then it has a moment of inertia about its COM of

$$I_{\text{bar}} = \frac{1}{12} m L^2. \tag{4.27}$$

Because the COM of the bar is at the same location as the COM of the masses, the total moment of inertia for the system is just the sum of the two:

$$I_{\text{tot}} = I_{\text{masses}} + I_{\text{bar}}. \tag{4.28}$$

The potential energy of the masses is

$$\text{PE}_{\text{masses}} = (m_a + m_b) g h = (m_a + m_b) g \left(V_0 \, t \sin\theta - \frac{1}{2} g t^2 \right), \tag{4.29}$$

while the potential energy of the bar just has $m_a + m_b$ replaced by m since both share the same COM location. The rotational kinetic energy of rotation is

$$\text{KE}_{\text{rot}} = \frac{1}{2} I \omega^2, \tag{4.30}$$

with ω the angular velocity and I the moment of inertia for either the masses or the bar (or the sum). The translational kinetic energy of the masses is

$$\text{KE}_{\text{trans}} = \frac{1}{2} m \left[(V_0 \, \sin\theta - g \, t)^2 + (V_0 \, \cos\theta)^2 \right], \tag{4.31}$$

with $m_a + m_b$ replaced by m for the bar's translational KE.

Applet To get a feel for the interestingly complicated motion of a baton, we recommend that the reader try out the applet **JParabola** (Fig. 4.7 right).

```
% appletviewer jcenterofmass.html
```

4.11 OOP DESIGN CONCEPTS (CS)

In accord with our belief that much of education is just being an understanding of what the words mean, we start by defining OOP as programming containing component objects with four characteristics [Smi 91]:

Encapsulation: The data and the *methods* used to produce or access data are encapsulated into entities called *objects.* For our problem, the data are initial positions, velocities, and properties of a baton, and the objects are the baton and its path. As part of the OOP philosophy, data are manipulated only via distinct methods.

Abstraction: Operations applied to objects are assumed to yield standard results according to the nature of the objects. To illustrate, summing two matrices always gives another matrix. By incorporating abstraction into programming, we concentrate more on solving the problem and less on the details of the implementation.

Inheritance: Objects inherit characteristics (including code) from their ancestors yet may be different from their ancestors. A baton inherits the motion of a point particle, which in this case describes the motion of the COM, and extends that by permitting rotations about

the COM. In addition, if we form a red baton, it inherits the characteristics of a colorless baton but with the property of color added to it.

Polymorphism: Methods with the same name may affect different objects differently. Child objects may have *member* functions with the same name but with properties differing from those of their ancestors (analogous to method overload, where the method used depends upon the method's arguments).

4.12 MULTIPLE CLASSES AND MULTIPLE INHERITANCES

We now solve our baton problem using OOP techniques. Although we can also solve it with the traditional techniques of procedural programming, this problem contains the successive layers of complexity that make it appropriate for OOP. We will use several source (`.py`) files for this problem, each yielding a different class corresponding to a different aspect of the baton. There is a class `Ball.py` (Listing 4.8) representing a ball on the end of the baton, a class `Path.py` (Listing 4.9) representing the trajectory of the COM, and a class `Baton.py` (Listing 4.10) assembling the other classes into an object representing the baton.

Listing 4.8 The class **Ball** representing the ball on the end of the baton.

```
# Ball.py: Isolated ball with Mass and Radius for OOP

class Ball:
    def __init__(self, mass, radius):          # ball class constructor
        self.m = mass                                 # initialize mass
        self.r = radius                             # initialize radius

    def getM1(self):                                   # get ball mass
        return self.m

    def getR(self):                                         # get radius
        return self.r                            # get moment of inertia

    def getI1(self):
        return (2.0/5.0)*self.m*(self.r)**2
```

The `Ball` class (Listing 4.8) creates a ball object of mass m, radius r, and moment of inertia I. It contains the three dynamic and short methods, `getM`, `getR`, and `getI` for extracting the mass, radius, and moment of inertia of a ball. Inasmuch as we use `Ball` as a building block, it is a good idea to keep the methods simple, and just add more methods to create more complicated objects. Take stock of how similar `Ball` is to the `Complex` class. In the present case, `m` and `r` behave like the `re` and `im` in the `Complex` class in that they too act by being attached to the end of an object's name.

Dynamic methods are like dynamic variables in that they behave differently depending on the object they modify. To cite an instance, `ball1.getR()` and `ball2.getR()` return different values if the balls have different radii. The methods `getM` and `getR` are *template* methods; that is, they do not compute anything now but are included to facilitate future extensions. To name an instance, if the `Ball` class becomes more complex, you may need to sum the masses of its constituent parts in order to return the ball's total mass. With the template in place, you do that without having to reacquaint yourself with the rest of the code first. This testing code creates a `Ball` object and prints out its properties by affixing `get` methods to the object.

Exercises: Compile and run the modified `Ball.py` to ensure that the `Ball` class still works properly. Test the class `Path`. Create a `Path` object and find its properties at several different times. Because this a test code, it does not need to do much; being able to make the object with expected properties is enough.

Listing 4.9 The class **Path** creating an object that represents the trajectory of the center of mass.

```
# Path.py: Parabolic Trajectory of COM for OOP

import math
class Path:                                                  # Path of COM mass

    def __init__(self, v0, theta):                  # constructor for parabola
        self.g = 9.8                                                  # gravity
        self.v0 = v0                                            # initial speed
        self.theta = theta                                      # initial angle
        self.v0x = self.v0*math.cos(self.theta*math.pi/180.0)          # Vx
        self.v0y = self.v0*math.sin(self.theta*math.pi/180.0)          # Vy

    def getX(self, t):                                            # get Xcom
        self.t = t
        return self.v0x*self.t

    def getY(self, t):                                            # get Ycom
        self.t = t
        return self.v0y*self.t - 0.5*self.g*t**2
```

The class **`Path`** (Listing 4.9) creates an object that represents the trajectory $[(x(t), y(t)]$ of the COM. This class computes the initial velocity components V_{0x} and V_{0y}, and is another building block that we will use to construct the baton's trajectory. The acceleration due to gravity g, which is the same for all objects, is associated with **`Path`**. The constructor method for the class **`Path`** takes the polar coordinates (V_0, θ) as arguments and computes the components of the initial velocity, (**`v0x, v0y`**).

Listing 4.10 **Baton.py** combines ball and path classes to form a baton.

```
# Baton.py: Combine Ball and Path classes to form Baton

from visual.graph import *                      # graphics and math classes
import Ball, Path                                   # import other classes

class Baton(Ball.Ball, Path.Path):   # Baton inherits Ball and Path props
    def __init__(self, mass, radius, v0, theta, Ll, wl): #construct Baton
        Ball.Ball.__init__(self, mass, radius)            # construct Ball
        Path.Path.__init__(self, v0, theta)               # construct Path
        self.L = Ll                                          # Baton length
        self.w = wl                                     # Baton ang velocity

    def getM(self):
        return 2.0*self.getM1()

    def getI(self):
        return (2*self.getI1() + 0.5*self.getM()*self.L**2)

    def getXa(self, t):
        xa = self.getX(t) + 0.5*self.L*cos(self.w*t)
        return xa

    def getYa(self, t):
        return self.getY(t) + 0.5*self.L*sin(self.w*t)

    def getXb(self, t):
        return self.getX(t) - 0.5*self.L*cos(self.w*t)

    def getYb(self, t):
        return self.getY(t) - 0.5*self.L*sin(self.w*t)

    def scenario(self, mytitle, myxtitle, myytitle, xma, xmi, yma, ymi):
        graph = gdisplay(x = 0, y = 0, width = 500, height = 500,
                title=mytitle, xtitle=myxtitle, ytitle=myytitle, xmax=xma,
                xmin=xmi, ymax=yma, ymin=ymi, foreground=color.black,
                background = color.white)
    def position(self):
        batonmassa = gcurve(color=color.blue)          # blue trajectory a
        batonmassb = gcurve(color=color.red)            # red trajectory b
        batoncm = gcurve(color = color.magenta)

        t = 0.0                                     # start motion at time 0
        count = 4
        yy = self.getYa(t)                                   # initial y a
        while (self.getYa(t)>= 0.0):                       # do till y Yb <0
            xa = self.getXa(t)                                        # Xa
```

```
            ya = self.getYa(t)                                          # Yb
                                                                  # plot a
            xb = self.getXb(t)
            yb = self.getYb(t)
            points=[(xa,ya),(xb,yb)]
            gcurve(color=(0.8,0.8,0.8),pos=points)
            batonmassa.plot(pos = (xa, ya) )
            batonmassb.plot(pos = (xb, yb) )                      # plots b

            # if count%4 == 0: # plots baton if uncommented  next 2 lines
                # gcurve(pos = [(xa, ya), (xb, yb)], color=color.cyan)
            xcm = self.getX(t)                                          # Xcom
            ycm = self.getY(t)                                          # Ycom
            rate(5)
            batoncm.plot(pos = (xcm, ycm) )                       # plot COM
            t  += 0.02                                      # increment time
            count   += 1

mybaton = Baton(0.5, 0.4, 15.0, 34.0, 2.5, 15.0)        # m r v0 theta L w
                #next title and geometry for the graph
mybaton.scenario('Positions of mass a(blue), b(red) and COM (magenta)',
                 'x', 'y', 20, 0, 5, -1)                  # xmx=20, xmin=0
mybaton.position()                                       # ymax=10, ymin=-1
```

Now that we have created the building-block classes, we combine them to create the baton's trajectory in **`Baton.py`** (Listing 4.10). The class **`Baton`** inherits the properties of mass, radius and moment of inertia from the **`Ball`** class, as well as the methods **`getR, getM`** and **`getI`**. In addition to its inherited properties, the **`Baton`** class adds properties describing the length L and the angular velocity w of the bar, and methods **`getXa`**, **`getYa`**, **`getXb`** and **`getYb`** that extract the positions of the bar ends. The method **`scenario`** set up the plot, using **`position`** to locate the balls. Notice how the constructor **`Baton(Path p, Ball b, ...)`** takes the **`Path`** and **`Ball`** objects as arguments and constructs the **`Baton`** object from them. This is interesting because the **`Ball`** and **`Path`** belong to classes not included here. For this to work, we must place the **`Ball`** and **`Path`** class files in the directory in which we are creating a **`Baton`**. The Python compiler will then find them.

Next we define the **`get`** methods for manipulating baton objects. This is the standard way of indicating that a method will retrieve or extract some property from the object to which the method is appended. For instance, **`Baton.getM`** returns **`2m`**, that is, the sum of the masses of the two spheres, while **`getI`** returns the moment of inertia. The methods **`getXa`**, **`getYa`**, **`getXb`**, and **`getYb`** take time t as an argument and return the coordinates of the baton's ends at that time. These methods first determine the position of the COM by calling **`path.getX`** or **`path.getY`**, and then add on the relative coordinates of the ends.

Listing 4.11 **Dumbbell.py** combines ball and path classes to form a dumbbell.

```
# Dumbbell.py: Combines classes to form a Baton with Mass and Radius

from visual.graph import *                      # graphics and math classes
class Ball:
    def __init__(self, mass, radius):                   # Ball constructor
        self.m = mass                                    # initialize mass
        self.r = radius                                 # initialize radius

    def getM1(self):                                        # get Ball mass
        return self.m

    def getR(self):                                        # get Ball radius
        return self.r

    def getI1(self):
        return (2.0/5.0)*self.m*(self.r)**2

class Path:                                         # parabolic path of COM
    def __init__(self, v0, theta):                      # Path constructor
        self.g = 9.8                                              # gravity
        self.v0 = v0                                        # initial speed
        self.theta = theta                                  # initial angle
```

```
        self.v0x = self.v0*cos(self.theta*pi/180.0)          # initial Vx
        self.v0y = self.v0*sin(self.theta*pi/180.0)          # initial Vy

    def getX(self, t):                                         # get Xcom
        self.t = t
        return self.v0x*self.t

    def getY(self, t):                                         # get Ycom
        self.t = t
        return self.v0y*self.t - 0.5*self.g*t**2

class Baton(Ball, Path):          # Baton inherints Ball & Path properties
    def __init__(self, mass, radius, v0, theta, L1, w1):
        Ball.__init__(self, mass, radius)                # contructs Ball
        Path.__init__(self, v0, theta)              # constructs classPath
        self.L = L1                                     # Lenght of  Baton
        self.w = w1                                   # ang velocity Baton

    def getM(self):
        return 2.0*self.getM1()

    def getI(self):
        return (2*self.getI1() + 0.5*self.getM()*self.L**2)

    def getXa(self, t):
        xa = self.getX(t) +  0.5*self.L*cos(self.w*t)
        return xa

    def getYa(self, t):
        return self.getY(t) +  0.5*self.L*sin(self.w*t)

    def getXb(self, t):
        return self.getX(t) -  0.5*self.L*cos(self.w*t)

    def getYb(self, t):
        return self.getY(t) -  0.5*self.L*sin(self.w*t)

    def scenario(self, mytitle, myxtitle, myytitle, xma, xmi, yma, ymi):
        graph = gdisplay(x = 0, y = 0, width = 500, height = 500,
                title=mytitle, xtitle=myxtitle, ytitle=myytitle, xmax=xma,
                xmin=xmi, ymax=yma, ymin=ymi, foreground=color.black,
                background=color.white)

    def position(self):
        batonmassa = gcurve(color = color.blue)       # blue a trajectory
        batonmassb = gcurve(color = color.red)         # red b trajectory
        batoncm = gcurve(color = color.magenta)
        t = 0.0                                     # start motion at t=0
        count = 4
        yy = self.getYa(t)                                  # initial Ya
        while (self.getYa(t)>= 0.0):                      # do till Yb <0
            xa = self.getXa(t)                                      # Xa
            ya = self.getYa(t)                                      # Yb
            batonmassa.plot(pos = (xa, ya) )                     # plot a
            xb = self.getXb(t)
            yb = self.getYb(t)
            batonmassb.plot(pos = (xb, yb) )                     # plot b
            # if count%4 == 0:
                                  #  uncommented 2 two lines to plot baton
                # gcurve(pos = [(xa, ya), (xb, yb)], color = color.cyan)
            xcm = self.getX(t)                                     # Xcom
            ycm = self.getY(t)                                     # Ycom
            batoncm.plot(pos = (xcm, ycm) )                   # plot COM
            rate(10)
            t  += 0.02                          # increments time in 0.02
            count   += 1

mybaton = Baton(0.5, 0.4, 15.0, 34.0, 2.5, 15.0)  # m radius v0 theta L w
mybaton.scenario('Positions of mass a(blue), b(red) and COM (magenta)',
                 'x', 'y', 20, 0, 5, -1)            # xmx = 20, xmin = 0
mybaton.position()                                # ymax = 10, ymin = -1
```

4.13 MULTIPLE INHERITANCES, CLASSES IN ONE FILE

When dealing with multiple classes, you have the choice of placing all the classes in the same file, or having each class in a different file. While the former is more self-contained, the latter is more flexible since then the classes can then be used by a variety of programs. The process of inheritance from multiple classes differs for each point of view. For example, the

program **Dumbbell.py** in Listing 4.11 has all the **Ball**, **Path** and **Baton** classes in the same file. Individual files with each class are also given in the same directory. The inheritance of the two classes begins with the statement

```
class Baton(Ball, Path):                                                    37
```

in which the **Ball** and **Path** class names are passed as arguments to **Baton**. Note next that the constructor of **Baton** has six arguments, besides **self**,

```
def __init__(self, mass, radius, v0, theta, L1, w1):                       39
```

The arguments **mass** and **radius** are used to initialize **Ball**, the arguments **v0** and **theta** are used to initialize **Path**, while **L1** and **w1** are used to initialize **Baton**:

```
Ball.__init__(self,mass,radius)                                  Construct Ball
Path.__init__(self,v0,theta)                                     Construct Path
self.L = L1                                                      Length of Baton
self.w = w1                                                      Ang velocity of Baton
```

Now **Baton** can use the methods of **Ball** and **Path**, as if they were its own:

```
def getXa(self,t):                                                          52
      xa = self.getX(t) + 0.5 * self.L * math.cos(self.w*t)                 53
```

Here the x position is obtained via the method **getX** defined in **Path**. The last lines (92–95) in the program define an instance **myBaton** of the class **Baton** and initializes the three classes using **Baton**'s methods **scenario** and **position**.

4.14 MULTIPLE INHERITANCE, SEPARATE FILES

The same **OOP** directory we have been using also contains the separate class files **Ball.py**, **Path.py** and **Baton.py**. When used as separate files, the classes **Ball** and **Path** have to be compiled to produce the class (byte) files **Ball.pyc** and **Path.pyc**. This is done automatically by IDLE when the *Run/Run Module* item is selected for **Baton.py**, which contains the critical statement

```
import Ball, Path                                                # import other classes
```

Then **Baton** will look for these **.pyc** files when it imports **Ball** and **Path**.

Note next the unusual way in which the **Ball** and **Path** classes are initialized with the use of double names:

```
class Baton(Ball.Ball,Path.Path):        # Baton inherits Ball and Path properties
   def __init__(self,mass,radius,v0,theta,L1,w1):
       Ball.Ball.__init__(self,mass,radius)          # construct Ball
       Path.Path.__init__(self,v0,theta)             # construct Path
       self.L=L1                                    # length of Baton
       self.w=w1
```

The first of the double names refers to the class, the second to the object. The rest of the program is the same as before and the results are identical.

The technique of constructing complex objects from simpler ones is called *composition.* As a consequence of the simple objects being contained within the more complex ones, the former are described as *active class variables*. This means that their properties change depending upon which object they are within. When you use composition to create more complex objects, you are working at a *higher level of abstraction.* Ideally, composition hides the distracting details of the simpler objects from your view so that you focus on the major task to be

accomplished. This is analogous to first designing the general form of a bridge before worrying about the colors of the cables to be used.

4.14.1 Composition Exercise

1. Compile and run the latest version of **`Baton.py`**. The program should run and plot the trajectory of one end of the baton as it travels through the air (Figure 4.7).
2. Plot the trajectory of end b on the same graph that shows the trajectory of end a, but in a different color. You may do this by copying and pasting the **`for`** loop for a and then modifying it for b.
3. Change the mass of the **`ball`** in the **`Baton`** constructor method to some large number, for instance, 50 kg. Add print statements in the constructor and the main program to show how the **`ball`** class variable and the **`myBall`** object are affected by the new mass. You should find that **`ball`** and **`myBall`** both reference the same object since they both refer to the same memory location. In this way changes to one object are reflected in the other object.

In Python, an object is passed between methods and manipulated by *reference*. This means that its address in memory is passed and not the actual values of all the component parts of it. On the other hand, primitive data types like **`int`** and **`double`** are manipulated by *value*. At times we may actually say that objects are references. This means that when one object is set equal to another, both objects *point* to the same location in memory (the start location of the first component of the object). Therefore all three variables **`myBall`**, **`p`**, and **`q`** refer to the same object in memory. If we change the mass of the ball, all three variables will reflect the new mass value. This also works for object arguments: If you pass an object as an argument to a method and the method modifies the object, then the object in the calling program will also be modified.

4.14.2 Calculating the Baton's Energy (Extension)

Extend your classes so that they plot the energy of the baton as a function of time. Plot the kinetic energy of translation, the kinetic energy of rotation, and the potential energy as functions of time:

1. The translational kinetic energy of the baton is the energy associated with the motion of the center of mass. Write a **`getKEcm`** method in the **`Baton`** class that returns the kinetic energy of translation $\mathrm{KE}_{\mathrm{cm}}(t) = mv_{\mathrm{cm}}(t)^2/2$. In terms of pseudocode the method is

```
Get present value of Vx.
Get present value of Vy.
Compute V^2 = Vx^2 + Vy^2.
Return mV^2/2.
```

   Before you program this method, write **`getVx`** and **`getVy`** methods that extract the COM velocity from a baton. Seeing as how the **`Path`** class already computes $x_{\mathrm{cm}}(t)$ and $y_{\mathrm{cm}}(t)$, it is the logical place for the velocity methods. As a guide, we suggest consulting the **`getX`** and **`getY`** methods.
2. Next we need the method **`getKEcm`** in the **`Baton`** class to compute $\mathrm{KE}_{\mathrm{cm}}(t)$. Inasmuch as the method will be in the **`Baton`** class, we may call any of the methods in **`Baton`**, as well as access the **`path`** and **`ball`** subobjects there (subobjects because they reside inside the

Baton object or class). We obtain the velocity components by applying the **getV** methods to the **path** subobject within the **Baton** object:

```
def getKEcm(self,t):                                    # COM translational KE
    return self.getM()*((self.getVx(t))**2 +(self.getVy(t))**2)/2.
```

Even though the method is in a different class than the object, Python handles this. Study how **getM()**, being within **getKEcm**, acts on the same object as does **getKEcm** without explicitly specifying the object.

3. Compile the modified **Baton** and **Path** classes.
4. Modify **Baton.py** to plot the translational kinetic energy of the center of mass as a function of time. Comment out the old **for** loops used for plotting the positions of the baton's ends and add the code

```
def energies(self):
    batonKEr = gcurve(color=color.blue) # blue: rotation KE
    batonPE = gcurve(color=color.green)          # green: PE
    batonKEcm = gcurve(color=color.magenta) # magenta: KEcm
    batonEtotal = gcurve(color=color.red)    # red: total E
    t = 0.0#start motion at time 0
    yy = self.getYa(t)                          # initial Ya
    while (self.getYa(t)>=0.0):              # do till Ya < 0
        KEcm = self.getKEcm(t)                        # KEcom
        PE = self.getPE(t)                              # PE
        KEr = self.getKEr(t)                 # rotational KE
        xcm=self.getX(t)                              # Xcom
        Etot = KEcm + PE + KEr                      # total E
        batonKEcm.plot( pos=(xcm, KEcm) )        # plot KEcom
        batonKEr.plot( pos=(xcm, KEr) )          # plot KErot
        batonEtotal.plot( pos=(xcm, Etot) )  # plot total E
        batonPE.plot( pos=(xcm, PE) )
        t += 0.02                                # increment t
```

Compile the modified **Baton.py** and check that your plot is physically reasonable. The translational kinetic energy should decrease and then increase as the baton goes up and comes down.

5. Write a method in the **Baton** class that computes the kinetic energy of rotation about the COM, $\mathrm{KE}_{ro} = \frac{1}{2}I\omega^2$. Call **getI** to extract the moment of inertia of the baton and check that all classes still compile properly.
6. The potential energy of the baton $\mathrm{PE}(t) = mgy_{\mathrm{cm}}(t)$ is that of a point particle with the total mass of the baton located at the COM. Write a method in the **Baton** class that computes PE. Use **getM** to extract the mass of the baton and use **path.g** to extract the acceleration due to gravity g. To determine the height as a function of time, write a method **path.getY(t)** that accesses the **path** object. Make sure that the methods **getKEcm**, **getKEr**, and **getPE** are in the **Baton** class.
7. Plot on one graph the translational kinetic energy, the kinetic energy of rotation, the potential energy, and the total energy.
   Check that the total energy and rotational energies remain constant in time. However, the gravitational potential energy should fluctuate.

4.14.3 Examples of Inheritance and Object Hierarchies

Up until this point we have built up our classes via *composition*, that is, by placing objects inside other objects (using objects as arguments). As powerful as composition is, it is not appropriate in all circumstances. As a case in point, you may want to modify the **Baton** class to create similar, but not identical, objects such as 3-D batons. A direct approach to extending the program would be to copy and paste parts of the original code into a new class and then modify the new class. However, this is error-prone and leads to long, complicated, repetitive code. The OOP approach applies the concept of *inheritance* to allow us to create new objects

that inherit the properties of old objects but have additional properties as well. This is how we create entire hierarchies of objects.

As an example, let us say we want to place red balls on the ends of the baton. We make a new class **RedBall** that inherits properties from **Ball**:

```
class RedBall(Ball):                                # Defines RedBall as subclass of Ball
    . . .
```

As written, this code creates a **RedBall** class that is identical to the original **Ball** class. The key word **extends** tells Java to copy all the methods and variables from **Ball** into **RedBall**. In OOP terminology, the **Ball** class is the *parent class* or *superclass*, and the **RedBall** class is the *child class* or *subclass*. It follows then that a class hierarchy is a sort of family tree for classes, with the parent classes at the top of the tree. As things go, children beget children of their own and trees often grow high.

To make **RedBall** different from **Ball**, we add the property of color:

```
class RedBall(Ball):                                # Defines RedBall as subclass of Ball

    def __init__(self, mass, radius):                           # RedBall Constructor
        Ball.__init__(self, mass, radius)                          # Ball Constructor

    def getColor(self):                                     # New property of subclass
        return 'Red'

    def getR(self):                                     # Overrides Ball's getR method
        return 1.0
```

We see that the child class receives **mass** and **radius** as arguments in addition to **self**, and transfers them to **Ball** via its constructor. In addition there are the two methods **getColor** and **getR** that overrides the **Ball** methods with the same name. This means that **RedBall** inherits **mass** and **radius** from **Ball**, and that the new **getR** method *overrides* the original method.

4.14.4 Baton with a Lead Weight (Application)

As a second example, we employ inheritance to create a class of objects representing a baton with a weight at its center. We call this new class **LeadBaton.py** and make it a child of the parent class **Baton.py**. Consequently, the **LeadBaton** class inherits all the methods from the parent **Baton** class, in addition to having new ones of its own:

Listing 4.12 **LeadBaton.py** inherits ball and path properties from `Baton`.

```
# LeadBaton.py   inherits methods from baton, e.g. OOP

from visual.graph import *                    # graphics and math classes
from sys import version
if int(version[0])>2:   # raw_input deprecated in Python 3
    raw_input=input
import Ball, Path
'''LeadBaton class inheritance of methods from Baton'''

class Baton(Ball.Ball, Path.Path):# Baton inherits Ball & Path properties

    def __init__(self, mass, radius, v0, theta, Ll, wl):
        Ball.Ball.__init__(self, mass, radius)            # contruct Ball
        Path.Path.__init__(self, v0, theta)              # construct Path
        self.L = Ll                                    # Lenght of  Baton
        self.w = wl                                       # ang velocity

    def getM(self):
```

```
        return 2.0*self.getM1()

    def getI(self):
        return (2*self.getI1() + 0.5*self.getM()*self.L**2)

    def getXa(self, t):
        xa = self.getX(t) +  0.5*self.L*cos(self.w*t)
        return xa

    def getYa(self, t):
        return self.getY(t) +  0.5*self.L*sin(self.w*t)

    def getXb(self, t):
        return self.getX(t) -  0.5*self.L*cos(self.w*t)

    def getYb(self, t):
        return self.getY(t) -  0.5*self.L*sin(self.w*t)

    def scenario(self, mytitle, myxtitle, myytitle, xma, xmi, yma, ymi):
        graph = gdisplay(x = 0, y = 0, width = 500, height = 500,
                title=mytitle, xtitle=myxtitle, ytitle=myytitle, xmax=xma,
                xmin=xmi, ymax=yma, ymin=ymi, foreground=color.black,
                background = color.white)

    def position(self):
        batonmassa = gcurve(color = color.blue)      # blue trajectory  a
        batonmassb = gcurve(color = color.red)        # red trajectory  b
        batoncm = gcurve(color = color.magenta)
        t = 0.0                                  # start motion at time 0
        count = 4
        yy = self.getYa(t)                    # initial y position mass a
        while (self.getYa(t)>= 0.0):                      # do till Yb <0
            xa = self.getXa(t)                                     # Xa
            ya = self.getYa(t)                                    #  Yb
            batonmassa.plot(pos = (xa, ya) )                    # plot a
            xb = self.getXb(t)
            yb = self.getYb(t)
            batonmassb.plot(pos = (xb, yb) )                    # plot b
            # if count%4 == 0:  # plots baton if uncommented next 2 lines
                # gcurve(pos = [(xa, ya), (xb, yb)], color = color.cyan)
            xcm = self.getX(t)                                    # Xcom
            ycm = self.getY(t)                                    # Ycom
            batoncm.plot(pos = (xcm, ycm) )                  # plots COM
            t  += 0.02                                   # increments t
            count   += 1

class LeadBaton(Baton):

    def __init__(self, mass, radius, v0, theta, L1, w1, M1):# Constructor
        "To initialize Baton"
        Baton.__init__(self, mass, radius, v0, theta, L1, w1)# init Baton
        self.MM = M1 + self.getM1()              # lead mass + Baton mass
    def getM(self):
        return self.MM

myLeadBaton = LeadBaton(0.5, 0.4, 15.0, 34.0, 2.5, 15.0, 35.0)
print( "myLeadBaton mass = ", myLeadBaton.getM())
print( "Enter any character to finish")
s=raw_input()
```

The first line declares **LeadBaton** to be a subclass of **Baton**. The second line contains the constructor for a **LeadBaton** object, with the first six arguments (not counting **self**) needed to initialize the **Baton**, and the last argument being the lead mass. Note next that the **getM()** method for **LeadBaton** overrides the method with the same name in **Baton**, and returns the **LeadBaton** mass. Finally, the instance of **myLeadBaton** is built with all seven parameters (besides **self**) and with the mass property of **LeadBaton**.

Exercise:

1. Run and create plots from the **LeadBaton** class. Instead of creating **Baton**, now create a **LeadBaton** with

   ```
   LeadBaton myBaton = new LeadBaton(myPath, myBall, 2.5, 15., 10.);
   ```

Here the argument "`10.`" describes a 10-kg mass at the center of the baton.

2. Run the program and check that you get a plot of the energies of the lead baton *versus* time. Compare its energy to that of an ordinary baton and comment on the differences.
3. You should see now how OOP permits us to create many types of batons with only slight modifications of the code. You can switch between a **`Baton`** and a **`LeadBaton`** object with only a single change, a modification that would be significantly more difficult with procedural programming. ▌

4.14.5 Encapsulation to Protect Classes

In the previous section we created the classes for **`Ball`**, **`Path`**, and **`Baton`** objects. In all cases the Python source code for each class had the same basic structure: class variables, constructors, and **`get`** methods. Yet classes do different things, and it is common to categorize the functions of classes as either of the following.

Interface: How the outside world manipulates an object; all methods that are applied to that object; or

Implementation: The actual internal workings of an object; how the methods make their changes.

As applied to our program, the interface for the **`Ball`** class includes a constructor **`Ball`** and the **`getM`**, **`getR`**, and **`getI`** methods. The interface for the **`Path`** class includes a constructor **`Path`** and the **`getX`** and **`getY`** methods.

Pure OOP strives to keep objects abstract and to manipulate them only through methods. This makes it easy to follow and to control where variables are changed and thereby makes modifying an existing program easier and less error-prone. With this purpose in mind, we separate methods into those that perform calculations and those that cause the object to do things.

As programmers, we may rewrite an object's code as we want and still have the same working object with a *fixed interface* for the rest of the world. Furthermore, since the object's interface is constant, even though we may change the object, there is no need to modify any code that uses the object. This is a great advance in the ability to reuse code and to use other people's codes properly.

This two-step process of creating and protecting abstract objects is known as *encapsulation*. An encapsulated object may be manipulated only in a general manner that keeps the irrelevant details of its internal workings safely hidden within. Just what constitutes an "irrelevant detail" is in the eye of the programmer. In general, you should place the **`private`** key word before every nonstatic class variable and then write the appropriate methods for accessing the relevant variables. This OOP process of hiding the object's variables is called *data hiding*.

4.14.6 Encapsulation Exercise

1. Place the **`private`** key word before all the class variables in **`Ball.py`**. This accomplishes the first step in encapsulating an object. Print out **`myBall.m`** and **`myBall.r`**.

2. Create methods that allow you to manipulate the object in an abstract way; for example, to modify the mass of a **`Ball`** object and assign it to the private class variable **`m`**, include the command **`myBall.setM(5.0)`**. This is the second step in encapsulation. We already have the methods **`getM`**, **`getR`**, and **`getI`**, and the object constructor **`Ball`**, but they do not assign a mass to the ball. Insofar as we have used a method to change the private variable **`m`**, we have kept our code as general as possible and still have our objects encapsulated.
3. When we write **`getM()`**, we are saying that **`M`** is the *property* to be retrieved from a **`Ball`** object. Inversely, the method **`setM`** sets the property **`M`** of an object equal to the argument that is given. This is part of encapsulation because with both **`get`** and **`set`** methods on hand, you do not need to access the class variables from outside the class. The use of **`get`** and **`set`** methods is standard practice in Python. You do not have to write **`get`** and **`set`** methods for every class you create, but you should create these methods for any class you want encapsulated.

4.14.7 Complex Object Interface (Extension)

In Listing 4.13 we display **`KomplexPolar.py`**, our design for an interface for complex numbers. In Listing 4.14 we give **`KomplexPolarTest.py`**, a program that tests our interface. To avoid confusion with **`Complex`** objects, we call the new object **`Komplex`**. We include methods for addition, subtraction, multiplication, division, negation, and conjugation, as well as **`get`** and **`set`** methods for the real, imaginary, modulus, and phase. Remember, an interface must give the arguments and return type for each method.

Listing 4.13 **KomplexPolar.py** is an interface for complex numbers and is used in `KomplexPolarTest.py`.

```
''' KomplexPolar: complex numbes via modulus
    Cartesian/polar complex via package
    "typec = 0" -> polar representation, else: rectangular'''

import math

class Komplex:

    def __init__(self, x, y, typec):                # Kopmplex constructor
        if typec == 0:                                  # 0: for polar rep
            self.mod = x                                        # magnitude
            self.theta = y                                    # polar angle
        else:                                     # rectangular representation
           self.re = x
           self.im = y

    def getRe(self):                          # return rectangular real part
        return self.mod*math.cos(self.theta)

    def getIm(self):                          # return rectangular imag part
        return self.mod*math.sin(self.theta)

    def setRe(self):                         # returns rectangular real part
        re = self.mod*math.cos(self.theta)
        return re

    def setIm(self):                          # return rectangular imag part
        im = self.mod*math.sin(self.theta)
        return im

    def add(self, other, typec):                  # add two complex numbers
        if typec == 0:                                 # polar representation
            tempMod = math.sqrt(self.mod*self.mod + other.mod*other.mod
            + 2*self.mod*other.mod*math.cos(self.theta - other.theta) )
            angtheta  = math.atan2(self.mod*math.sin(self.theta)
            +  other.mod*math.sin(other.theta), self.mod*math.cos(self.theta)
                                    +  other.mod*math.cos(other.theta) )
            return Komplex(tempMod, angtheta, 0) # sum in polar coord.
        else:                                       # sum in rectangular coord
            return Komplex(self.re  +  other.re, self.im  +  other.im, 1)
```

```
    def sub(self, other, typec):                    # now for subtraction
        if typec == 0:
            tempmod = math.sqrt(self.mod*self.mod  +   other.mod*other.mod  -
            2*self.mod*other.mod*(math.cos(self.theta)*math.cos(other.theta)
             +  math.sin(self.theta)*math.sin(other.theta) ))
            y = self.mod*math.sin(self.theta) - other.mod*math.sin(other.theta)
            x = self.mod*math.cos(self.theta) - other.mod*math.cos(other.theta)
            angtheta =  math.atan2(y, x)
            return Komplex(tempmod, angtheta, 0)
        else:
            return Komplex(self.re  -  other.re, self.im  -  other.im, 1)

    def  div(self, other, typec):                   # complex division z1/z2
        if typec == 0:
            return Komplex(self.mod/other.mod, self.theta-other.theta,0)
        else:
            numre = self.re*other.re  + self.im*other.im
            deno = other.re*other.re  +  other.im*other.im
            numim = self.im*other.re  - self.re*other.im
            return Komplex(numre/deno, numim/deno, 1)

    def mult(self, other, typec):                   # complex multiplication
        if typec == 0:
            return Komplex(self.mod*other.mod, self.theta+other.theta, 0)
        else:
            return Komplex(self.re*other.re - self.im*other.im,
                   self.re*other.im  +  self.im*other.re, 1)

    def conj(self, typec):                                  # complex conj
        if typec == 0:
            self.mod = self.mod
            self.theta = - self.theta
            return(self.mod, self.theta, 0)
        else:
            return Komplex(self.re, - self.im, 1)
```

We still represent complex numbers in Cartesian or polar coordinates:

$$z = x + iy = re^{i\theta}. \tag{4.32}$$

Insofar as the complex number itself is independent of representation, we must be able to switch between the rectangular or polar representation. This is useful because certain manipulations are simpler in one representation than in the other; for example, division is easier in polar representation:

$$\frac{z_1}{z_2} = \frac{a+ib}{c+id} = \frac{ac+bd+i(bc-ad)}{c^2+d^2} = \frac{r_1 e^{i\theta_1}}{r_2 e^{i\theta_2}} = \frac{r_1}{r_2} e^{i(\theta_1-\theta_2)}. \tag{4.33}$$

Listings 4.13 and 4.14 are our implementation of an interface that permits us to use either representation when manipulating complex numbers. There are three files, **`Komplex`**, **`KomplexInterface`**, and **`KomplexTest`**, all given in the listings. Because these classes call each other, each must be in a class by itself. However, for the compiler to find all the classes, they need to be in the same directory.

Listing 4.14 **KomplexPolarTest.py** manipulates complex numbers using the interface `KomplexInterface`.

```
# KomplexPolarTest.py: tests Komplex package

from KomplexPolar import Komplex          # bring Komplex from KomplexPolar
from sys import version
if int(version[0])>2:     # raw_input deprecated in Python 3
    raw_input = input
import math

a = Komplex(1.0, 1.0, 1)                  # initialize Komplex a :cartesian
b = Komplex(1.0, 2.0, 1)                  # initialize Komplex b :cartesian

print("Cartesian: Re a =  ", a.re, " Im a =  ", a.im)   # real & im parts
print("Cartesian: Re b =  ", b.re, " Im b =  ", b.im)
c = Komplex.add(a, b, 1)                                        # a + b = c
```

```
print("Cartesian:  c =  a + b = (", c.re, ", ", c.im, ")")# re & im parts
e = b
print("Cartesian:  e =  b = (", e.re, ", ", e.im, ")")# Komplex cartesian
"Polar version, uses get and set methods"            # now polar examples
a = Komplex(math.sqrt(2.0), math.pi/4.0, 0)            #  use R and theta
b = Komplex(math.sqrt(5.), math.atan2(2.0, 1.0), 0)   # b = (R, theta, 0)
print("Polar: Re a = ", a.getRe(), ", Im a = ", a.getIm()) # cartesian re
print("Polar: Re b = ", b.getRe(), ", Im b = ", b.getIm()) # cartesian im
c = Komplex.add(a, b, 0)                                  # a + b  = c
e = c                                                      # defines e
print("Polar: Re e = ", e.getRe(), ", Im e = ", e.getIm())  # carte re,im
print("Enter any character to finish")
s = raw_input()
```

You should observe how **`KomplexInterface`** requires us to have methods for getting and setting the real and imaginary parts of **`Komplex`** objects, as well as adding, subtracting, multiplying, dividing, and conjugating complex objects. (In the comments we see the suggestion that there should also be methods for getting and setting the modulus and phase.)

The class **`Komplex`** contains the constructors for **`Komplex`** objects. This differs from our previous implementation **`Complex`** by having the additional integer variable **`type`**. If **`type = 0`**, then the complex numbers are in polar representation, else they are in Cartesian representation. So, for example, the method for arithmetic, such as the **`add`** method on line 22, is actually two different methods depending upon the value of **`type`**. In contrast, the **`get`** and **`set`** methods for real and imaginary parts are needed only for the polar representation, and so the value of **`type`** is not needed.

4.15 OOP EXAMPLE: SUPERPOSITION OF MOTIONS

The isotropy of space implies that motion in one direction is independent of motion in other directions. So, 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 to uniform motion, the distance covered due to the acceleration adds to the distance covered due to uniform velocity. Your **problem** is 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 from both the programming philosophy and the basic physics.

4.16 NEWTON'S LAWS OF MOTION (THEORY)

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:

$$\mathbf{F} = m\mathbf{a}, \quad F_i = m\frac{d^2x_i}{dt^2}, \quad (i = 1, 2, 3). \tag{4.34}$$

If the force in the x direction vanishes, $F_x = 0$, the equation of motion (4.34) has a solution corresponding to uniform motion in the x direction with a constant velocity v_{0x}:

$$x = x_0 + v_{0x}t. \tag{4.35}$$

Equation (4.35) is the *base* or *parent* object in our example. If the force in the y direction also vanishes, then there will also be uniform y motion:

$$y = y_0 + v_{0y}t. \tag{4.36}$$

We consider uniform x motion as a parent and view uniform y motion as a child.

Equation (4.34) 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

$$x = x_0 + v_{0x}t + \tfrac{1}{2}a_x t^2. \tag{4.37}$$

For projectile motion without air resistance, we usually have $a_x = 0$ and $a_y = -g = -9.8\,\mathrm{m/s^2}$:

$$y = y_0 + v_{0y}t - \tfrac{1}{2}gt^2. \tag{4.38}$$

This y motion is a child of the parent x motion.

4.17 OOP CLASS STRUCTURE (METHOD)

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

Parent class Um1D: 1-D uniform motion for given initial conditions,
Child class Um2D: uniform 2-D motion; child class of Um1D,
Child class Am2d: 2-D accelerated motion; child class of Um2D.

The *member functions* include

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

For our projectile motion, *encapsulation* is a 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 1-D motion `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 the x and y directions, it being created from the parent class `Um1D` of 1-D 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 different for 1-D and 2-D motions. In this implementation of OOP, the class `Accm2D` for accelerated motion in two dimensions inherits uniform motion in two dimensions (which in turn inherits uniform 1-D motion) and adds to it the attribute of acceleration.

4.18 PYTHON IMPLEMENTATION

Listing 4.15 **Accm2D.py** is an OOP program for accelerated motion in two dimensions.

```
# Accmd.Py: Python accelerated motion in 2D

from visual.graph import *

class UmlD:

    def __init__(self, x0, dt, vx0, ttot):            # class constructor
        self.x00 = 0                                 # initial x position
        self.delt = dt                                  # time increment
        self.vx = vx0                                       # x velocity
        self.time = ttot                                  # total tieme
        self.steps = int(ttot/self.delt)          # total int number steps

    def x(self, tt):                                # x position at time tt
        return self.x00 + tt*self.vx
    '''to be used in graphics'''

    def scenario(self,mxx,myy,mytitle,myxtitle,myytitle,xma,xmi,yma,ymi):
```

```
        graph = gdisplay(x = mxx, y = myy, width = 500, height = 200,
                title=mytitle, xtitle=myxtitle, ytitle=myytitle, xmax=xma,
                xmin=xmi, ymax=yma, ymin=ymi, foreground=color.black,
                background=color.white)

    def archive(self):                    # produce file, plot 1D x motion
        unimotion1D = gcurve(color = color.blue)
        tt = 0.0
        f = open('unimot1D.dat', 'w')           # Disk file for 1D motion
        for i in range (self.steps):
            xx = self.x(tt)
            unimotion1D.plot(pos = (tt, xx) )           # Plots x vs time
            f.write(" %f   %f\n"%(tt, xx) )             # x vs t for file
            tt   += self.delt                             # increase time
        f.closed                                        # close disk file
'''Uniform motion in 2D'''

class Um2D(Um1D):                                   # Um2D subclass of Um1D

    def __init__(self, x0, dt, vx0, ttot, y0, vy0):    # Constructor Um2D
        Um1D.__init__(self, x0, dt, vx0, ttot)        # to construct Um1D
        self.y00 = y0                             # initializes y position
        self.vy = vy0                             # initializes y velocity

    def y(self, tt):                              # produces y at time tt
        return self.y00 + tt*self.vy

    def archive(self):                          # overrides archive for 1D
        unimot2d = gcurve(color = color.magenta)
        tt = 0.0
        f = open('Um2D.dat', 'w')                   # Opens new Um2D file
        for i in range (self.steps):
            xx = self.x(tt)
            yy = self.y(tt)
            unimot2d.plot(pos = (xx, yy) )        # plots y vs x position
            f.write(" %f   %f\n"%(xx, yy) )       # writes x y in archive
            tt   += self.delt
        f.closed                                  # closes open Um2D file

'''Accelerated motion in 2D'''
class Accm2D(Um2D):                                     # Daugther of U21D
    def __init__(self, x0, dt, vx0, ttot, y0, vy0, accx, accy): # constru
        Um2D.__init__(self, x0, dt, vx0, ttot, y0, vy0)     # constructor
        self.ax = accx                              # adds acceleretions
        self.ay = accy                                   # to this class

    def xy(self, tt, i):
        self.xxac = self.x(tt) + self.ax*tt**2
        self.yyac = self.y(tt) + self.ay*tt**2
        if(i == 1):                               # if acceleration in x
          return self.xxac
        else:
          return self.yyac                        # if acceletation in y

    def archive(self):
        acmotion = gcurve(color = color.red)
        tt = 0.0
        f = open('Accm2D.dat', 'w')
        for i in range (self.steps):
            self.xxac = self.xy(tt, 1)
            self.yyac = self.xy(tt, 2)
            f.write(" %f   %f\n"%(self.xxac, self.yyac) )  # to disk file
            acmotion.plot(pos = (self.xxac, self.yyac) )    # plot motion
            tt   += self.delt
        f.closed

#comment unmd um2d or myAcc to change plot
unmd = Um1D(0.0, 0.1, 2.0, 4.0)                       # x0, dt, vx0, ttot
unmd.scenario(0, 0, 'Uniform motion in  1D ',     # for 1D uniform motion
                'time', 'x', 4.0, 0, 10.0, 0)      # tmax tmin xmax xmin
unmd.archive()                                                # archive 1D
um2d = Um2D(0.0, 0.1, 2.0, 4.0, 0.0, 5.0)    # x0, dt, vx0, ttot, y0, vy0
um2d.scenario(0, 200, 'Uniform motion in  2D ',   # for 2D uniform motion
                'x', 'y', 10.0, 0, 25.0, 0)        # xmx xmin  ymax ymin
um2d.archive()                              # archive in two dim. motion
myAcc = Accm2D(0.0, 0.1, 14.0, 4.0, 0.0, 14.0, 0.0, -9.8)
myAcc.scenario(0, 400, 'Accelerated motion ', 'x', 'y', 55, 0, 5,  -100.)
myAcc.archive()                          # archive in accelerated motion
```

Chapter Five

Monte Carlo Simulations (Nonthermal)

Unit I of this chapter addresses the problem of how computers generate numbers that appear random and how we can determine how random they are. Unit II shows how to use these random numbers to simulate physical processes. In Chapter 6, "Integration," we see show how to use these random numbers to evaluate integrals, and in Chapter 15, "Thermodynamic Simulations & Feynman Quantum Path Integration," *we investigate the use of random numbers to simulate thermal processes and the fluctuations in quantum systems.*

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Random Numbers for Monte Carlo	pdf	5.1–5.2	Monte Carlo Simulations	pdf	5.3–5.6

Applets Applet

Name	*Sections*
Radio Decay	5.5–5.62

5.1 UNIT I. DETERMINISTIC RANDOMNESS

Some people are attracted to computing because of its deterministic nature; it's nice to have a place in one's life where nothing is left to chance. Barring machine errors or undefined variables, you 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 include 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 computational physics' recognition has come about from the ability of computers to solve previously intractable problems using Monte Carlo techniques.

5.2 RANDOM SEQUENCES (THEORY)

We define a sequence $r_1, r_2, \ldots$ as *random* if there are no correlations among the numbers. Yet being random does not mean that all the numbers in the sequence are equally likely to occur. If all the numbers in a sequence are equally likely to occur, then the sequence is said to be *uniform*, and the numbers can be random as well. To illustrate, 1, 2, 3, 4, ... is uniform but probably not random. Further, it is possible to have a sequence of numbers that, in some sense, are random but have very short-range correlations among themselves, for example,

$$r_1,\ (1-r_1),\ r_2,\ (1-r_2),\ r_3,\ (1-r_3), \ldots$$

have short-range but not long-range correlations.

Mathematically, the likelihood of a number occurring is described by a distribution function $P(r)$, where $P(r)\,dr$ is the probability of finding r in the interval $[r, r+dr]$. A *uniform* distribution means that $P(r) =$ a constant. The standard random-number generator on computers generates uniform distributions between 0 and 1. In other random 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 the nature of their construction, computers are deterministic and so cannot create a random sequence. By the nature of their creation, computed random number sequences must contain correlations and in this way are not truly random. 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 are said to generate *pseudorandom numbers* (yet with our incurable laziness we won't bother saying "pseudo" all the time). While more sophisticated generators do a better job at hiding the correlations, experience shows that if you look hard enough or use these numbers long enough, you will notice correlations. A primitive alternative to generating random numbers is to read in a table of true random numbers determined by naturally random processes such as radioactive decay or to connect the computer to an experimental device that measures random events. This alternative is not good for production work but may be a useful check in times of doubt.

5.2.1 Random-Number Generation (Algorithm)

The *linear congruent* or *power residue* method is the common way of generating a pseudorandom sequence of numbers $0 \le r_i \le M-1$ over the interval $[0, M-1]$. To obtain the next random number r_{i+1}, you multiply the present random number r_i by the constant a, add another constant c, take the *modulus* by M, and then keep just the fractional part (remainder)[1]:

$$r_{i+1} \stackrel{\text{def}}{=} (a\,r_i + c) \bmod M = \text{remainder}\left(\frac{a\,r_i + c}{M}\right). \tag{5.1}$$

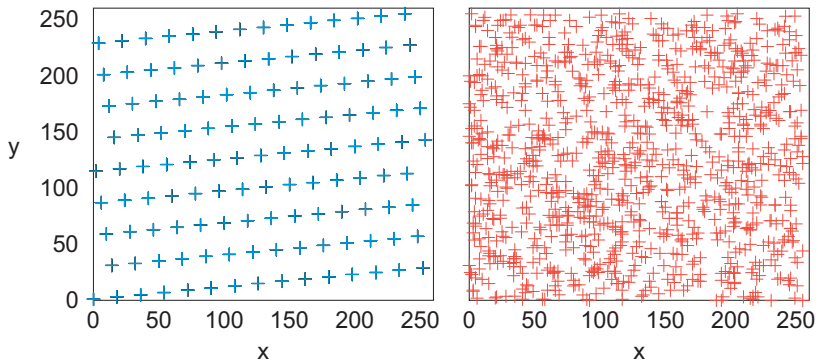
The value for r_1 (the *seed*) is frequently supplied by the user, and *mod* is a built-in function on your computer for *remaindering*. In Python the percent sign % is the modulus operator. This is essentially a bit-shift operation that ends up with the least significant part of the input number and thus counts on the randomness of round-off errors to generate a random sequence.

For example, if $c = 1, a = 4, M = 9$, and you supply $r_1 = 3$, then you obtain the sequence

$$r_1 = 3, \tag{5.2}$$

[1] 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*.

Figure 5.1 *Left:* A plot of successive random numbers $(x, y) = (r_i, r_{i+1})$ generated with a deliberately "bad" generator. *Right:* A plot generated with the built in random number generator. While the plot on the right is not proof that the distribution is random, the plot on the left is proof that the distribution is not random.



$$r_2 = (4 \times 3 + 1) \text{mod } 9 = 13 \text{ mod } 9 = \text{rem } \frac{13}{9} = 4, \tag{5.3}$$

$$r_3 = (4 \times 4 + 1) \text{mod } 9 = 17 \text{ mod } 9 = \text{rem } \frac{17}{9} = 8, \tag{5.4}$$

$$r_4 = (4 \times 8 + 1) \text{mod } 9 = 33 \text{ mod } 9 = \text{ rem } \frac{33}{9} = 6, \tag{5.5}$$

$$r_{5-10} = 7,\ 2,\ 0,\ 1,\ 5,\ 3. \tag{5.6}$$

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 a sequence of integers. If random numbers in the range $[A, B]$ are needed, you only need to **scale**:

$$x_i = A + (B - A) r_i, \quad 0 \le r_i \le 1, \ \Rightarrow \ A \le x_i \le B. \tag{5.7}$$

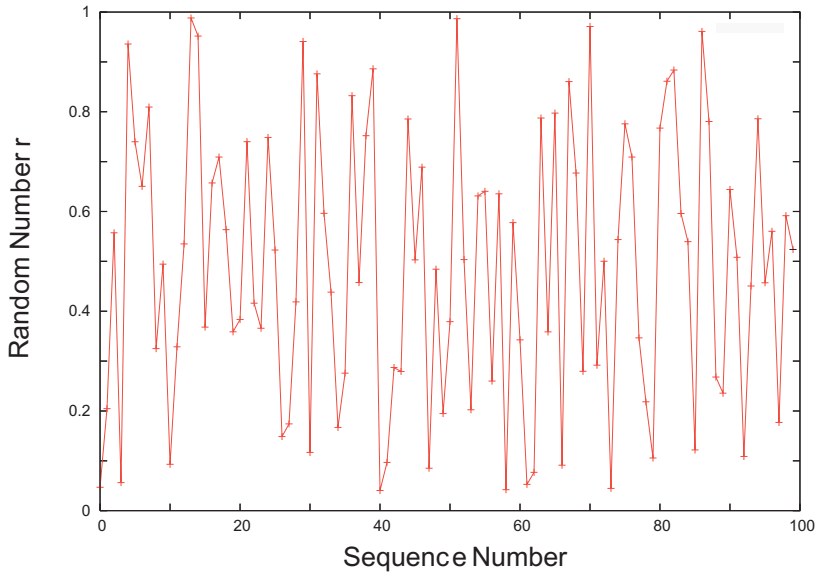
As a rule of thumb: *Before using a random-number generator in your programs, you should check its range and that it produces numbers that "look" random.*

Although not a mathematical test, you should always make a graphical display of your random numbers. Your visual cortex is quite refined at recognizing patterns and will tell you immediately if there is one in your random numbers. For instance, Figure 5.1 shows generated sequences from "good" and "bad" generators, and it is clear which is not random (although if you look hard enough at the random points, your mind may well pick out patterns there too).

The linear congruent method (5.1) produces integers in the range $[0, M-1]$ and therefore becomes completely correlated if a particular integer comes up a second time (the whole cycle then 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 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 generator 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 the cycle repeating.

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 the help

Figure 5.2 A plot of a uniform pseudorandom sequence r_i *versus* i. The points are connected to make it easier to follow the order. While this does not prove that a distribution is random, it at least shows the range of values and that there is fluctuation.



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[2]

$$M = 2^{48}, \quad c = B \text{ (base 16)} = 13 \text{ (base 8)}, \tag{5.8}$$

$$a = 5\text{DEECE66D (base 16)} = 273673163155 \text{ (base 8)}. \tag{5.9}$$

To initialize the random sequence, you need to plant a seed in it. In Fortran you call the subroutine *srand48* to plant your seed, while in Python the statement `random.seed(None)` seeds the generator with the system time (see `Walk.py` in Listing 5.1).

5.2.2 Implementation: Random Sequence

1. Write a simple program to generate random numbers using the linear congruent method (5.1).
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 $(x_i, y_i) = (r_{2i-1}, r_{2i})$, $i = 1, 2, \ldots$. (Do *not* connect the points with lines.) You may "see" correlations (Figure 5.1), which means that you should not use this sequence for serious work.
4. Make your own version of Figure 5.2; that is, plot r_i *versus* i.
5. 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.)

[2]Unless you know how to do 48-bit arithmetic and how to input numbers in different bases, it may be better to enter large numbers like $M = 112233$ and $a = 9999$.

Table 5.1 A table of a uniform, pseudo-random sequence r_i generated by Python's `random` method.

0.04689502438508175	0.20458779675039795	0.5571907470797255	0.05634336673593088
0.9360668645897467	0.7399399139194867	0.6504153029899553	0.8096333704183057
0.3251217462543319	0.49447037101884717	0.14307712613141128	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	0.14690149294881334	0.2869627609844023	0.27915054491588953
0.7854419848382436	0.502978394047627	0.688866810791863	0.08510414855949322
0.48437643825285326	0.19479360033700366	0.3791230234714642	0.9867371389465821

6. Test the linear congruent method again with reasonable constants like those in (5.8) and (5.9). Compare the scatterplot you obtain with that of the built-in random-number generator. (This, too, should be good for serious work.)

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.

5.2.3 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 with each other. This can affect a simulation that assumes random events. Therefore it is wise for you to test a random-number generator to obtain a numerical measure of its uniformity and randomness before you stake your scientific reputation on it. In fact, some tests are simple enough for you to 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, Table 5.1 presents some output from Python's **`random`** method. If you just look at these numbers, you will 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. As we have seen, a quick visual test (Figure 5.2) involves taking this same list and plotting it with r_i as ordinate and i as abscissa. 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 try to recognize some kind of pattern).
3. A simple test of uniformity evaluates the kth moment of a distribution:

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k. \tag{5.10}$$

If the numbers are distributed *uniformly*, then (5.10) is approximately the moment of the distribution function $P(x)$:

$$\frac{1}{N} \sum_{i=1}^{N} x_i^k \simeq \int_0^1 dx\, x^k P(x) \simeq \frac{1}{k+1} + O\left(\frac{1}{\sqrt{N}}\right). \tag{5.11}$$

If (5.11) holds for your generator, then you know that the distribution is uniform. If the deviation from (5.11) varies as $1/\sqrt{N}$, then you *also* know that the distribution is random.

4. Another simple test determines the near-neighbor correlation in your random sequence by taking sums of products for small k:

$$C(k) = \frac{1}{N}\sum_{i=1}^{N} x_i\, x_{i+k}, \quad (k = 1, 2, \ldots). \tag{5.12}$$

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 (5.12) can be approximated as an integral:

$$\frac{1}{N}\sum_{i=1}^{N} x_i\, x_{i+k} \simeq \int_0^1 dx \int_0^1 dy\, xy\, P(x, y) = \frac{1}{4}. \tag{5.13}$$

If (5.13) holds for your random numbers, then you know that they are uniform and independent. If the deviation from (5.13) varies as $1/\sqrt{N}$, then you *also* know that the distribution is random.

5. As we have seen, an effective test for randomness is performed 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) (Figure 5.1).
6. Test your random-number generator with (5.11) for $k = 1, 3, 7$ and $N = 100, 10,000, 100,000$. In each case print out

$$\sqrt{N}\left|\frac{1}{N}\sum_{i=1}^{N} x_i^k - \frac{1}{k+1}\right| \tag{5.14}$$

to check that it is of order 1. ▌

5.3 UNIT II. MONTE CARLO APPLICATIONS

Now that we have an idea of how to use the computer to generate pseudorandom numbers, we build some confidence that we can use these numbers to incorporate the element of chance into a simulation. We do this first by simulating a random walk and then by simulating an atom decaying spontaneously. After that, we show how knowing the statistics of random numbers leads to the best way to evaluate multidimensional integrals.

5.4 A RANDOM WALK (PROBLEM)

Consider a perfume molecule released in the front of a classroom. It collides randomly with other molecules in the air and eventually reaches your nose even though you are hidden in the last row. The **problem** is to determine how many collisions, on the average, a perfume molecule makes in traveling a distance R. You are given the fact that a molecule travels an average (*root-mean-square*) distance r_{rms} between collisions.

Listing 5.1 **Walk.py** calls the random-number generator from the random package. Note that a different seed is needed for a different sequence.

```
# Walk.py  Random walk with graph

from visual.graph import *
import random
```

```
random.seed(None)                    # Seed generator, None => system clock
jmax = 100
x    = 0.;          y = 0.                                  # Start at origin

graph1 = gdisplay(width=500, height=500, title='Random Walk', xtitle='x',
                  ytitle='y')
pts = gcurve(color = color.yellow)

for i in range(0, jmax + 1):
    pts.plot(pos = (x, y) )                                   # Plot points
    x += (random.random() - 0.5)*2.                          # -1 =< x =< 1
    y += (random.random() - 0.5)*2.                          # -1 =< y =< 1
    pts.plot(pos = (x, y))
    rate(100)
```

5.4.1 Random-Walk Simulation

There are a number of ways to simulate a random walk with (surprise, surprise) different assumptions yielding different physics. We will present the simplest approach for a 2-D 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 Chapter 13, *"Fractals & Statistical Growth*," and demonstrated by some of the corresponding applets given by links.

In our random-walk simulation (Figure 5.3) an artificial *walker* takes sequential steps with the *direction* of each step *independent* of the direction of the previous step. For our model we start at the origin and take N steps in the *xy* plane of *lengths* (not coordinates)

$$(\Delta x_1, \Delta y_1),\ (\Delta x_2, \Delta y_2),\ (\Delta x_3, \Delta y_3), \ldots,\ (\Delta x_N, \Delta y_N). \tag{5.15}$$

Even though each step may be in a different direction, the distances along each Cartesian axis just add algebraically (aren't vectors great?). Accordingly, the radial distance R from the starting point after N steps is

$$\begin{aligned} R^2 &= (\Delta x_1 + \Delta x_2 + \cdots + \Delta x_N)^2 + (\Delta y_1 + \Delta y_2 + \cdots + \Delta y_N)^2 \\ &= \Delta x_1^2 + \Delta x_2^2 + \cdots + \Delta x_N^2 + 2\Delta x_1 \Delta x_2 + 2\Delta x_1 \Delta x_3 + 2\Delta x_2 \Delta x_1 + \cdots \\ &\quad + (x \to y). \end{aligned} \tag{5.16}$$

If the walk is random, the particle is equally likely to travel in any direction at each step. If we take the average of a large number of such random steps, all the cross terms in (5.16) will vanish and we will be left with

$$\begin{aligned} R_{\text{rms}}^2 &\simeq \langle \Delta x_1^2 + \Delta x_2^2 + \cdots + \Delta x_N^2 + \Delta y_1^2 + \Delta y_2^2 + \cdots + \Delta y_N^2 \rangle \\ &= \langle \Delta x_1^2 + \Delta y_1^2 \rangle + \langle \Delta x_2^2 + \Delta y_2^2 \rangle + \cdots \\ &= N \langle r^2 \rangle = N r_{\text{rms}}^2, \\ &\Rightarrow\ R_{\text{rms}} \simeq \sqrt{N} r_{\text{rms}}, \end{aligned} \tag{5.17}$$

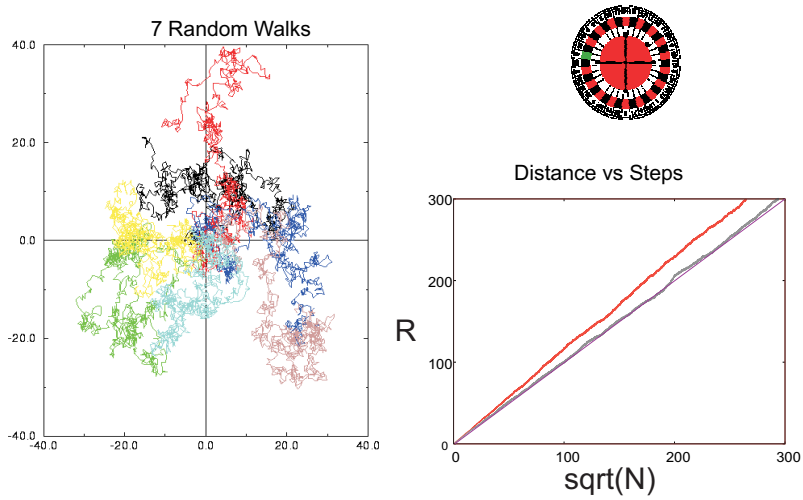
where $r_{\text{rms}} = \sqrt{\langle r^2 \rangle}$ is the *root-mean-square* step size.

To summarize, if the walk is random, then we expect that after a large number of steps the average *vector* distance from the origin will vanish:

$$\langle \vec{R} \rangle = \langle x \rangle \vec{i} + \langle y \rangle \vec{j} \simeq 0. \tag{5.18}$$

Figure 5.3 *Left:* A schematic of the N steps in a random walk simulation that end up a distance R from the origin. Notice how the Δx's for each step add algebraically. *Right:* A simulated walk in 3-D from *Walk3D.py*.

Figure 5.4 *Left:* The steps taken in seven 2-D random walk simulations. *Right:* The distance covered in two walks of N steps using different schemes for including randomness. The theoretical prediction (5.17) is the straight line.



However, (5.17) indicates that the average *scalar* distance from the origin is $\sqrt{N} r_{\text{rms}}$, where each step is of average length r_{rms}. In other words, the vector endpoint will be distributed uniformly in all quadrants, and so the displacement vector averages to zero, but the length of that vector does not. For large N values, $\sqrt{N} r_{\text{rms}} \ll N r_{\text{rms}}$ but does not vanish. In our experience, practical simulations agree with this theory, but rarely perfectly, with the level of agreement depending upon the details of how the averages are taken and how the randomness is built into each step.

5.4.2 Implementation: Random Walk

The program **`Walk.py`** in Listing 5.1 is a sample random-walk simulation. It's key element is random values for the x and y components of each step,

```
x += (random.random() - 0.5)*2.          # -1 =< x =< 1
y += (random.random() - 0.5)*2.          # -1 =< y =< 1
```

Here we omit the scaling factor that normalizes each step to length 1. When using your computer to simulate a random walk, you should expect to obtain (5.17) only 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 (Figure 5.4 right).

Start at the origin and take a 2-D 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 each 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 2-D random walks, each of 1000 steps. Using evidence from your simulations, comment on whether these look like what you would expect of a random walk.
3. If you have your walker taking N steps in a single trial, then conduct a total number $K \simeq \sqrt{N}$ of 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:
$$\langle\, R^2(N)\,\rangle = \frac{1}{K}\sum_{k=1}^{K} R^2_{(k)}(N).$$
5. Repeat the preceding and following analysis for a 3-D walk as well.
6. Check the validity of the assumptions made in deriving the theoretical result (5.17) by checking how well
$$\frac{\langle \Delta x_i \Delta x_{j\neq i}\rangle}{R^2} \simeq \frac{\langle \Delta x_i \Delta y_j\rangle}{R^2} \simeq 0.$$
Do your checking for both a single (long) run and for the average over trials.
7. Plot the root-mean-square distance $R_{\rm rms} = \sqrt{\langle R^2(N)\rangle}$ 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 two or three places of accuracy should be expected on the average.

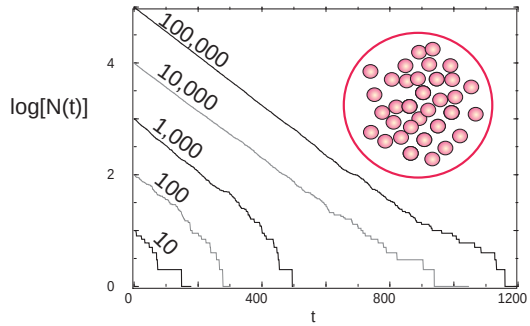
5.5 RADIOACTIVE DECAY (PROBLEM)

Your **problem** is to simulate how a small number N of radioactive particles decay.[3] In particular, you are to determine when radioactive decay looks like exponential decay and when it looks *stochastic* (containing elements of chance). Because the exponential decay law is a large-number approximation to a natural process that always ends with small numbers, our simulation should be closer to nature than is the exponential decay law (Figure 5.5). In fact, if you "listen" to the output of the decay simulation code, what you will hear sounds very much like a Geiger counter, a convincing demonstration of the realism of the simulation.

Spontaneous decay is a natural process in which a particle, with no external stimulation, decays into other particles. Even though the probability of decay of any one particle in any time interval is constant, just when it decays is a random event. Because the exact moment when any one particle decays is random, it does not matter how long the particle has been around or whether some other particles have decayed. In other words, the probability $\mathcal{P}$ of any one particle decaying per unit time interval is a constant, and when that particle decays, it is gone forever. Of course, as the total number of particles decreases with time, so will the number of decays, but the probability of any one particle decaying in some time interval is always the same constant as long as that particle exists.

[3] Spontaneous decay is also discussed in Chapter 8, "Solving Systems of Equations with Matrices; Data Fitting," where we fit an exponential function to a decay spectrum.

Figure 5.5 The circle shows a sample of N nuclei, each of which has the same probability $1/\lambda$ of decaying per unit time. The semilog plots show the results from several simulations of the nucleus decaying. Notice how the decay appears exponential (like a straight line) when the number of nuclei is large, but stochastic when there are 100 or fewer nuclei.



5.5.1 Discrete Decay (Model)

Imagine having a sample of $N(t)$ radioactive nuclei at time t (Figure 5.5 inset). Let ΔN be the number of particles that decay in some small time interval Δt. We convert the statement "the probability $\mathcal{P}$ of any one particle decaying per unit time is a constant" into the equation

$$\mathcal{P} = \frac{\Delta N(t)/N(t)}{\Delta t} = -\lambda, \tag{5.19}$$

$$\Rightarrow \quad \frac{\Delta N(t)}{\Delta t} = -\lambda N(t), \tag{5.20}$$

where the constant λ is called the *decay rate*. Of course the real decay rate or *activity* is $\Delta N(t)/\Delta t$, which varies with time. In fact, because the total activity is proportional to the total number of particles still present, it too is stochastic with an exponential-like decay in time. [Actually, because the number of decays $\Delta N(t)$ is proportional to the difference in random numbers, its stochastic nature becomes evident before that of $N(t)$.]

Equation (5.20) is a *finite-difference equation* in terms of the experimental measurables $N(t)$, $\Delta N(t)$, and Δt. Although it cannot be integrated the way a differential equation can, it can be solved numerically when we include the fact that the decay process is random. Because the process is random, we cannot predict a single value for $\Delta N(t)$, although we can predict the average number of decays when observations are made of many identical systems of N decaying particles.

5.5.2 Continuous Decay (Model)

When the number of particles $N \to \infty$ and the observation time interval $\Delta t \to 0$, an approximate form of the radioactive decay law (5.20) results:

$$\frac{\Delta N(t)}{\Delta t} \longrightarrow \frac{dN(t)}{dt} = -\lambda N(t). \tag{5.21}$$

This can be integrated to obtain the time dependence of the total number of particles and of the total activity:

$$N(t) = N(0)e^{-\lambda t} = N(0)e^{-t/\tau}, \tag{5.22}$$

$$\frac{dN(t)}{dt} = -\lambda N(0)e^{-\lambda t} = \frac{dN}{dt}e^{-\lambda t}(0). \tag{5.23}$$

We see that in this limit we obtain exponential decay, which leads to the identification of the decay rate λ with the inverse lifetime:

$$\lambda = \frac{1}{\tau}. \tag{5.24}$$

So we see from its derivation that exponential decay is a good description of nature for a large number of particles where $\Delta N/N \simeq 0$. The basic law of nature (5.19) is always valid, but as we will see in the simulation, exponential decay (5.23) becomes less and less accurate as the number of particles becomes smaller and smaller.

5.5.3 Decay Simulation with Geiger Counter Sound

A program for simulating radioactive decay is surprisingly simple but not without its subtleties. We increase time in discrete steps of Δt, and for each time interval we count the number of nuclei that have decayed during that Δt. The simulation quits when there are no nuclei left to decay. Such being the case, we have an outer loop over the time steps Δt and an inner loop over the remaining nuclei for each time step. The pseudocode is simple (as is the code): (DecaySound.py)

```
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
```

When we pick a value for the decay rate $\lambda = 1/\tau$ to use in our simulation, we are setting the scale for times. If the actual decay rate is $\lambda = 0.3 \times 10^6\,\mathrm{s}^{-1}$ and if we decide to measure times in units of 10^{-6} s, then we will choose random numbers $0 \le r_i \le 1$, which leads to λ values lying someplace near the middle of the range (e.g., $\lambda \simeq 0.3$). Alternatively, we can use a value of $\lambda = 0.3 \times 10^6\,\mathrm{s}^{-1}$ in our simulation and then scale the random numbers to the range $0 \le r_i \le 10^6$. However, unless you plan to compare your simulation to experimental data, you do not have to worry about the scale for time but instead should focus on the physics behind the slopes and relative magnitudes of the graphs.

Listing 5.2 **DecaySound.py** simulates spontaneous decay in which a decay occurs if a random number is smaller than the decay parameter.

```
# DecaySound.py spontaneous decay simulation

from visual.graph import *
import random
import winsound
lambda1 = 0.001                                                    # Decay constant
max = 200.;   time_max = 500;   seed = 68111                              # Params
number = nloop = max                                               # Initial value
# random.seed(seed)                                          # Seed number generator
```

```
graph1 = gdisplay(width=1000, height=1000, title ='Spontaneous Decay',xtitle='Time',
                  ytitle = 'Number left',xmax=500,xmin=0,ymax=300,ymin=0)
decayfunc = gcurve(color = color.green)

for time in arange(0, time_max + 1):

    # Time loop
    for atom in arange(1, number + 1 ):

        # Decay loop
        decay = random.random()
        if (decay < lambda1):
            nloop = nloop - 1     # A decay
            winsound.Beep(600, 100)
    number = nloop
    decayfunc.plot( pos = (time, number) )
    rate(30)
```

Decay.py is our sample simulation of spontaneous decay. An extension of this program, **DecaySound.py**, adds a beep each time an atom decays (unfortunately this works only with Windows). When we listen to the simulation, it sounds like a Geiger counter, with its randomness and its slowing down with time. This provides some rather convincing evidence of the realism of the simulation.

5.6 DECAY IMPLEMENTATION AND VISUALIZATION

Write a program to simulate radioactive decay using the simple program in Listing 5.2 as a guide. You should obtain results like those in Figure 5.5.

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 Δt (generation number).
2. Check that you obtain what looks like exponential decay when you start with large values for $N(0)$, but that the decay displays its stochastic nature for small $N(0)$ [large $N(0)$ values are also stochastic; they just don't look like it].
3. Create two plots, one showing that the slopes of $N(t)$ *versus* t are *independent* of $N(0)$ and another showing that the slopes are proportional to λ.
4. Create a plot showing that within the expected statistical 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 can lead to exponential decay.
6. How does your simulation show that the decay is exponential-like and not a power law such as $N = \beta t^{-\alpha}$?

Chapter Six

Integration

In this chapter we discuss numerical integration, a basic tool of scientific computation. We derive the Simpson and trapezoid rules but just sketch the basis of Gaussian quadrature, which, though our standard workhorse, is long in derivation. We do discuss Gaussian quadrature in its various forms and indicate how to transform the Gauss points to a wide range of intervals. We end the chapter with a discussion of Monte Carlo integration, which is fundamentally different from other integration techniques.

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*
Numerical Integration	pdf	6.1–6.3

Applets Applet

Name	*Sections*
Area with MC	6.5

6.1 INTEGRATING A SPECTRUM (PROBLEM)

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 for an arbitrary decay rate

$$N(1) = \int_0^1 \frac{dN(t)}{dt}\, dt. \tag{6.1}$$

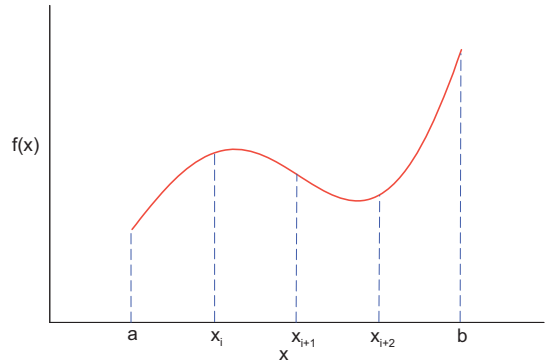
6.2 QUADRATURE AS BOX COUNTING (MATH)

The integration of a function may require some cleverness to do analytically but is relatively straightforward on a computer. A traditional way to perform 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 (Figure 6.1):

$$\int_a^b f(x)\, dx = \lim_{h\to 0}\left[h \sum_{i=1}^{(b-a)/h} f(x_i)\right]. \tag{6.2}$$

Figure 6.1 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 h.



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:

$$\int_a^b f(x)\,dx \simeq \sum_{i=1}^{N} f(x_i)w_i, \tag{6.3}$$

which is similar to the Riemann definition (6.2) except that there is no limit to an infinitesimal box size. Equation (6.3) 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 (6.3) gives the exact integral only when $N \to \infty$, it may be exact for finite N if the integrand is a polynomial. The different integration algorithms amount to different ways of choosing the points and weights. Generally, the precision increases as N gets larger, with 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 switch from one method to another and change the methods for different intervals until they find ones that work well for each interval.

In general you should not attempt a numerical integration of an integrand that contains a singularity without first removing the singularity by hand.[1] 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 an integration point is not placed or by a change of variable:

$$\int_{-1}^{1} |x| f(x)\,dx = \int_{-1}^{0} f(-x)\,dx + \int_{0}^{1} f(x)\,dx, \tag{6.4}$$

$$\int_0^1 x^{1/3}\,dx = \int_0^1 3y^3\,dy, \quad (y = x^{1/3}), \tag{6.5}$$

$$\int_0^1 \frac{f(x)\,dx}{\sqrt{1-x^2}} = 2\int_0^1 \frac{f(1-y^2)\,dy}{\sqrt{2-y^2}}, \quad (y^2 = 1-x). \tag{6.6}$$

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.

[1] In Chapter 20, "Integral Equations in Quantum Mechanics," we show how to remove such a singularity even when the integrand is unknown.

Elementary Weights for Uniform-Step Integration Rules

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

Listing 6.1 **TrapMethods.py** integrates the arbitrary function $f(y)$ via the trapezoid rule. Note how the step size h depends on the interval and how the weights at the ends and middle differ.

```
# TrapMethods        Trapezoid integration of function

from numpy import *
from sys import version

if int(version[0])>2:  raw_input=input  # raw_input deprecated in Python3
A = 0.0;  B = 3.0; N = 4

def f(y):                                       # function being integrated
    print(" y  f(y)  = ", y, y*y)
    return y*y

def wTrap(i, h):                                        # determines weight
    if ( (i == 1) or (i == N) ):   wLocal = h/2.0
    else: wLocal = h
    return wLocal

h = (B - A)/(N - 1)
suma = 0.0

for i in range(1, N + 1):
    t = A + (i - 1)*h
    w = wTrap(i, h)
    suma  = suma + w * f(t)

print(' Total sum = ', suma)
print("Enter and return any character to quit")
s = raw_input()
```

6.2.1 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:

$$h = \frac{b-a}{N-1}, \quad x_i = a + (i-1)h, \;\; i = 1, N, \tag{6.7}$$

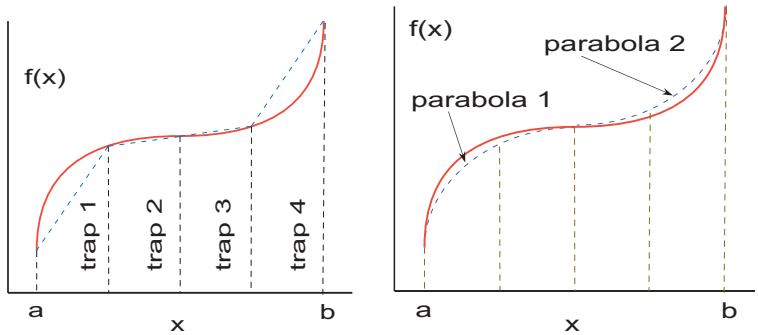
where we start our counting at $i = 1$. The trapezoid rule takes each integration interval i and constructs a trapezoid of width h in it (Figure 6.2). This approximates $f(x)$ by a straight line in each interval i and uses the average height $(f_i + f_{i+1})/2$ as the value for f. The area of each such trapezoid is

$$\int_{x_i}^{x_i+h} f(x)\,dx \simeq \frac{h(f_i + f_{i+1})}{2} = \frac{1}{2}hf_i + \frac{1}{2}hf_{i+1}. \tag{6.8}$$

In terms of our standard integration formula (6.3), the "rule" in (6.8) is for $N = 2$ points with weights $w_i \equiv \frac{1}{2}$ (Table 6.2.1).

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

Figure 6.2 Different shapes used to approximate the areas under the curve. *Left:* Straight-line sections used for the trapezoid rule. *Right:* Two parabolas used in Simpson's rule.



from each subinterval:

$$\int_a^b f(x)\,dx \simeq \frac{h}{2}f_1 + hf_2 + hf_3 + \cdots + hf_{N-1} + \frac{h}{2}f_N. \tag{6.9}$$

You will notice that because the internal points are counted twice (as the end of one interval and as the beginning of the next), they have weights of $h/2 + h/2 = h$, whereas the endpoints are counted just once and on that account have weights of only $h/2$. In terms of our standard integration rule (6.33), we have

$$w_i = \left\{\frac{h}{2}, h, \ldots, h, \frac{h}{2}\right\} \qquad \text{(trapezoid rule).} \tag{6.10}$$

In Listing 6.1 we provide a simple implementation of the trapezoid rule.

6.2.2 Algorithm: Simpson's Rule

For each interval, Simpson's rule approximates the integrand $f(x)$ by a parabola (Figure 6.2 right):

$$f(x) \simeq \alpha x^2 + \beta x + \gamma, \tag{6.11}$$

with all intervals equally spaced. The area under the parabola for each interval is

$$\int_{x_i}^{x_i+h} (\alpha x^2 + \beta x + \gamma)\,dx = \frac{\alpha x^3}{3} + \frac{\beta x^2}{2} + \gamma x\Big|_{x_i}^{x_i+h}. \tag{6.12}$$

In order to relate the parameters α, β, and γ to the function, we consider an interval from -1 to $+1$, in which case

$$\int_{-1}^{1} (\alpha x^2 + \beta x + \gamma)\,dx = \frac{2\alpha}{3} + 2\gamma. \tag{6.13}$$

But we notice that

$$f(-1) = \alpha - \beta + \gamma, \;\; f(0) = \gamma, \;\; f(1) = \alpha + \beta + \gamma, \tag{6.14}$$

$$\Rightarrow \alpha = \frac{f(1) + f(-1)}{2} - f(0), \beta = \frac{f(1) - f(-1)}{2}, \gamma = f(0). \tag{6.15}$$

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

three points:

$$\int_{-1}^{1} (\alpha x^2 + \beta x + \gamma)\, dx = \frac{f(-1)}{3} + \frac{4f(0)}{3} + \frac{f(1)}{3}. \tag{6.16}$$

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 in the middle (Table 6.2.1):

$$\begin{aligned}\int_{x_i-h}^{x_i+h} f(x)\, dx &= \int_{x_i}^{x_i+h} f(x)\, dx + \int_{x_i-h}^{x_i} f(x)\, dx \\ &\simeq \frac{h}{3} f_{i-1} + \frac{4h}{3} f_i + \frac{h}{3} f_{i+1}.\end{aligned} \tag{6.17}$$

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 that 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:

$$\int_a^b f(x)dx \simeq \frac{h}{3} f_1 + \frac{4h}{3} f_2 + \frac{2h}{3} f_3 + \frac{4h}{3} f_4 + \cdots + \frac{4h}{3} f_{N-1} + \frac{h}{3} f_N. \tag{6.18}$$

In terms of our standard integration rule (6.3), we have

$$w_i = \left\{ \frac{h}{3}, \frac{4h}{3}, \frac{2h}{3}, \frac{4h}{3}, \ldots, \frac{4h}{3}, \frac{h}{3} \right\} \quad \text{(Simpson's rule)}. \tag{6.19}$$

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

$$\sum_{i=1}^{N} w_i = (N-1)h. \tag{6.20}$$

Remember, the number of points N must be odd for Simpson's rule.

6.2.3 Integration Error (Analytic Assessment)

In general, you should choose an integration rule that gives an accurate answer using the least number of integration points. We obtain a crude estimate of the *approximation* or *algorithmic error* $\mathcal{E}$ and the relative error ϵ 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 rules this yields

$$\mathcal{E}_t = O\left(\frac{[b-a]^3}{N^2}\right) f^{(2)}, \quad \mathcal{E}_s = O\left(\frac{[b-a]^5}{N^4}\right) f^{(4)}, \quad \epsilon_{t,s} = \frac{\mathcal{E}_{t,s}}{f}. \tag{6.21}$$

We see that the third-derivative term in Simpson's rule cancels (much like the central-difference method does in differentiation). Equations (6.21) are illuminating in showing how increasing the sophistication of an integration rule leads to an error that decreases with a higher inverse power of N yet is also proportional to higher derivatives of f. Consequently, for small intervals and functions $f(x)$ with well-behaved derivatives, Simpson's rule should converge more rapidly than the trapezoid rule.

To model the error in integration, we assume that after N steps the *relative* round-off error is random and of the form

$$\epsilon_{\text{ro}} \simeq \sqrt{N}\epsilon_m, \tag{6.22}$$

where ϵ_m is the machine precision, $\epsilon \sim 10^{-7}$ for single precision and $\epsilon \sim 10^{-15}$ for double precision (the standard for science). 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 round-off errors:

$$\epsilon_{\text{tot}} \simeq \epsilon_{\text{ro}} + \epsilon_{\text{approx}}. \tag{6.23}$$

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

$$\epsilon_{\text{ro}} = \epsilon_{\text{approx}} = \frac{\mathcal{E}_{\text{trap,simp}}}{f}. \tag{6.24}$$

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

$$\frac{f^{(n)}}{f} \simeq 1, \quad b - a = 1 \quad \Rightarrow \quad h = \frac{1}{N}. \tag{6.25}$$

The estimate (6.24), when applied to the **trapezoid rule**, yields

$$\sqrt{N}\epsilon_m \simeq \frac{f^{(2)}(b-a)^3}{fN^2} = \frac{1}{N^2}, \tag{6.26}$$

$$\Rightarrow \quad N \simeq \frac{1}{(\epsilon_m)^{2/5}} = \left(\frac{1}{10^{-15}}\right)^{2/5} = 10^6, \tag{6.27}$$

$$\Rightarrow \epsilon_{\text{ro}} \simeq \sqrt{N}\epsilon_m = 10^{-12}. \tag{6.28}$$

The estimate (6.24), when applied to **Simpson's rule**, yields

$$\sqrt{N}\epsilon_m = \frac{f^{(4)}(b-a)^5}{fN^4} = \frac{1}{N^4}, \tag{6.29}$$

$$\Rightarrow N = \frac{1}{(\epsilon_m)^{2/9}} = \left(\frac{1}{10^{-15}}\right)^{2/9} = 2154, \tag{6.30}$$

$$\Rightarrow \epsilon_{\text{ro}} \simeq \sqrt{N}\epsilon_m = 5 \times 10^{-14}. \tag{6.31}$$

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 achieved by letting $N \to \infty$ but with a relatively small $N \leq 1000$.

6.2.4 Algorithm: Gaussian Quadrature

It is often useful to rewrite the basic integration formula (6.3) such that we separate a weighting function $W(x)$ from the integrand:

$$\int_a^b f(x)\,dx \equiv \int_a^b W(x)g(x)\,dx \simeq \sum_{i=1}^{N} w_i g(x_i). \tag{6.32}$$

In the Gaussian quadrature approach to integration, the N points and weights are chosen to make the approximation error vanish if $g(x)$ were a $(2N-1)$-degree polynomial. To obtain this incredible optimization, the points x_i end up having a specific distribution over $[a,b]$. In general, if $g(x)$ is smooth or can be made smooth by factoring out some $W(x)$ (Table 6.2.4), Gaussian algorithms will produce higher accuracy than the trapezoid and Simpson rules for the same number of points. Sometimes the integrand 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 "smart" integration subroutines decide for themselves how many intervals to use and what rule to use in each.

All the rules indicated in Table 6.2.4 are Gaussian with the general form (6.32). 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 the values of the points and weights change as the number of points N changes. Although we will leave the derivation of the Gaussian points and weights to the references on numerical methods, 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 [A&S 72], or can be computed. The *gauss* subroutines we provide also scale the points to span specified regions. As a check that your points are correct, you may want to compare them to the four-point set in Table 6.1.

Mapping Integration Points

Our standard convention (6.3) for the general interval $[a,b]$ is

$$\int_a^b f(x)\,dx \simeq \sum_{i=1}^{N} f(x_i)w_i. \tag{6.33}$$

With Gaussian points and weights, the y interval $-1 < y_i \le 1$ must be *mapped* onto the x interval $a \le x \le b$. 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. $[-1,1] \to [a,b]$ uniformly, $(a+b)/2 =$ **midpoint:**

$$x_i = \frac{b+a}{2} + \frac{b-a}{2}y_i, \quad w_i = \frac{b-a}{2}w_i^{'}, \tag{6.34}$$

$$\Rightarrow \quad \int_a^b f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f[x(y)]\,dy. \tag{6.35}$$

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		

Table 6.1 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

2. $[0 \to \infty]$, $a =$ **midpoint:**

$$x_i = a\frac{1+y_i}{1-y_i}, \quad w_i = \frac{2a}{(1-y_i)^2}w_i^{'}. \tag{6.36}$$

3. $[-\infty \to \infty]$, **scale set by** a**:**

$$x_i = a\frac{y_i}{1-y_i^2}, \quad w_i = \frac{a(1+y_i^2)}{(1-y_i^2)^2}w_i^{'}. \tag{6.37}$$

4. $[a \to \infty]$, $a + 2b =$ **midpoint:**

$$x_i = \frac{a+2b+ay_i}{1-y_i}, \quad w_i = \frac{2(b+a)}{(1-y_i)^2}w_i^{'}. \tag{6.38}$$

5. $[0 \to b]$, $ab/(b+a) =$ **midpoint:**

$$x_i = \frac{ab(1+y_i)}{b+a-(b-a)y_i}, \quad w_i = \frac{2ab^2}{(b+a-(b-a)y_i)^2}w_i^{'}. \tag{6.39}$$

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 gets larger but always remains finite.

6.2.5 Implementation and Error Assessment

1. Write a double-precision program to integrate an arbitrary function numerically using the trapezoid rule, the Simpson rule, and Gaussian quadrature. For our assumed **problem** there is an analytic answer:

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

2. Compute the relative error $\epsilon = |(\text{numerical-exact})/\text{exact}|$ in each case. Present your data in the tabular form

N	ϵ_T	ϵ_S	ϵ_G
2	...	...	...
10	...	...	...

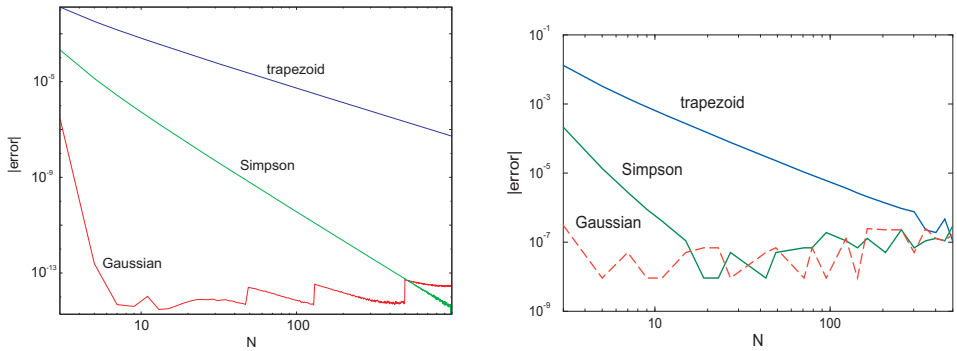
with spaces or tabs separating the fields. Try N values of 2, 10, 20, 40, 80, 160, ... (*Hint*: Even numbers may not be the assumption of every rule.)

3. Make a log-log plot of relative error *versus* N (Figure 6.3). You should observe that

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

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 of decimal places of precision in your calculation.

Figure 6.3 Log-log plots of the error in the integration of exponential decay using the trapezoid rule, Simpson's rule, and Gaussian quadrature *versus* the number of integration points N. Approximately 15 decimal places of precision are attainable with double precision (*left*), and 7 places with single precision (*right*). The algorithms are seen to stop converging when round-off error (the fluctuating and increasing part near the bottom) starts to dominate.



4. Use your plot or table to estimate the power-law dependence of the error ϵ 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 round-off error regimes. (Note that it may be hard to reach the round-off error regime for the trapezoid rule because the approximation error is so large.)

In Listing 6.2 we give a sample program that performs an integration with Gaussian points. The method **gauss** generates the points and weights and may be useful in other applications as well.

Listing 6.2 **IntegGauss.py** integrates the function $f(x)$ via Gaussian quadrature. The points and weights are generated in the method `gauss`, which remains fixed for all applications. Note that 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 Gaussian points onto arbitrary intervals (they are generated for $-1 \leq x \leq 1$).

```
# IntegGauss.py: Gaussian quadrature generator of pts & wts

from numpy import *
from sys import version
if int(version[0])>2:                    # raw_input deprecated in Python 3
    raw_input=input
max_in = 11                               # Numb intervals
vmin = 0.; vmax = 1.                      # Int ranges
ME = 2.718281828459045235 4E0             # Euler's const
w = zeros( (2001), float)
x = zeros( (2001), float)

def f(x):                                 # The integrand
    return (exp( - x) )

def gauss(npts, job, a, b, x, w):
    m  = i = j = t = t1 = pp = p1 = p2 = p3 = 0.
    eps = 3.E-14                     # Accuracy: ******ADJUST THIS*******!
    m = (npts + 1)/2
    for i in range(1, m + 1):
        t = cos(math.pi*(float(i) - 0.25)/(float(npts) + 0.5) )
        t1 = 1
        while( (abs(t - t1) ) >= eps):
            p1 = 1. ;  p2 = 0.
            for j in range(1, npts + 1):
                p3 = p2;   p2 = p1
                p1 = ((2.*float(j)-1)*t*p2 - (float(j)-1.)*p3)/(float(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]
    if (job == 0):
        for i in range(0, npts):
```

```
            x[i] = x[i]*(b - a)/2. + (b + a)/2.
            w[i] = w[i]*(b - a)/2.
    if (job == 1):
        for i in range(0, npts):
            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 in range(0, npts):
            xi = x[i]
            x[i] = (b*xi +  b + a + a) / (1. - xi)
            w[i] = w[i]*2.*(a + b)/( (1. - xi)*(1. - xi) )

def gaussint (no, min, max):
    quadra = 0.
    gauss (no, 0, min, max, x, w)                      # Returns pts & wts
    for n in xrange(0, no):
        quadra   += f(x[n]) * w[n]                    # Calculate integral
    return (quadra)

for i in range(3, max_in + 1, 2):
    result = gaussint(i, vmin, vmax)
    print (" i ", i, " err ", abs(result - 1 + 1/ME))
print ("Enter and return any character to quit")
s = raw_input()
```

6.3 EXPERIMENTATION

Try two integrals for which the answers are less obvious:

$$F_1 = \int_0^{2\pi} \sin(100x)\,dx, \quad F_2 = \int_0^{2\pi} \sin^x(100x)\,dx. \tag{6.40}$$

Explain why the computer may have trouble with these integrals. ∎

6.4 HIGHER-ORDER RULES (ALGORITHM)

As in numerical differentiation, we can use the known functional dependence of the error on interval size h to reduce the integration error. For simple rules like the trapezoid and Simpson rules, we have the analytic estimates (6.24), 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 intervals h and $h/2$, respectively, and we know that the integrals have expansions with a leading error term proportional to h^2,

$$A(h) \simeq \int_a^b f(x)\,dx + \alpha h^2 + \beta h^4 + \cdots, \tag{6.41}$$

$$A\left(\frac{h}{2}\right) \simeq \int_a^b f(x)\,dx + \frac{\alpha h^2}{4} + \frac{\beta h^4}{16} + \cdots. \tag{6.42}$$

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

$$\frac{4}{3}A\left(\frac{h}{2}\right) - \frac{1}{3}A(h) \simeq \int_a^b f(x)\,dx - \frac{\beta h^4}{4} + \cdots. \tag{6.43}$$

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 Table 6.2.1 we gave the weights for several equal-interval rules. Whereas the Simpson rule used two intervals, the three-eighths rule uses three, and the Milne[2] rule four. (These

[2]There is, not coincidentally, a Milne Computer Center at Oregon State University, although there no longer is a central

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 are 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 $b - a$):

$$\sum_{i=1}^{N} w_i = h \times N_{\text{intervals}} = b - a. \tag{6.44}$$

6.5 PROBLEM: MONTE CARLO INTEGRATION BY STONE THROWING

Imagine yourself as a farmer walking to your furthermost field to add algae-eating fish to a pond having an algae explosion. You get there only to read the instructions and discover that you need to know the area of the pond in order to determine the correct number of the fish to add. Your **problem** is to measure the area of this irregularly shaped pond with just the materials at hand [G,T&C 06].

It is hard to believe that Monte Carlo techniques can be used to evaluate integrals. After all, we do not want to gamble on the values! While it is true that other methods are preferable for single and double integrals, Monte Carlo techniques are best when the dimensionality of integrations gets large! For our pond problem, we will use a *sampling* technique (Figure 6.4):

1. Walk off a box that completely encloses the pond and remove any pebbles lying on the ground 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_{box}.
3. Grab a bunch of pebbles, count their number, and then throw them up in the air in random directions.
4. Count the number of splashes in the pond N_{pond} and the number of pebbles lying on the ground within your box N_{box}.
5. Applet 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_{pond}. You determine that area from the simple ratio

$$\frac{N_{\text{pond}}}{N_{\text{pond}} + N_{\text{box}}} = \frac{A_{\text{pond}}}{A_{\text{box}}} \quad \Rightarrow \quad A_{\text{pond}} = \frac{N_{\text{pond}}}{N_{\text{pond}} + N_{\text{box}}} A_{\text{box}}. \tag{6.45}$$

6.5.1 Stone Throwing Implementation

Use sampling (Figure 6.4) to perform a 2-D integration and thereby determine π:

1. Imagine a circular pond enclosed in a square of side $2\,(r = 1)$.
2. We know the analytic area of a circle $\oint dA = \pi$.
3. Generate a sequence of random numbers $-1 \le r_i \le +1$.
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 $N_{\text{pond}} = N_{\text{pond}} + 1$; otherwise let $N_{\text{box}} = N_{\text{box}} + 1$.
6. Use (6.45) to calculate the area, and in this way π.

---

computer there.

Figure 6.4 Throwing stones into a pond as a technique for measuring its area. The ratio of "hits" to total number of stones thrown equals the ratio of the area of the pond to that of the box.

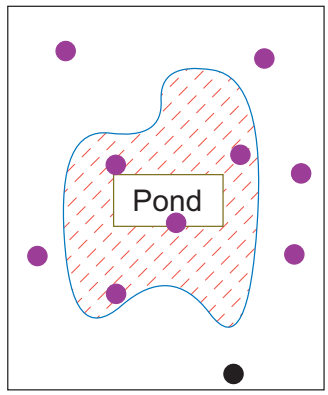
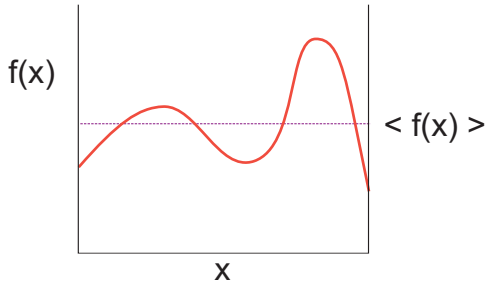


Figure 6.5 The area under the curve $f(x)$ is the same as that under the horizontal line whose height $y = \langle f \rangle$.



7. Increase N until you get π to three significant figures (we don't ask much — that's only slide-rule accuracy).

6.5.2 Integration by Mean Value (Math)

The standard Monte Carlo technique for integration is based on the *mean value theorem* (presumably familiar from elementary calculus):

$$I = \int_a^b dx\, f(x) = (b-a)\langle f \rangle. \tag{6.46}$$

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 mean value of the function over that interval $\langle f \rangle$ (Figure 6.5). The integration algorithm uses Monte Carlo techniques to evaluate the mean in (6.46). With a sequence $a \le x_i \le b$ of N uniform random numbers, we want to determine the *sample mean* by *sampling* the function $f(x)$ at these points:

$$\langle f \rangle \simeq \frac{1}{N} \sum_{i=1}^{N} f(x_i). \tag{6.47}$$

This gives us the very simple integration rule:

$$\int_a^b dx\, f(x) \simeq (b-a)\frac{1}{N} \sum_{i=1}^{N} f(x_i) = (b-a)\langle f \rangle. \tag{6.48}$$

Equation (6.48) looks much like our standard algorithm for integration (6.3) with the "points" x_i chosen randomly and with uniform weights $w_i = (b-a)/N$. Because no attempt has been made to obtain 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 $N \to \infty$ or if we keep the number of samples finite and take the average of infinitely many runs, the laws of statistics assure us that (6.48) will approach the correct answer, at least if there were no round-off errors.

For readers who are 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 σ_I. If σ_f is the standard deviation of the integrand f in the sampling, then for normal distributions we have

$$\sigma_I \simeq \frac{1}{\sqrt{N}}\sigma_f. \tag{6.49}$$

So for large N, the error in the value obtained for the integral decreases as $1/\sqrt{N}$.

6.6 HIGH-DIMENSIONAL INTEGRATION (PROBLEM)

Let's 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 atomic wave functions over the three coordinates of each of 12 electrons. This amounts to a $3 \times 12 = 36$-D integral. If we use 64 points for each integration, this requires about $64^{36} \simeq 10^{65}$ evaluations of the integrand. If the computer were fast and could evaluate the integrand a million times per second, this would take about 10^{59} s, which is significantly longer than the age of the universe ($\sim 10^{17}$ s).

Your **problem** is to find a way to perform multidimensional integrations so that you are still alive to savor the answers. Specifically, evaluate the 10-D integral

$$I = \int_0^1 dx_1 \int_0^1 dx_2 \cdots \int_0^1 dx_{10} \left(x_1 + x_2 + \cdots + x_{10}\right)^2. \tag{6.50}$$

Check your numerical answer against the analytic one, $\frac{155}{6}$.

6.6.1 Multidimensional Monte Carlo

It is easy to generalize mean value integration to many dimensions by picking random points in a multidimensional space. For example,

$$\int_a^b dx \int_c^d dy\, f(x,y) \simeq (b-a)(d-c)\frac{1}{N}\sum_i^N f(\mathbf{x}_i) = (b-a)(d-c)\langle f\rangle. \tag{6.51}$$

6.6.2 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 1-D integrals using a rule such as Simpson's, 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 will find that at a value of $D \simeq 3$–4 the error in Monte Carlo integration is similar to that of conventional schemes. For larger values of D, the Monte Carlo method is always more accurate!

6.6.3 Implementation: 10-D Monte Carlo Integration

Use a built-in random-number generator to perform the 10-D Monte Carlo integration in (6.50).

1. Conduct 16 trials and take the average as your answer.
2. Try sample sizes of $N = 2, 4, 8, \ldots, 8192$.
3. Plot the error *versus* $1/\sqrt{N}$ and see if linear behavior occurs.

6.7 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 points to obtain even modest accuracy. Your **problem** is to make Monte Carlo integration more efficient for rapidly varying integrands.

6.7.1 Variance Reduction (Method)

If the function being integrated never differs much from its average value, then the standard Monte Carlo mean value method (6.48) should work well with a large, but manageable, 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 [Pres 94, Pres 00] and [Koon 86] 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 $[a, b]$:

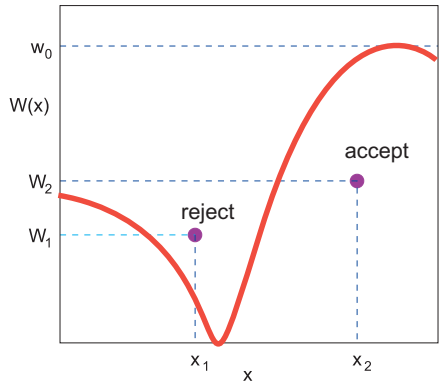
$$|f(x) - g(x)| \leq \epsilon, \qquad \int_a^b dx\, g(x) = J. \tag{6.52}$$

We now evaluate the integral of $f(x) - g(x)$ and add the result to J to obtain the required integral

$$\int_a^b dx\, f(x) = \int_a^b dx\, [f(x) - g(x)] + J. \tag{6.53}$$

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.

Figure 6.6 The Von Neumann rejection technique for generating random points with weight $W(x)$. A random point is accepted if it lies below the curve of $W(x)$ and rejected if it lies above. This generates a random distribution weighted by whatever function is plotted.



6.7.2 Importance Sampling (Method)

A second method for improving 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

$$I = \int_a^b dx\, f(x) = \int_a^b dx\, w(x) \frac{f(x)}{w(x)}. \tag{6.54}$$

If we now use $w(x)$ as the *weighting function* or *probability distribution* for our random numbers, the integral can be approximated as

$$I = \left\langle \frac{f}{w} \right\rangle \simeq \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{w(x_i)}. \tag{6.55}$$

The improvement from (6.55) is that a judicious choice of weighting function $w(x) \propto f(x)$ makes $f(x)/w(x)$ more constant and thus easier to integrate.

6.7.3 Von Neumann Rejection (Method)

A simple, 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 a pond, only now the pond has been 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 (Figure 6.6). 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:

$$(x_i, W_i) = (r_{2i-1}, W_0 r_{2i}). \tag{6.56}$$

We then reject all x_i that do not fall into the pond:

$$\text{If } W_i < w(x_i), \text{ accept}, \quad \text{If } W_i > w(x_i), \text{ reject}. \tag{6.57}$$

The x_i values so accepted will have the weighting $w(x)$ (Figure 6.6). The largest acceptance occurs where $w(x)$ is large, in this case for midrange x. In Chapter 15, "Thermodynamic

Simulations & Feynman Quantum Path Integration," we apply a variation of the rejection technique known as the *Metropolis algorithm*. This algorithm has now become the cornerstone of computation thermodynamics.

Listing 6.3 **vonNeuman.py** uses vonNeuman rejection to generate a weighted random distribution of numbers.

```
#  vonNeuman: Monte-Carlo integration via stone throwing

import random
from visual.graph import *

N        = 100   # points to plot the function
graph    = display(width=500,height=500,title='vonNeumann Rejection Int')
xsinx    = curve(x=list(range(0,N)), color=color.yellow, radius=0.5)
pts      = label(pos=(-60, -60), text='points=', box=0)          # labels
pts2     = label(pos=(-30, -60), box=0)
inside   = label(pos=(30,-60), text='accepted=', box=0)
inside2  = label(pos=(60,-60), box=0)
arealbl  = label(pos=(-65,60), text='area=', box=0)
arealbl2 = label(pos=(-35,60), box=0)
areanal  = label(pos=(30,60), text='analytical=', box=0)
zero     = label(pos=(-85,-48), text='0', box=0)
five     = label(pos=(-85,50), text='5', box=0)
twopi    = label(pos=(90,-48), text='2pi',box=0)

def fx (x):  return x*sin(x)*sin(x)                          # Integrand

def plotfunc():              # Plot function and the box
    incr = 2.0*pi/N
    for i in range(0,N):
        xx          = i*incr
        xsinx.x[i] = ((80.0/pi)*xx-80)
        xsinx.y[i] = 20*fx(xx)-50
    box             = curve(pos=[(-80,-50), (-80,50), (80,50)
                    ,(80,-50), (-80,-50)], color=color.white)        # box

plotfunc()                     # the area of the box = height*width=5*2pi
j           = 0
Npts        = 3001                     # points generated inside the box
analyt      = (pi)**2                                  # analytical integral
areanal.text = 'analytical=%8.5f'%analyt     # Output analytical integral
genpts      = points(size=2)
for i in range(1,Npts):                # generates points inside the box
    rate(500)                # to slow the process to see point generation
    x = 2.0*pi*random.random()                                # 0=< x <= 2pi
    y = 5*random.random()                                       # 0=< x <= 5
    xp = x*80.0/pi-80
    yp = 20.0*y-50
    pts2.text = '%4s' %i
    if y <= fx(x):                                     # case below the curve
        j += 1                                  # increase count below curve
        genpts.append(pos=(xp,yp), color=color.cyan)
        inside2.text='%4s'%j                        # accepted points label
    else:  genpts.append(pos=(xp,yp), color=color.green)
    boxarea = 2.0*pi*5.0                          # area of box propto Npts
    area = boxarea*j/(Npts-1)                      # area of curve propto j
    arealbl2.text = '%8.5f'%area              # write current computed area
```

6.7.4 Simple 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

$$x_N = \sum_{i=1}^{N} r_i \tag{6.58}$$

is distributed normally. This means that the generated x values have the distribution

$$P_N(x) = \frac{\exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]}{\sqrt{2\pi\sigma^2}}, \quad \mu = N\langle r\rangle, \;\; \sigma^2 = N(\langle r^2\rangle - \langle r\rangle^2). \tag{6.59}$$

6.8 NONUNIFORM ASSESSMENT ⊙

Use the von Neumann rejection technique to generate a normal distribution of standard deviation 1 and compare to the simple Gaussian method.

6.8.1 Implementation⊙

In order for $w(x)$ to be the weighting function for random numbers over $[a, b]$, we want it to have the properties

$$\int_a^b dx\, w(x) = 1, \quad [w(x) > 0], \quad d\mathcal{P}(x \to x + dx) = w(x)\, dx, \tag{6.60}$$

where $d\mathcal{P}$ is the probability of obtaining an x in the range $x \to x + dx$. 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 takes our original integral I (6.54) to the form

$$I = \int_a^b dx\, f(x) = \int_0^1 dW \frac{f[x(W)]}{w[x(W)]}. \tag{6.61}$$

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 $[0, 1]$,

$$u(r) = \begin{cases} 1, & \text{for } 0 \le r \le 1, \\ 0, & \text{otherwise.} \end{cases} \tag{6.62}$$

We want to find a mapping $r \leftrightarrow x$ or probability function $w(x)$ for which probability is conserved:

$$w(x)\, dx = u(r)\, dr, \quad \Rightarrow \quad w(x) = \left|\frac{dr}{dx}\right| u(r). \tag{6.63}$$

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 $x \to x + dx$ equal to that of r lying in $r \to r + dr$.

To find the mapping between x and r (the tricky part), we change variables to $W(x)$ defined by the integral

$$W(x) = \int_{-\infty}^{x} dx'\, w(x'). \tag{6.64}$$

We recognize $W(x)$ as the (incomplete) integral of the probability density $u(r)$ up to some point x. It is another type of distribution function, 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 (6.64) that $W(x)$

has the properties

$$W(-\infty)=0; \quad W(\infty)=1, \tag{6.65}$$

$$\frac{dW(x)}{dx}=w(x), \quad dW(x)=w(x)\,dx=u(r)\,dr. \tag{6.66}$$

Consequently, $W_i=\{r_i\}$ is a uniform sequence of random numbers, and we just need to invert (6.64) to obtain x values distributed with probability $w(x)$.

The crux of this technique is being able to invert (6.64) 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 with the familiar uniform distribution

$$w(x)=\begin{cases}\frac{1}{b-a}, & \text{if } a\le x\le b,\\ 0, & \text{otherwise.}\end{cases} \tag{6.67}$$

After following the rules, this leads to

$$W(x)=\int_a^x dx'\frac{1}{b-a}=\frac{x-a}{b-a} \tag{6.68}$$

$$\Rightarrow \quad x=a+(b-a)W \quad \Rightarrow \quad W^{-1}(r)=a+(b-a)r, \tag{6.69}$$

where $W(x)$ is always taken as uniform. In this way we generate uniform random $0\le r\le 1$ and uniform random $a\le x\le b$.

Exponential weight: We want random points with an exponential distribution:

$$\begin{aligned} w(x) &= \begin{cases}\frac{1}{\lambda}e^{-x/\lambda}, & \text{for } x>0,\\ 0, & \text{for } x<0,\end{cases}\\ W(x) &= \int_0^x dx'\frac{1}{\lambda}e^{-x'/\lambda}=1-e^{-x/\lambda}, \end{aligned} \tag{6.70}$$

$$\Rightarrow \quad x = -\lambda\ln(1-W)\equiv-\lambda\ln(1-r). \tag{6.71}$$

In this way we generate uniform random $r\colon[0,1]$ and obtain $x=-\lambda\ln(1-r)$ distributed with an exponential probability distribution for $x>0$. Notice that our prescription (6.54) and (6.55) tells us to use $w(x)=e^{-x/\lambda}/\lambda$ to remove the exponential-like behavior from an integrand and place it in the weights and scaled points $(0\le x_i\le\infty)$. Because the resulting integrand will vary less, it may be approximated better as a polynomial:

$$\int_0^\infty dx\,e^{-x/\lambda}f(x)\simeq\frac{\lambda}{N}\sum_{i=1}^N f(x_i), \quad x_i=-\lambda\ln(1-r_i). \tag{6.72}$$

Gaussian (normal) distribution: We want to generate points with a normal distribution:

$$w(x')=\frac{1}{\sqrt{2\pi}\sigma}e^{-(x'-\overline{x})^2/2\sigma^2}. \tag{6.73}$$

This by itself is rather hard but is made easier by generating uniform distributions in angles and then using trigonometric relations to convert them to a Gaussian distribution. But before doing that, we keep things simple by realizing that we can obtain (6.73) with mean $\overline{x}$ and standard deviation σ by scaling and a translation of a simpler $w(x)$:

$$w(x)=\frac{1}{\sqrt{2\pi}}\,e^{-x^2/2}, \quad x'=\sigma x+\overline{x}. \tag{6.74}$$

We start by generalizing the statement of probability conservation for two different distributions (6.63) to two dimensions [Pres 94]:

$$p(x,y)\,dx\,dy = u(r_1,r_2)\,dr_1\,dr_2 \tag{6.75}$$

$$\Rightarrow \quad p(x,y) = u(r_1,r_2)\left|\frac{\partial(r_1,r_2)}{\partial(x,y)}\right|. \tag{6.76}$$

We recognize the term in vertical bars as the Jacobian determinant:

$$J = \left|\frac{\partial(r_1,r_2)}{\partial(x,y)}\right| \stackrel{\text{def}}{=} \frac{\partial r_1}{\partial x}\frac{\partial r_2}{\partial y} - \frac{\partial r_2}{\partial x}\frac{\partial r_1}{\partial y}. \tag{6.77}$$

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

$$x = \sqrt{-2\,\ln r_1}\cos 2\pi r_2, \quad y = \sqrt{-2\,\ln r_1}\sin 2\pi r_2. \tag{6.78}$$

The inversion of this mapping produces the Gaussian distribution

$$r_1 = e^{-(x^2+y^2)/2}, \quad r_2 = \frac{1}{2\pi}\tan^{-1}\frac{y}{x}, \quad J = -\frac{e^{-(x^2+y^2)/2}}{2\pi}. \tag{6.79}$$

The solution to our problem is at hand. We use (6.78) with r_1 and r_2 uniform random distributions, and x and y are then Gaussian random distributions centered around $x = 0$.

Chapter Seven

Differentiation & Searching

In this chapter we add two more tools to our computational toolbox: numerical differentiation and trial-and-error searching. In Unit I we derive the forward-difference, central-difference, and extrapolated-difference methods for differentiation. They will be used throughout the book, especially for partial differential equations. In Unit II we devise ways to search for solutions to nonlinear equations by trial and error and apply our new-found numerical differentiation tools there. Although trial-and-error searching may not sound very precise, it is in fact widely used to solve problems where analytic solutions do not exist or are not practical. In Chapter 8, "Solving Systems of Equations with Matrices; Data Fitting," *we extend these search and differentiation techniques to the solution of simultaneous equations using matrix techniques. In Chapter 9,* "Differential Equation Applications," *we combine trial-and-error searching with the solution of ordinary differential equations to solve the quantum eigenvalue problem.*

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Numerical Differentiation	pdf	7.1–7.6	Trial and Error Searching	pdf	7.7–7.10
N-Dimensional Searching	pdf	8.2			

7.1 UNIT I. NUMERICAL DIFFERENTIATION

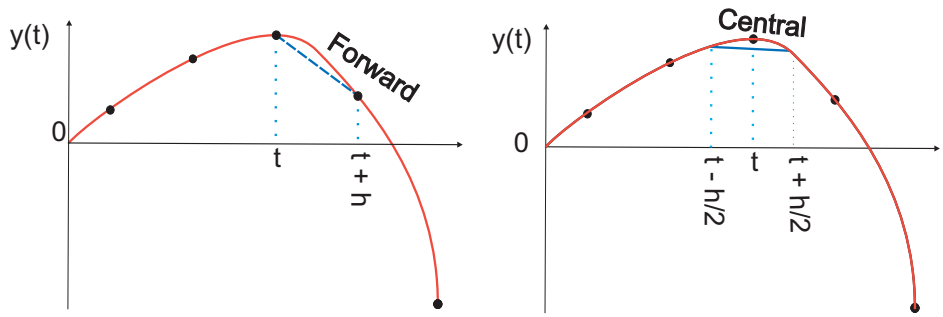
Problem: Figure 7.1 shows the trajectory of a projectile with air resistance. The dots indicate the times t at which measurements were made and tabulated. Your **problem** is to determine the velocity dy/dt as a function of time. Note that since there is realistic air resistance present, there is no analytic function to differentiate, only this table of numbers.

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

$$\frac{dy(t)}{dt} \stackrel{\text{def}}{=} \lim_{h\to 0} \frac{y(t+h)-y(t)}{h}. \tag{7.1}$$

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

Figure 7.1 Forward-difference approximation (slanted dashed line) and central-difference approximation (horizontal line) for the numerical first derivative at time t. (A tangent to the curve at t yields the correct derivative.) The central difference is seen to be more accurate. (The trajectory is that of a projectile with air resistance.)



7.2 FORWARD DIFFERENCE (ALGORITHM)

The most direct method for numerical differentiation starts by expanding a function in a Taylor series to obtain its value a small step h away:

$$y(t+h) = y(t) + h\,\frac{dy(t)}{dt} + \frac{h^2}{2!}\,\frac{d^2y(t)}{dt^2} + \frac{h^3}{3!}\,\frac{dy^3(t)}{dt^3} + \cdots . \tag{7.2}$$

We obtain the *forward-difference* derivative algorithm by solving (7.2) for $y'(t)$:

$$\left.\frac{dy(t)}{dt}\right|_{\text{fd}} \stackrel{\text{def}}{=} \frac{y(t+h)-y(t)}{h}. \tag{7.3}$$

An approximation for the error follows from substituting the Taylor series:

$$\left.\frac{dy(t)}{dt}\right|_{\text{fd}} \simeq \frac{dy(t)}{dt} + \frac{h}{2}\,\frac{dy^2(t)}{dt^2} + \cdots . \tag{7.4}$$

You can think of this approximation as using two points to represent the function by a straight line in the interval from x to $x+h$ (Figure 7.1 left).

The approximation (7.3) has an error proportional to h (unless the heavens look down upon you kindly and make y'' vanish). We can make the approximation error smaller by making h smaller, yet precision will be lost through the subtractive cancelation on the left-hand side (LHS) of (7.3) for too small an h. To see how the forward-difference algorithm works, let $y(t) = a + bt^2$. The exact derivative is $y' = 2bt$, while the computed derivative is

$$\left.\frac{dy(t)}{dt}\right|_{\text{fd}} \simeq \frac{y(t+h)-y(t)}{h} = 2bt + bh. \tag{7.5}$$

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

7.3 CENTRAL DIFFERENCE (ALGORITHM)

An improved approximation to the derivative starts with the basic definition (7.1) or geometrically as shown in Figure 7.1 on the right. Now, rather than making a single step of h forward, we form a *central difference* by stepping forward half a step and backward half a step:

$$\left.\frac{dy(t)}{dt}\right|_{\text{cd}} \equiv D_{cd}\,y(t) \stackrel{\text{def}}{=} \frac{y(t+h/2)-y(t-h/2)}{h}. \tag{7.6}$$

We estimate the error in the central-difference algorithm by substituting the Taylor series for $y(t \pm h/2)$ into (7.6):

$$
\begin{aligned}
y\left(t+\frac{h}{2}\right) - y\left(t-\frac{h}{2}\right) &\simeq \left[y(t) + \frac{h}{2}y'(t) + \frac{h^2}{8}y''(t) + \frac{h^3}{48}y'''(t) + \mathcal{O}(h^4)\right] \\
&\quad - \left[y(t) - \frac{h}{2}y'(t) + \frac{h^2}{8}y''(t) - \frac{h^3}{48}y'''(t) + \mathcal{O}(h^4)\right] \\
&= hy'(t) + \frac{h^3}{24}y'''(t) + \mathcal{O}(h^5), \\
\Rightarrow \quad \left.\frac{dy(t)}{dt}\right|_{\text{cd}} &\simeq y'(t) + \frac{1}{24}h^2 y'''(t) + \mathcal{O}(h^4). \qquad (7.7)
\end{aligned}
$$

The important difference between this algorithm and the forward difference from (7.3) is that when $y(t-h/2)$ is subtracted from $y(t+h/2)$, all terms containing an even power of h in the two Taylor series cancel. This make the central-difference algorithm accurate to order h^2 (h^3 before division by h), while the forward difference is accurate only to order h. If the $y(t)$ is smooth, that is, if $y'''h^2/24 \ll y''h/2$, then you can expect the central-difference error to be smaller. If we now return to our parabola example (7.5), we will see that the central difference gives the exact derivative independent of h:

$$
\left.\frac{dy(t)}{dt}\right|_{\text{cd}} \simeq \frac{y(t+h/2) - y(t-h/2)}{h} = 2bt. \qquad (7.8)
$$

7.4 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 reduce the theoretical error further by forming a combination of algorithms whose summed errors extrapolate to zero. One algorithm is the central-difference algorithm (7.6) using a half-step back and a half-step forward. The second algorithm is another central-difference approximation using quarter-steps:

$$
\begin{aligned}
\left.\frac{dy(t,h/2)}{dt}\right|_{\text{cd}} &\stackrel{\text{def}}{=} \frac{y(t+h/4) - y(t-h/4)}{h/2} \qquad (7.9) \\
&\simeq y'(t) + \frac{h^2}{96}\frac{d^3y(t)}{dt^3} + \cdots .
\end{aligned}
$$

A combination of the two eliminates both the quadratic and linear terms:

$$
\begin{aligned}
\left.\frac{dy(t)}{dt}\right|_{\text{ed}} &\stackrel{\text{def}}{=} \frac{4D_{\text{cd}}y(t,h/2) - D_{\text{cd}}y(t,h)}{3} \qquad (7.10) \\
&\simeq \frac{dy(t)}{dt} - \frac{h^4 y^{(5)}(t)}{4 \times 16 \times 120} + \cdots . \qquad (7.11)
\end{aligned}
$$

Here (7.10) is the extended-difference algorithm, (7.11) gives its error, and D_{cd} represents the central-difference algorithm. If $h = 0.4$ and $y^{(5)} \simeq 1$, then there will be only one place of round-off error and the truncation error will be approximately machine precision ϵ_m; this is the best you can hope for.

When working with these and similar higher-order methods, it is important to remem-

ber that while they may work as designed for well-behaved functions, they may fail badly for functions containing noise, as may data from computations or measurements. If noise is evident, it may be better to first smooth the data or fit them with some analytic function using the techniques of Chapter 8, "Solving Systems of Equations with Matrices; Data Fitting," and then differentiate.

7.5 ERROR ANALYSIS (ASSESSMENT)

The approximation errors in numerical differentiation decrease with decreasing step size h, while round-off errors increase with decreasing step size (you have to take more steps and do more calculations). Remember from our discussion in Chapter 2, "Errors & Uncertainties in Computations," that the best approximation occurs for an h that makes the total error $\epsilon_{\text{approx}} + \epsilon_{\text{ro}}$ a minimum, and that as a rough guide this occurs when $\epsilon_{\text{ro}} \simeq \epsilon_{\text{approx}}$.

We have already estimated the approximation error in numerical differentiation rules by making a Taylor series expansion of $y(x+h)$. The approximation error with the forward-difference algorithm (7.3) is $\mathcal{O}(h)$, while that with the central-difference algorithm (7.7) is $\mathcal{O}(h^2)$:

$$\epsilon_{\text{approx}}^{\text{fd}} \simeq \frac{y''h}{2}, \quad \epsilon_{\text{approx}}^{\text{cd}} \simeq \frac{y'''h^2}{24}. \tag{7.12}$$

To obtain a rough estimate of the round-off error, we observe that differentiation essentially subtracts the value of a function at argument x from that of the same function at argument $x+h$ and then divides by h: $y' \simeq [y(t+h) - y(t)]/h$. As h is made continually smaller, we eventually reach the round-off error limit where $y(t+h)$ and $y(t)$ differ by just machine precision ϵ_m:

$$\epsilon_{\text{ro}} \simeq \frac{\epsilon_m}{h}. \tag{7.13}$$

Consequently, round-off and approximation errors become equal when

$$\begin{aligned} &\epsilon_{\text{ro}} \simeq \epsilon_{\text{approx}}, \\ &\frac{\epsilon_m}{h} \simeq \epsilon_{\text{approx}}^{\text{fd}} = \frac{y^{(2)}h}{2}, \qquad \frac{\epsilon_m}{h} \simeq \epsilon_{\text{approx}}^{\text{cd}} = \frac{y^{(3)}h^2}{24}, \\ &\Rightarrow \quad h_{\text{fd}}^2 = \frac{2\epsilon_m}{y^{(2)}}, \qquad \Rightarrow \quad h_{\text{cd}}^3 = \frac{24\epsilon_m}{y^{(3)}}. \end{aligned} \tag{7.14}$$

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

$$\begin{aligned} &h_{\text{fd}} \simeq 4 \times 10^{-8}, \qquad h_{\text{cd}} \simeq 3 \times 10^{-5}, \\ &\Rightarrow \quad \epsilon_{\text{fd}} \simeq \frac{\epsilon_m}{h_{\text{fd}}} \simeq 3 \times 10^{-8}, \qquad \Rightarrow \quad \epsilon_{\text{cd}} \simeq \frac{\epsilon_m}{h_{\text{cd}}} \simeq 3 \times 10^{-11}. \end{aligned} \tag{7.15}$$

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 about 1000 times smaller than the error in the forward-difference method.

The programming for numerical differentiation is so simple that we need give only the lines here

```
FD = ( y(t+h) − y(t) ) /h;                    // forward diff
CD = ( y(t+h/2) − y(t−h/2) ) /h;              // central diff
ED = (8*(y(t+h/4)−y(t−h/4)) − (y(t+h/2)−y(t−h/2)))/3/h; //extrap
```

1. Use forward-, central-, and extrapolated-difference algorithms to differentiate the functions $\cos t$ and e^t at $t = 0.1, 1.$, 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 $h \simeq \epsilon_m$.
   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 algorithmic (series truncation) error dominates at large h and round-off error at small h in your plot. Do the slopes agree with our model's predictions?

7.6 SECOND DERIVATIVES (PROBLEM)

Let's say that you have measured the position $y(t)$ *versus* time for a particle (Figure 7.1). Your **problem** now is to determine the force on the particle. Newton's second law tells us that force and acceleration are linearly related:

$$F = ma = m\frac{d^2y}{dt^2}, \tag{7.16}$$

where F is the force, m is the particle's mass, and a is the acceleration. So if we can determine the acceleration $a(t) = d^2y/dt^2$ from the $y(t)$ 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 cancelations. Let's look again at the central-difference method:

$$\frac{dy(t)}{dt} \simeq \frac{y(t+h/2) - y(t-h/2)}{h}. \tag{7.17}$$

This algorithm gives the derivative at t by moving forward and backward from t by $h/2$. We take the second derivative d^2y/dt^2 to be the central difference of the first derivative:

$$\frac{d^2y(t)}{dt^2} \simeq \frac{y'(t+h/2) - y'(t-h/2)}{h}$$

$$\simeq \frac{[y(t+h) - y(t)] - [y(t) - y(t-h)]}{h^2} \tag{7.18}$$

$$= \frac{y(t+h) + y(t-h) - 2y(t)}{h^2}. \tag{7.19}$$

As we did for first derivatives, we determine the second derivative at t by evaluating the function in the region surrounding t. Although the form (7.19) is more compact and requires fewer steps than (7.18), it may increase subtractive cancelation by first storing the "large" number $y(t+h) + y(t-h)$ and then subtracting another large number $2y(t)$ from it. We ask you to explore this difference as an exercise.

7.6.1 Second-Derivative Assessment

Write a program to calculate the second derivative of $\cos t$ using the central-difference algorithms (7.18) and (7.19). Test it over four cycles. Start with $h \simeq \pi/10$ and keep reducing h until you reach machine precision.

7.7 UNIT II. TRIAL-AND-ERROR SEARCHING

Many computer techniques are well-defined sets of procedures leading to definite outcomes. In contrast, some computational techniques are trial-and-error algorithms in which decisions on what steps to follow are made based on the current values of variables, and the program quits only when it thinks it has solved the problem. (We already did some of this when we summed a power series until the terms became small.) Writing this type of program is usually interesting because we must foresee how to have the computer act intelligently in all possible situations, and running them is very much like an experiment in which it is hard to predict what the computer will come up with.

7.8 QUANTUM STATES IN A SQUARE WELL (PROBLEM)

Probably the most standard problem in quantum mechanics[1] is to solve for the energies of a particle of mass m bound within a 1-D square well of radius a:

$$V(x) = \begin{cases} -V_0, & \text{for } |x| \le a, \\ 0, & \text{for } |x| \ge a. \end{cases} \tag{7.20}$$

As shown in quantum mechanics texts [Gott 66], the energies of the bound states $E = -E_B < 0$ within this well are solutions of the transcendental equations

$$\sqrt{10 - E_B}\,\tan\left(\sqrt{10 - E_B}\right) = \sqrt{E_B} \quad \text{(even)}, \tag{7.21}$$

$$\sqrt{10 - E_B}\,\operatorname{cotan}\left(\sqrt{10 - E_B}\right) = \sqrt{E_B} \quad \text{(odd)}, \tag{7.22}$$

where even and odd refer to the symmetry of the wave function. Here we have chosen units such that $\hbar = 1$, $2m = 1$, $a = 1$, and $V_0 = 10$. Your **problem** is to

1. Find several bound-state energies E_B for even wave functions, that is, the solution of (7.21).
2. 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.

7.9 TRIAL-AND-ERROR ROOTS VIA BISECTION ALGORITHM

Trial-and-error root finding looks for a value of x at which

$$f(x) = 0,$$

where the 0 on the right-hand side (RHS) is conventional (an equation such as $10 \sin x = 3x^3$ can easily be written as $10 \sin x - 3x^3 = 0$). The search procedure starts with a guessed value for x, substitutes that guess into $f(x)$ (the "trial"), and then sees how far the LHS is from zero (the "error"). The program then changes x based on the error and tries out the new guess in $f(x)$. The procedure continues until $f(x) \simeq 0$ to some desired level of precision, until the changes in x are insignificant, or until it appears that progress is not being made.

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

[1] We solve this same problem in §9.9 using an approach that is applicable to almost any potential and which also provides the wave functions. The approach of this section works only for the eigenenergies of a square well.

zero occurs. Other techniques, such as the Newton–Raphson method we describe next, may converge more quickly, but if the initial guess is not close, it may become unstable and fail completely.

The basis of the bisection algorithm is shown on the left in Figure 7.2. 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_+:

$$f(x_-) < 0, \quad f(x_+) > 0. \tag{7.23}$$

(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 (implemented in **Bisection.py**) then picks the new x as the bisection of the interval and selects as its new interval the half in which the sign change occurs:

```
x = ( xPlus + xMinus ) / 2
if ( f(x) f(xPlus) > 0 ) xPlus = x
 else xMinus = x
```

This process continues until the value of $f(x)$ is less than a predefined level of precision or until a predefined (large) number of subdivisions occurs.

Listing 7.1 The **Bisection.py** is a simple implementation of the bisection algorithm for finding a zero of a function, in this case $2\cos x - x$.

```
# Bisection.py  Bisection algorithm finds a zero of f(x)

from visual import *
from visual.graph import *

eps = 1e-6                                          # Precision of zero
xminus = 0.0; xplus = 7.0                   # x range must contain a zero
imax = 100                                          # Max no. iterations

def f(x):                                   # the function to find zero of
    return 2*cos(x) - x

print("Iteration  x              f(x)")

for it in range(0, imax):
    x = ( xplus  +  xminus )/2.
    print("  ", it, "   ", x, "  ", f(x))
    if ( f(xplus)*f(x) > 0. ):
        xplus = x                                   # Bisection algorithm
    else:
        xminus =  x
    if ( abs(f(x) ) < eps ):                        # Convergence check
        break
print("Enter and return a character to finish")
s = raw_input()
```

The example in Figure 7.2 on the left 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 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 become too small for us to show.

7.9.1 Bisection Algorithm Implementation

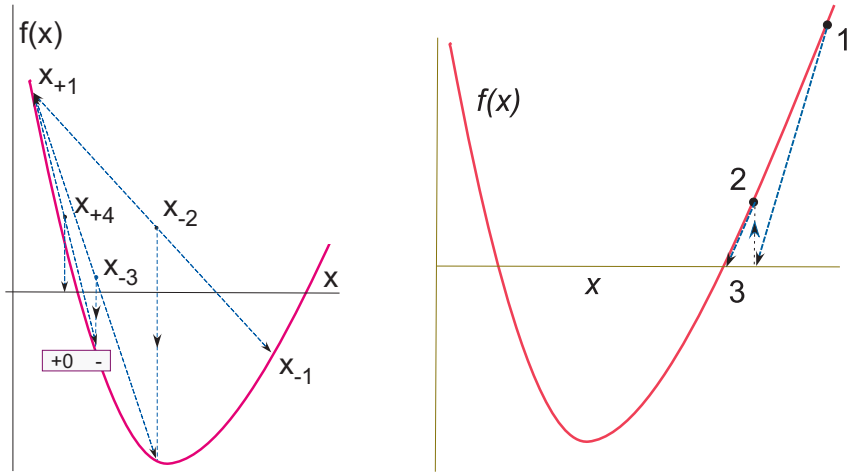
1. The first step in implementing any search algorithm is to get an idea of what your function looks like. For the present problem you do this by making a plot of $f(E) = \sqrt{10 - E_B}\,\tan(\sqrt{10 - E_B}) - \sqrt{E_B}$ *versus* E_B. Note from your plot some approximate values at which $f(E_B) = 0$. 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 (7.21).
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 (7.21) is
$$\sqrt{E}\,\cot(\sqrt{10 - E}) - \sqrt{10 - E} = 0. \tag{7.24}$$
4. Make a second plot of (7.24), 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.

7.10 NEWTON–RAPHSON SEARCHING (IMPROVED ALGORITHM)

The Newton–Raphson algorithm finds approximate roots of the equation
$$f(x) = 0$$
more quickly than the bisection method. As we see graphically in Figure 7.2 on the right, 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 of the line with the x axis at $x = -b/m$ 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 (Figure 7.2), then the convergence is much more rapid than for the bisection algorithm.

The analytic formulation of the Newton–Raphson algorithm starts with an old guess x_0 and expresses a new guess x as a correction Δx to the old guess:
$$x_0 = \text{old guess}, \quad \Delta x = \text{unknown correction} \tag{7.25}$$
$$\Rightarrow \quad x = x_0 + \Delta x = \text{(unknown) new guess}. \tag{7.26}$$
We next expand the known function $f(x)$ in a Taylor series around x_0 and keep only the linear terms:
$$f(x = x_0 + \Delta x) \simeq f(x_0) + \left.\frac{df}{dx}\right|_{x_0} \Delta x. \tag{7.27}$$
We then determine the correction Δx by calculating the point at which this linear approximation to $f(x)$ crosses the x axis:
$$f(x_0) + \left.\frac{df}{dx}\right|_{x_0} \Delta x = 0, \tag{7.28}$$
$$\Rightarrow \quad \Delta x = -\frac{f(x_0)}{df/dx|_{x_0}}. \tag{7.29}$$

Figure 7.2 A graphical representation of the steps involved in solving for a zero of $f(x)$ using the bisection algorithm (*left*) and the Newton–Raphson method (*right*). 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. The Newton–Raphson method takes the new guess as the zero of the line tangent to $f(x)$ at the old guess. Four steps are shown for the bisection algorithm, but only two for the more rapidly convergent Newton–Raphson method.



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

The Newton–Raphson algorithm (7.29) 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 build it into the algorithm. However, especially for more complicated problems, it is simpler and less error-prone to use a numerical forward-difference approximation to the derivative:[2]

$$\frac{df}{dx} \simeq \frac{f(x+\delta x) - f(x)}{\delta x}, \tag{7.30}$$

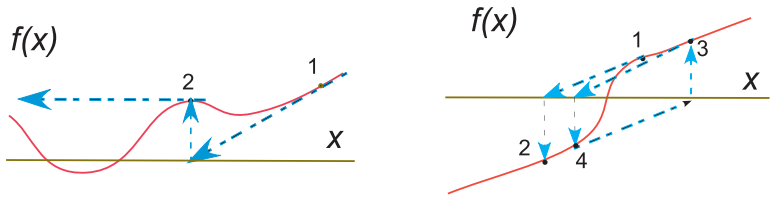
where δx is some small change in x that you just chose [different from the Δ used for searching in (7.29)]. While a central-difference approximation for the derivative would be more accurate, it would require additional evaluations of the f's, and once you find a zero, it does not matter how you got there. In Listing 7.2) and **`NewtonFD.py`** we give the programs **`NewtonCD.py`** that implement the derivative both ways.

Listing 7.2 **NewtonCD.py** 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.

```
# NewtonCD.py     Newton Search with central difference

from sys import version
if int(version[0])>2:   # raw_input deprecated in Python 3
    raw_input=input
from math import cos

x = 4.;          dx = 3.e-1;        eps = 0.2;                # Parameters
imax = 100;                                          # Max no of iterations
def f(x):                                                   # function def
    return 2*cos(x) - x

for it in range(0, imax + 1):
    F = f(x)
    if ( abs(F) <= eps ):                            # Check for convergence
        print("Root found, tolerance eps = " , eps)
        break
    print("Iteration # = ", it , " x = ", x, " f(x) = ", F)
    df = ( f(x + dx/2)  -  f(x - dx/2) )/dx          # Central diff deriv
    dx = - F/df
```

[2]We discuss numerical differentiation in Chapter 7, "Differentiation & Searching."

Figure 7.3 Two examples of how the Newton–Raphson algorithm 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/maximum, 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. The technique know as "backtracking" could eliminate this problem.




```
    x    += dx                                          # New guess
print("Enter and return any character to quit")
s = raw_input()
```

7.10.1 Newton–Raphson with Backtracking

Two examples of possible problems with the Newton–Raphson algorithm are shown in Figure 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 with a different guess and pray that you do not fall into this trap again. In 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.

In Figure 7.3 on the right 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, in 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 your search with the bisection algorithm and then switch to the faster Newton–Raphson algorithm when you get closer to the zero.

7.10.2 Newton–Raphson Implementation

1. Use the Newton–Raphson algorithm to find some energies E_B that are solutions of (7.21). 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 closer to the answer for the Newton–Raphson algorithm to work.)

3. Modify your algorithm to include backtracking and then try it out on some problem cases.

Chapter Eight

Solving Systems of Equations with Matrices; Data Fitting

Unit I applies the trial-and-error techniques developed in Chapter 7, "Differentiation & Searching," *to solve a set of simultaneous nonlinear equations. This leads us into general matrix computing using scientific libraries. In Unit II we look at several ways in which theoretical formulas are fit to data and see that these often require the matrix techniques of Unit I.*

VIDEO LECTURES, APPLETS AND ANIMATIONS FOR THIS CHAPTER

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Matrix Computing	pdf	8.1	Matrix Computing II	pdf	8.4
Trial and Error Searching	pdf	7.7–7.10	N-Dimensional Searching	pdf	8.2.2
Interpolation	pdf	8.5	Least Square Fitting	pdf	8.7

Applets

Name	*Sections*
Lagrange Interpolation	8.5
Spline Interpolation	8.5

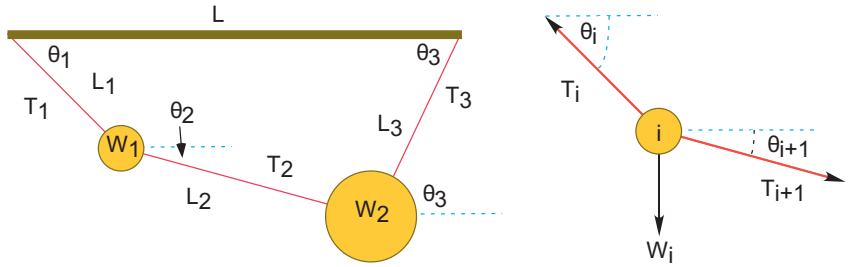
8.1 UNIT I. SYSTEMS OF EQUATIONS, MATRIX COMPUTING

Physical systems are often modeled by systems of simultaneous equations written in matrix form. As the models are made more realistic, the matrices often become large, and computers become an excellent tool for solving such problems. What makes computers so good is that matrix manipulations intrinsically involve the continued repetition of a small number of simple instructions, and algorithms exist to do this quite efficiently. Further speedup may be achieved by *tuning* the codes to the computer's architecture, as discussed in Chapter 14, "High-Performance Computing Hardware, Tuning, & Parallel Computing."

Industrial-strength subroutines for matrix computing are found in well-established scientific libraries. These subroutines are usually an order of magnitude or more faster than the elementary methods found in linear algebra texts,[1] are usually designed to minimize round-off 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 library routines is that you can often run the same program either on a desktop machine or on a parallel supercomputer, with matrix routines automatically adapting to the local architecture.

[1] Although we prize the book [Pres 94] and what it has accomplished, we cannot recommend taking subroutines from it. They are neither optimized nor documented for easy, stand-alone use, whereas the subroutine libraries recommended in this chapter are.

Figure 8.1 *Left:* Two weights connected by three pieces of string and suspended from a horizontal bar of length L. The lengths are all known but the angles and the tensions in the strings need to be determined. *Right:* A free body diagram for one weight in equilibrium. Balancing the forces in the x and y directions for all weights leads to the equations of static equilibrium.



The thoughtful reader may be wondering when a matrix is "large" enough to require the use of a library routine. While in the past large may have meant a good fraction of your computer's random-access memory (RAM), we now advise that a library routine be used whenever the matrix computations are so numerically intensive that you must wait for results. In fact, even if the sizes of your matrices are small, as may occur in graphical processing, there may be library routines designed just for that which speed up your computation.

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, 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 properly. And because some of the libraries are in Fortran, if you are a C programmer you may also be taxed by having to call a Fortran routine from your C program. However, there are now libraries available in most languages.

8.2 TWO MASSES ON A STRING

Problem: Two weights $(W_1, W_2) = (10, 20)$ are hung from three pieces of string with lengths $(L_1, L_2, L_3) = (3, 4, 4)$ and a horizontal bar of length $L = 8$ (Figure 8.1). Find the angles assumed 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 are inhumanely painful to solve analytically. However, we will show you how the computer can solve this problem, but even then only by a trial-and-error technique with no guarantee of success. Your **problem** is to test this solution for a variety of weights and lengths and then to extend it to the three-weight problem (not as easy as it may seem). 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 to see at what point your initial guess gets so bad that the computer is unable to find a physical solution.

8.2.1 Statics (Theory)

We start with the geometric constraints that the horizontal length of the structure is L and that the strings begin and end at the same height (Figure 8.1 left):

$$L_1 \cos\theta_1 + L_2 \cos\theta_2 + L_3 \cos\theta_3 = L, \tag{8.1}$$

$$L_1 \sin\theta_1 + L_2 \sin\theta_2 - L_3 \sin\theta_3 = 0, \tag{8.2}$$

$$\sin^2\theta_1 + \cos^2\theta_1 = 1, \tag{8.3}$$

$$\sin^2\theta_2 + \cos^2\theta_2 = 1, \tag{8.4}$$

$$\sin^2\theta_3 + \cos^2\theta_3 = 1. \tag{8.5}$$

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 (Figure 8.1 right):

$$T_1 \sin\theta_1 - T_2 \sin\theta_2 - W_1 = 0, \tag{8.6}$$

$$T_1 \cos\theta_1 - T_2 \cos\theta_2 = 0, \tag{8.7}$$

$$T_2 \sin\theta_2 + T_3 \sin\theta_3 - W_2 = 0, \tag{8.8}$$

$$T_2 \cos\theta_2 - T_3 \cos\theta_3 = 0. \tag{8.9}$$

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

8.2.2 Multidimensional Searching

Equations (8.1)–(8.9) are nine simultaneous nonlinear equations. While linear equations can be solved directly, nonlinear equations cannot [Pres 00]. You can use the computer to search for a solution by guessing, but there is no guarantee of finding one. We apply to our set the same Newton–Raphson algorithm as used to solve a single equation by renaming the nine unknown angles and tensions as the subscripted variable y_i and placing the variables together as a vector:

$$\vec{y} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} \sin\theta_1 \\ \sin\theta_2 \\ \sin\theta_3 \\ \cos\theta_1 \\ \cos\theta_2 \\ \cos\theta_3 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix}. \tag{8.10}$$

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

$$f_i(x_1, x_2, \ldots, x_N) = 0, \quad i = 1, N, \tag{8.11}$$

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

The solution to these equations requires a set of nine x_i values that 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. The search algorithm is to guess a solution, expand the nonlinear equations into linear form, solve the resulting linear equations, and continue to improve the guesses based on how close the previous one was to making $\vec{f} = 0$.

Explicitly, let the approximate solution at any one stage be the set $\{x_i\}$ and let us assume that there is an (unknown) set of corrections $\{\Delta x_i\}$ for which

$$f_i(x_1 + \Delta x_1, x_2 + \Delta x_2, \ldots, x_9 + \Delta x_9) = 0, \quad i = 1, 9. \tag{8.13}$$

We solve for the approximate Δx_i's by assuming that our previous solution is close enough to the actual one for two terms in the Taylor series to be accurate:

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

We now have a solvable set of nine linear equations in the nine unknowns Δx_i, which we express as a single matrix equation

$$\begin{aligned} f_1 + \partial f_1/\partial x_1 \, \Delta x_1 + \partial f_1/\partial x_2 \, \Delta x_2 + \cdots + \partial f_1/\partial x_9 \, \Delta x_9 &= 0, \\ f_2 + \partial f_2/\partial x_1 \, \Delta x_1 + \partial f_2/\partial x_2 \, \Delta x_2 + \cdots + \partial f_2/\partial x_9 \, \Delta x_9 &= 0, \\ &\ddots \\ f_9 + \partial f_9/\partial x_1 \, \Delta x_1 + \partial f_9/\partial x_2 \, \Delta x_2 + \cdots + \partial f_9/\partial x_9 \, \Delta x_9 &= 0, \end{aligned}$$

$$\begin{pmatrix} f_1 \\ f_2 \\ \ddots \\ f_9 \end{pmatrix} + \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_9 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_9 \\ \ddots & & & \\ \partial f_9/\partial x_1 & \partial f_9/\partial x_2 & \cdots & \partial f_9/\partial x_9 \end{pmatrix} \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \ddots \\ \Delta x_9 \end{pmatrix} = 0.$$

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 the Δx_i values is unknown. We write this equation in matrix notation as

$$\vec{f} + \mathbf{F}'\,\vec{\Delta x} = 0, \quad \Rightarrow \quad \mathbf{F}'\vec{\Delta x} = -\vec{f}, \tag{8.15}$$

$$\vec{\Delta x} = \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \ddots \\ \Delta x_9 \end{pmatrix}, \quad \vec{f} = \begin{pmatrix} f_1 \\ f_2 \\ \ddots \\ f_9 \end{pmatrix}, \quad \mathbf{F}' = \begin{pmatrix} \partial f_1/\partial x_1 & \cdots & \partial f_1/\partial x_9 \\ \partial f_2/\partial x_1 & \cdots & \partial f_2/\partial x_9 \\ \ddots & & \\ \partial f_9/\partial x_1 & \cdots & \partial f_9/\partial x_9 \end{pmatrix}.$$

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

The equation $\mathbf{F}'\vec{\Delta x} = -\vec{f}$ is in the standard form for the solution of a linear equation (often written $\mathbf{A}\vec{x} = \vec{b}$), where $\vec{\Delta x}$ is the vector of unknowns and $\vec{b} = -\vec{f}$. 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 (8.15) is obtained by multiplying both sides of the equation by the inverse of the $\mathbf{F}'$ matrix:

$$\vec{\Delta x} = -\mathbf{F}'^{-1}\vec{f}, \tag{8.16}$$

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 1-D problem, $\Delta x = -(1/f')f$. In fact, one of the reasons we use formal or abstract notation for matrices is to reveal the simplicity that lies within.

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 would be both time-consuming and error-prone. In contrast, especially for more complicated problems, it is straightforward to program a forward-difference approximation for the derivatives,

$$\frac{\partial f_i}{\partial x_j} \simeq \frac{f_i(x_j + \Delta x_j) - f_i(x_j)}{\Delta x_j}, \tag{8.17}$$

where each individual x_j is varied independently since these are partial derivatives and δx_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 also discussed for the 1-D Newton–Raphson method (§7.10.1), 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 Δx_i until $|f|^2 = |f_1|^2 + |f_2|^2 + \cdots + |f_N|^2$ decreases.

8.3 CLASSES OF MATRIX PROBLEMS (MATH)

It helps to remember that the rules of mathematics apply even to the world's most powerful computers. For example, 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. An overdetermined problem is sometimes treated using 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**:

$$\mathbf{A}\vec{x} = \vec{b}, \quad \mathbf{A}_{N\times N}\,\vec{x}_{N\times 1} = \vec{b}_{N\times 1}, \tag{8.18}$$

where $\mathbf{A}$ is a known $N\times N$ matrix, $\vec{x}$ is an unknown vector of length N, and $\vec{b}$ is a known vector of length N. The best way to solve this equation is by Gaussian elimination or lower-upper (LU) decomposition. This yields the vector $\vec{x}$ 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 (8.18) by $\mathbf{A}^{-1}$:

$$\vec{x} = \mathbf{A}^{-1}\,\vec{b}. \tag{8.19}$$

Both the direct solution of (8.18) and the determination of a matrix's inverse are standards in a matrix subroutine library.

If you have to solve the matrix equation

$$\mathbf{A}\vec{x} = \lambda\vec{x}, \tag{8.20}$$

with $\vec{x}$ an unknown vector and λ an unknown parameter, then the direct solution (8.19) will not be of much help because the matrix $\vec{b} = \lambda\vec{x}$ contains the unknowns λ and $\vec{x}$. Equation (8.20) is the *eigenvalue problem*. It is harder to solve than (8.18) because solutions exist for only certain λ values (or possibly none depending on $\mathbf{A}$). We use the identity matrix to rewrite (8.20) as

$$[\mathbf{A} - \lambda\mathbf{I}]\vec{x} = 0, \tag{8.21}$$

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

$$\vec{x} = 0 \quad \text{(trivial solution)}. \tag{8.22}$$

While the trivial solution is a bona fide solution, it is trivial. A more interesting solution requires the existence of a condition that forbids us from multiplying both sides of (8.21) by $[\mathbf{A} - \lambda\mathbf{I}]^{-1}$. That condition is the nonexistence of the inverse, and if you recall that Cramer's rule for the inverse requires division by $\det[\mathbf{A} - \lambda\mathbf{I}]$, it is clear that the inverse fails to exist (and in this way eigenvalues *do* exist) when

$$\det[\mathbf{A} - \lambda\mathbf{I}] = 0. \tag{8.23}$$

The λ values that satisfy this *secular equation* are the eigenvalues of (8.20).

If you are interested in only the eigenvalues, you should look for a matrix routine that solves (8.23). To do that, first you need a subroutine to calculate the determinant of a matrix, and then a search routine to zero in on the solution of (8.23). Such routines are available in libraries. The traditional way to solve the eigenvalue problem (8.20) 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 $\mathbf{U}$:

$$\mathbf{U}\mathbf{A}(\mathbf{U}^{-1}\mathbf{U})\vec{x} = \lambda\mathbf{U}\vec{x}, \tag{8.24}$$

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

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

$$\mathbf{UAU}^{-1} = \begin{pmatrix} \lambda_1' & & \cdots & 0 \\ 0 & \lambda_2' & \cdots & 0 \\ 0 & 0 & \lambda_3' & \cdots \\ 0 & \cdots & & \lambda_N' \end{pmatrix}. \tag{8.26}$$

The diagonal values of $\mathbf{UAU}^{-1}$ are the eigenvalues with eigenvectors

$$\vec{x}_i = \mathbf{U}^{-1}\hat{e}_i; \tag{8.27}$$

that is, the eigenvectors are the columns of the matrix $\mathbf{U}^{-1}$. A number of routines of this type are found in subroutine libraries.

8.3.1 Practical Matrix Computing

Many scientific programming bugs arise from the improper use of arrays.[2] This may be due to the extensive use of matrices in scientific computing or to the complexity of keeping track of indices and dimensions. In any case, here are some rules of thumb to observe.

Computers are finite: Unless you are careful, your matrices will use so much memory that your computation will slow down significantly, especially if it starts to use virtual memory. As a case in point, let's say that you store data in a 4-D array with each index having a *physical dimension* of 100: `A[100] [100] [100] [100]`. This array of $(100)^4$ 64-byte words occupies $\simeq$1 GB 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 2-D square matrix (as happens when the number of integration steps is doubled) leads to an *eightfold* increase in processing time.

Paging: Many operating systems have *virtual memory* in which disk space is used when a program runs out of RAM (see Chapter 14, "High-Performance Computing Hardware, Tuning, and Parallel Computing," 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 a matrix's dimension may lead to an order-of-magnitude increase in execution time.

Matrix storage: While we think of matrices as multidimensional blocks of stored numbers, the computer stores them as linear strings. For instance, a matrix `a[3,3]` in Python, `a[3][3]` in Java is stored in *row-major order* :

$$a_{1,1} \;\; a_{1,2} \;\; a_{1,3} \;\; a_{2,1} \;\; a_{2,2} \;\; a_{2,3} \;\; a_{3,1} \;\; a_{3,2} \;\; a_{3,3} \;\; \ldots,$$

while in Fortran it is stored in *column-major order*

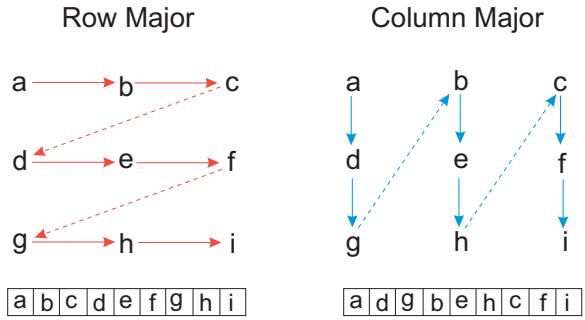
$$a_{1,1} \;\; a_{2,1} \;\; a_{3,1} \;\; a_{1,2} \;\; a_{2,2} \;\; a_{3,2} \;\; a_{1,3} \;\; a_{2,3} \;\; a_{3,3} \;\; \ldots.$$

It is important to keep this linear storage scheme in mind in order to write proper code and to permit the mixing of Python and Fortran programs.

When dealing with matrices, you have to balance the clarity of the operations being performed against the efficiency with which the computer performs 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

[2] Even a vector $V(N)$ is called an array, albeit a 1-D one.

Figure 8.2 *Left:* Row-major order used for matrix storage in Python and Java. *Right:* Column-major order used for matrix storage in Fortran. The table on the bottom shows how successive matrix elements are actually stored in a linear fashion in memory.



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]`.

Subscript 0: It is standard in Python, 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, in addition to the different locations in memory due to row-major and column-major ordering, the same matrix element may 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)`

Physical and logical dimensions: When you run a program, you issue commands such as `zeros((3,3), Float)`, `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 use arrays 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 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, requires 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 elements with primes 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 often need to know *both* the physical and logical sizes of your arrays.

Passing sizes to subprograms ⊙: This 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 those using external libraries, you must also watch that the

sizes of your matrices do not exceed the bounds that 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 Python program that calls a Fortran subroutine, you will 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 with pointers only as subprogram arguments). Here we have a program possibly running some data stored nearby:

```
main                                          // In main program
  dimension a(100), b(400)
  function Sample(a)                            // In subroutine
  dimension a(10)                             // Smaller dimension
  a(300) = 12                        //  Out of bounds, but no message
```

One way to ensure size compatibility among main programs and subroutines is to declare array sizes only in your main program and then pass those sizes along to your subprograms as arguments.

Equivalence, pointers, references manipulations ⊙: Once upon a time computers had such limited memories that programmers conserved 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 the use of **`Common`** and **`Equivalence`** statements in Fortran and by manipulations using pointers and references in other languages. These types of manipulations are now obsolete (the bane of object-oriented programming) and can cause endless grief; do not use them unless it is a matter of "life or death"!

Say what's happening: You decrease programming errors by using self-explanatory labels for your indices (subscripts), stating what your variables mean, and describing your storage schemes.

Tests: Always test a library routine on a small problem whose answer you know (such as the exercises in §8.4.3). Then you'll know if you are supplying it with the right arguments and if you have all the links working.

8.3.2 Implementation: Scientific Libraries, World Wide Web

Some major scientific and mathematical libraries available include the following:

NETLIB	A WWW metalib of free math libraries	ScaLAPACK	Distributed memory LAPACK
LAPACK	Linear Algebra Pack	JLAPACK	LAPACK library in Java
SLATEC	Comprehensive math and statistical pack	ESSL	Engineering and Science Subroutine Library (IBM)
IMSL	International Math and Statistical Libraries	CERNLIB	European Centre for Nuclear Research Library
BLAS	Basic Linear Algebra	JAMA	Java Matrix Library
NAG	Numerical Algorithms Group (UK Labs)	LAPACK ++	Linear algebra in C++
TNT	C++ Template Numerical Toolkit	GNU Scientific GSL	Full scientific libraries in C and C++

Except for ESSL, IMSL, and NAG, all these libraries are in the public domain. However, even the 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 optimized routines for almost everything you might want to do, such as

Linear algebra manipulations	Matrix operations	Interpolation, fitting
Eigensystem analysis	Signal processing	Sorting and searching
Solutions of linear equations	Differential equations	Roots, zeros, and extrema
Random-number operations	Statistical functions	Numerical quadrature

You can search the Web to find out about these libraries or to download one if it is not already on your computer. Alternatively, an excellent place to start looking for a library is Netlib, a repository of free software, documents, and databases of interest to computational scientists.

Linear Algebra Package (LAPACK) is a free, portable, modern (2010) library of Fortran90 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 Basic Linear Algebra Subroutines (BLAS). In contrast to LAPACK, the Sandia, Los Alamos, Air Force Weapons Laboratory Technical Exchange Committee (SLATEC) library contains general-purpose mathematical and statistical Fortran routines and is consequently more general. Nonetheless, it is not as tuned to the architecture of a particular machine as is LAPACK.

8.4 NUMERICAL PYTHON AND PYTHON MATRIX LIBRARY

Numerical Python (numpy) (`numpy.scipy.org`) provides a fast, compact, multidimensional array facility to Python. It is the successor to both Numeric and Numarray. In turn, SciPy (`www.scipy.org/`) is an open source library of scientific tools for Python that supplements NumPy. SciPy includes modules for linear algebra, optimization, integration, special functions, signal and image processing, statistics, genetic algorithms, ODE solvers, and others. In this section we will review some use of the **`array`** data type present in the Visual package, and in §8.4.2 give some examples of the use of the linear algebra package `linalg`.

8.4.1 Arrays in Python:

Python interprets a sequence of ordered items, $L = l_0, l_1, \ldots, l_{N-1}$, as a *list* and represents it with a single symbol **`L`**:

```
>>> L = [1, 2, 3, 4]                    Create list
>>> print L[0]                          Print first element
1
>>> print L                             Print the entire object
[1, 2, 3, 4]                            Python's output
```

Square brackets with comma separtors [.,.,.] are used for lists, while round parenthesis (.,.,.) are used for what Python calls *tuples*. Lists contain sequences of arbitrary objects that are mutable or changeable; tuplets in contrast cannot be changed. As we see in the **`print L[0]`** command, individual elements in a list are referenced by indices, while in the **`print L`** command we see that the whole list can be referenced as an object. While most languages use *arrays* to store sequences and require you to specify the dimension of the array, Python lists are dynamic, which means they adjust their sizes as needed. In our codes we often use tuplets to set up the equivalent of 2-D arrays as for example

```
>>> T = zeros( (Nx,2), float)           Set up array as tuplet
```

where the **zeros** method initializes all entries to 0. in the specified range.

Python can perform a large number of operations on lists, for example,

Operation	Effect	Operation	Effect
L = [1, 2, 3, 4]	Form list	L1 + L2	Concatenate lists
L[i]	i^{th} element	len(L)	Length of list L
i in L	True if i in L	L[i:j]	Slice from i to j
for i in L	Iteration index	L.append(x)	Append x to end of L
L.count(x)	Number of x's in L	L.index(x)	Location of 1st x in L
L.remove(x)	Remove 1st x in L	L.reverse()	Reverse elements in L
L.sort()	Order elements in L		

Although present in Java, C and Fortran, Python itself does not have **array** as a data type. It is, however, contained in the Visual package where it converts a sequence into a 1-D array. *We shall use Visual's array command to form computer representations of vectors and matrices* (which means that you must import Visual in your programs). For example, here we show the results of running our program **Matrix.py** from a shell:

```
>>> from visual import *                                  Need Visual for arrays
>>> vector1 = array([1, 2, 3, 4, 5])                      Fill 1-D array 1
>>> print 'vector1 = ', vector1                          Print entire array object
vector1 = [1 2 3 4 5]                                     Output 3
>>> vector2 = vector1 + vector1                           Add 2 vectors
>>> print 'vector1 + vector1 =', vector2                  Print array 5
vector1 + vector1 = [ 2 4 6 8 10]                         Output
>>> vector2 = 3 * vector1                                 Mult array by scalar 7
>>> print '3 * vector1 = ', vector2                       Print 1-D array
3 * vector1 = [ 3 6 9 12 15]                              Output 9
>>> matrix1 = array(([0,1],[1,3]))                        An array of arrays
>>> print 'matrix1 = ', matrix1                           Print 2-D array 11
matrix1 = [[0 1]                                          13
           [1 3]]
>>> print 'vector1.shape= ', vector1.shape
vector1.shape = (5)                                       Output of dimensions 16
>>> print 'matrix1 * matrix1 = ', matrix1 * matrix1       Matrix multiply
matrix1 * matrix1 = [[0 1]                                Output 18
                     [1 9]]                               20
```

On line 4 we add two 1-D array (vector) objects together and print out the answer to see that it works as expected. Likewise we see on lines 7–9 that multiplication of an array by a constant does in fact multiply each element. On line 10 we construct a "matrix" as a 1-D array of two 1-D arrays, and when we print it out we note that it does indeed look like a matrix. However, on lines 16–20 we multiply two of these matrices together only to discover that the result is not what one normally would expect from matrix multiplication.

8.4.2 Python's linalg Package

We have just seen that while Python is pretty good at treating an **array** object in an abstract sense when we use 1-D arrays for vectors, Python does not create what we normally think of

as abstract matrices when we use 2-D arrays. Fortunately, there is the **LinearAlgebra** package that treats 2-D arrays (a 1-D array of 1-D arrays) as abstract matrices, and also provides a simple interface to the powerful LAPACK linear algebra library. We give some examples in next, and recommend that you use these libraries rather than try to write your own matrix routines.

Our first example of linear algebra is the standard matrix equation

$$\mathbf{A}\vec{x} = \vec{b}. \tag{8.28}$$

This describes a set of linear equations with $\vec{x}$ an unknown vector and $\mathbf{A}$ a known matrix. We take $\mathbf{A}$ to be 3×3, $\vec{b}$ to be 3×1, and let the program figure out that $\vec{x}$ is a 3×1 vector[3]. We start by importing all the packages, by inputting a matrix and a vector, and then by printing out $\mathbf{A}$ and $\vec{x}$:

```
>>> from numpy import *;                                          import visual OK but bigger
>>> from numpy.linalg import*
>>> A = array( [ [1,2,3], [22,32,42], [55,66,100] ] )             Array of arrays
>>> print 'A =',A
A = [[ 1 2 3]
    [ 22 32 42]
    [ 55 66 100]]
>>> b = array([1,2,3])
>>> print 'b =',b
b = [1 2 3]
```

We now solve $\mathbf{A}\vec{x} = \vec{b}$ using the numpy **solve** command, and test how close $\mathbf{A}\vec{x} - \vec{b}$ comes to a zero vector:

```
>>> from numpy.linalg import solve
>>> x = solve(A, b)                                               Does solution
>>> print 'x =', x
xvec = [ -1.4057971 -0.1884058 0.92753623]
>>> print 'Residual =', multiply(A, x) - b                        LHS-RHS
Residual = [2.22044605e-016  1.77635684e-015  1.19904087e-014]    Tiny
```

This is really quite impressive. We have solved the entire set of linear equations (by elimination) with just the single command **solve**, performed a matrix multiplication with the single command **multiply**, did a matrix subtraction with the usual - command, and obtained close to machine precision from the numerics.

Although not as efficient numerically, a direct way of solving $\mathbf{A}\vec{x} = \vec{b}$ is by calculating the inverse $\mathbf{A}^{-1}$, and then multiplying the RHS by the inverse, $\vec{x} = \mathbf{A}^{-1}\vec{b}$:

```
>>> from numpy.linalg import inv
>>> print multiply(inv(A), A)                                     Test out inverse
[[1.00000000e+000 1.13797860e-015 1.47624968e-015]                Identity matrix?
  [-1.38777878e-017 1.00000000e+000 -8.88178420e-016]
  [-6.24500451e-016 -4.16333634e-016 1.00000000e+000]]
>>> print 'x =', multiply(inv(A), b)
x = [-1.4057971 -0.1884058 0.92753623]                            Solution
```

Here we first tested that **inv(A)** finds the inverse of matrix **A** by seeing if **A** times **inv(A)** equals the identity matrix. Then we used the inverse to solve the matrix equation directly, and, within

[3] Don't be bothered by the fact that although we think as these vectors as 3×1, they get printed out as 1×3.

precision, got the same answer as before.

Our second 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:

$$\mathbf{I}\,\vec{\omega} = \lambda\vec{\omega}, \tag{8.29}$$

where $\mathbf{I}$ is the inertia matrix, $\vec{\omega}$ is an eigenvector, and λ is an eigenvalue. The program **`Eigen.py`** in Listing 8.1 solves for the eigenvalues and vectors and produces output of the form

```
 I = [[ 0.66666667 -0.25       -0.25      ]
      [-0.25        0.66666667 -0.25      ]
      [-0.25       -0.25        0.66666667]]

 Eigenvalues = [ 0.91666667  0.16666667  0.91666667]

 Matrix of Eigenvectors = [[ 0.81649658 -0.40824829 -0.40824829]
                           [-0.57735027 -0.57735027 -0.57735027]
                           [ 0.43514263 -0.81589244  0.38074981]]

LHS - RHS = [ 1.11022302e-016   -5.55111512e-017   -1.11022302e-016]
```

Look at **`Eigen.py`** and notice how on line 5 we set up the array **`I`** with all the elements of the inertia tensor and on line 8 we solve for its eigenvalues and eigenvectors. On line 12 we extract the first eigenvector, and use it with the first eigenvalue to check if we have a solution by comparing the right and left sides of (8.29).

Listing 8.1 **Eigen.py** uses the linalg package and Visual's `array` to solve the eigenvalue problem. Note the matrix operations `eigenvectors` and `matrixmultiply`.

```
# Eigen.py  Solution of matrix eigenvalue problem

from numpy import*
from numpy.linalg import eig

I = array( [[2./3,-1./4,-1./4], [-1./4,2./3,-1./4], [-1./4,-1./4,2./3]] )
print(" I = ", I)

Es, evectors = eig(I)                          # Solves eigenvalue problem
print(' Eigenvalues = ', Es)
print(' ')
print(" Matrix of Eigenvectors =", evectors)

Vec = array([ evectors[0, 0], evectors[1, 0], evectors[2, 0] ] )

print(" ")
print(" A single eigenvector to test RHS vs LHS =", Vec)

LHS = dot(I, Vec)
RHS = dot(Vec, Es[0])

print(" ")
print('LHS - RHS = ', LHS-RHS)
print (" LHS = ", LHS)
print ('RHS =  ', RHS)
print("\n Enter and return any character to quit")
s = raw_input()
```

8.4.3 Exercises for Testing Matrix Calls

Before you direct the computer to go off crunching numbers on a million elements of some matrix, it's 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 take you only a short time to realize how hard it is to get the calling procedure perfectly right! Here are some exercises.

1. Find the numerical inverse of $\mathbf{A} = \begin{pmatrix} +4 & -2 & +1 \\ +3 & +6 & -4 \\ +2 & +1 & +8 \end{pmatrix}$.

   a. As a general procedure to always follow, applicable even if you do not know the analytic answer, check your inverse in both directions; that is, check that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, and note the number of decimal places to which this is true.

   b. Determine the number of decimal places of agreement there is between your numerical inverse and the analytic result: $\mathbf{A}^{-1} = \dfrac{1}{263}\begin{pmatrix} +52 & +17 & +2 \\ -32 & +30 & +19 \\ -9 & -8 & +30 \end{pmatrix}$.

2. Consider the same matrix $\mathbf{A}$ as before, now used to describe three simultaneous linear equations, $\mathbf{A}\vec{x} = \vec{b}$, or explicitly,

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

   Here the vector $\vec{b}$ on the RHS is assumed known, and the problem is to solve for the vector $\vec{x}$. Use an appropriate subroutine to solve these equations for the three different $\vec{x}$ vectors appropriate to these three different $\vec{b}$ values on the RHS:

$$\vec{b}_1 = \begin{pmatrix} +12 \\ -25 \\ +32 \end{pmatrix}, \quad \vec{b}_2 = \begin{pmatrix} +4 \\ -10 \\ +22 \end{pmatrix}, \quad \vec{b}_3 = \begin{pmatrix} +20 \\ -30 \\ +40 \end{pmatrix}.$$

   The solutions should be

$$\vec{x}_1 = \begin{pmatrix} +1 \\ -2 \\ +4 \end{pmatrix}, \quad \vec{x}_2 = \begin{pmatrix} +0.312 \\ -0.038 \\ +2.677 \end{pmatrix}, \quad \vec{x}_3 = \begin{pmatrix} +2.319 \\ -2.965 \\ +4.790 \end{pmatrix}.$$

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 α and β. Use a numerical eigenproblem solver to show that the eigenvalues and eigenvectors are the complex conjugates

$$\vec{x}_{1,2} = \begin{pmatrix} +1 \\ \mp i \end{pmatrix}, \quad \lambda_{1,2} = \alpha \mp i\beta.$$

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 combination of these eigenvectors is also an eigenvector.

   b. Verify that the eigenvector for $\lambda_1 = 5$ is proportional to

$$\vec{x}_1 = \frac{1}{\sqrt{6}}\begin{pmatrix} -1 \\ -2 \\ +1 \end{pmatrix}.$$

   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

$$\vec{x}_2 = \frac{1}{\sqrt{5}}\begin{pmatrix}-2\\+1\\+0\end{pmatrix}, \quad \vec{x}_3 = \frac{1}{\sqrt{10}}\begin{pmatrix}3\\0\\1\end{pmatrix}.$$

In this case it's not clear what your eigenproblem solver will give for the eigenvectors. Try to find a relationship between your computed eigenvectors with the eigenvalue -3 and these two linearly independent ones.

5. Your model of some physical system results in $N = 100$ coupled linear equations in N unknowns:

$$\begin{aligned} a_{11}y_1 + a_{12}y_2 + \cdots + a_{1N}y_N &= b_1, \\ a_{21}y_1 + a_{22}y_2 + \cdots + a_{2N}y_N &= b_2, \\ &\cdots \\ a_{N1}y_1 + a_{N2}y_2 + \cdots + a_{NN}y_N &= b_N. \end{aligned}$$

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 $\vec{b}$ as its first row:

$$[a_{ij}] = \mathbf{a} = \left[\frac{1}{i+j-1}\right] = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{100} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{101} \\ \ddots & & & & & \\ \frac{1}{100} & \frac{1}{101} & \cdots & & \cdots & \frac{1}{199} \end{pmatrix},$$

$$[b_i] = \vec{b} = \left[\frac{1}{i}\right] = \begin{pmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{3} \\ \ddots \\ \frac{1}{100} \end{pmatrix}.$$

Compare to the analytic solution

$$\begin{pmatrix} y_1 \\ y_2 \\ \ddots \\ y_N \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \ddots \\ 0 \end{pmatrix}.$$

8.4.4 Matrix Solution of the String Problem

We have now set up the solution to our problem of two masses on a string and have the matrix tools needed to solve it. Your **problem** is 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 (e.g., 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 similar in magnitude to the weights of the spheres. Our solution is given in `NewtonNDanimate.py`, which shows graphically the

step-by-step search for solution.

Listing 8.2 The code **NewtonNDanimate.py** that shows the step-by-step search for solution of the two-mass-on-a-string problem via a Newton–Raphson search.

```
# NewtonNDanimate.py  MultiDimension Newton Search

from numpy.linalg import solve
from visual.graph import *

scene = display(x=0,y=0,width=500,height=500,
                title='String and masses configuration')
tempe = curve(x=range(0,500),color=color.black)

n = 9
eps = 1e-6
deriv = zeros( (n, n), float)
f = zeros( (n), float)
x = array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1., 1., 1.])

def plotconfig():
    for obj in scene.objects:
        obj.visible=0           # to erase the previous configuration
    L1 = 3.0
    L2 = 4.0
    L3 = 4.0
    xa = L1*x[3]                # L1*cos(th1)
    ya = L1*x[0]                # L1 sin(th1)
    xb = xa+L2*x[4]             # L1*cos(th1)+L2*cos(th2)
    yb = ya+L2*x[1]             # L1*sin(th1)+L2*sen(th2)
    xc = xb+L3*x[5]             # L1*cos(th1)+L2*cos(th2)+L3*cos(th3)
    yc = yb-L3*x[2]             # L1*sin(th1)+L2*sen(th2)-L3*sin(th3)
    mx = 100.0                  # for linear coordinate transformation
    bx = -500.0                 # from 0=< x =<10
    my = -100.0                 # to     -500 =<x_window=>500
    by = 400.0                  # same transformation for y
    xap = mx*xa+bx              # to keep aspect ratio
    yap = my*ya+by
    ball1 = sphere(pos=(xap,yap), color=color.cyan,radius=15)
    xbp = mx*xb+bx
    ybp = my*yb+by
    ball2 = sphere(pos=(xbp,ybp), color=color.cyan,radius=25)
    xcp = mx*xc+bx
    ycp = my*yc+by
    x0 = mx*0+bx
    y0 = my*0+by
    line1 = curve(pos=[(x0,y0),(xap,yap)], color=color.yellow,radius=4)
    line2 = curve(pos=[(xap,yap),(xbp,ybp)], color=color.yellow,radius=4)
    line3 = curve(pos=[(xbp,ybp),(xcp,ycp)], color=color.yellow,radius=4)
    topline = curve(pos=[(x0,y0),(xcp,ycp)], color=color.red,radius=4)

def F(x, f):                    # Define F function
    f[0] = 3*x[3]  +  4*x[4]  +  4*x[5]  -  8.0
    f[1] = 3*x[0]  +  4*x[1]  -  4*x[2]
    f[2] = x[6]*x[0]  -  x[7]*x[1]  -  10.0
    f[3] = x[6]*x[3]  -  x[7]*x[4]
    f[4] = x[7]*x[1]  +  x[8]*x[2]  -  20.0
    f[5] = x[7]*x[4]  -  x[8]*x[5]
    f[6] = pow(x[0], 2)  +  pow(x[3], 2)  -  1.0
    f[7] = pow(x[1], 2)  +  pow(x[4], 2)  -  1.0
    f[8] = pow(x[2], 2)  +  pow(x[5], 2)  -  1.0

def dFi_dXj(x, deriv, n):       # Define derivative function
    h = 1e-4
    for j in range(0, n):
         temp = x[j]
         x[j] = x[j] +  h/2.
         F(x, f)
         for i in range(0, n):  deriv[i, j] = f[i]
         x[j] = temp

    for j in range(0, n):
         temp = x[j]
         x[j] = x[j] - h/2.
         F(x, f)
         for i in range(0, n): deriv[i, j] = (deriv[i, j] - f[i])/h
         x[j] = temp

for it in range(1, 100):
      rate(1)                   # 1 second between graphs
      F(x, f)
      dFi_dXj(x, deriv, n)
```

```
        B = array([[-f[0]], [-f[1]], [-f[2]], [-f[3]], [-f[4]], [-f[5]],\
        [-f[6]], [-f[7]], [-f[8]]])
        sol = solve(deriv, B)
        dx = take(sol, (0, ), 1)       # take the first column of matrix sol
        for i in range(0, n):
          x[i]  = x[i]  +  dx[i]
        plotconfig()
        errX = errF = errXi = 0.0

        for i in range(0, n):
          if ( x[i] !=  0.): errXi = abs(dx[i]/x[i])
          else:  errXi = abs(dx[i])
          if ( errXi > errX): errX = errXi
          if ( abs(f[i]) > errF ):  errF = abs(f[i])
          if ( (errX <=  eps) and (errF <=  eps) ): break

print('Number of iterations = ', it)
print('Solution:')
for i in range(0, n):
        print('x[', i, '] =  ', x[i])
```

8.4.5 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 Figure 8.1, you can observe that for some values of the weights and lengths, θ_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

$$f_7 = x_4 - \sqrt{1 - x_1^2}, \quad f_8 = x_5 - \sqrt{1 - x_2^2}, \quad f_9 = x_6 - \sqrt{1 - x_3^2}. \tag{8.30}$$

   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 is larger.

8.5 UNIT II. DATA FITTING

8.5.1 Fitting an Experimental Spectrum (Problem)

Data fitting is an art worthy of serious study by all scientists. In this unit we just scratch the surface by examining how to interpolate within a table of numbers and how to do a least-squares fit to data. We also show how to go about making a least-squares fit to nonlinear functions using some of the search techniques and subroutine libraries we have already discussed.

Problem: The cross sections measured for the resonant scattering of a neutron from a nucleus are given in Table 8.1 along with the measurement number (index), the energy, and the experimental error. Your **problem** is to determine values for the cross sections at energy values lying between those in the table.

You can solve this **problem** in a number of ways. The simplest is to numerically *interpolate* between the values of the experimental $f(E_i)$ given in Table 8.1. This is direct and easy but does not account for there being experimental noise in the data. A more appropriate

Table 8.1 Experimental values for a scattering cross section ($f(E)$ in the theory), each with absolute error $\pm\sigma_i$), as a function of energy (x_i in the theory).

i	1	2	3	4	5	6	7	8	9
E_i (MeV)	0	25	50	75	100	125	150	175	200
$g(E_i)$ (mb)	10.6	16.0	45.0	83.5	52.8	19.9	10.8	8.25	4.7
Error (mb)	9.34	17.9	41.5	85.5	51.5	21.5	10.8	6.29	4.14

way to solve this problem (discussed in §8.7) is to find the *best fit* of a theoretical function to the data. We start with what we believe to be the "correct" theoretical description of the data,

$$f(E) = \frac{f_r}{(E - E_r)^2 + \Gamma^2/4}, \tag{8.31}$$

where f_r, E_r, and Γ are unknown parameters. We then adjust the parameters to obtain the best fit. 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 [B&R 02].

These two 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 like Table 8.1. 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 show nonphysical behavior (such as large oscillations) between the data points. The rule of thumb is that if you must interpolate, keep it local and view global interpolations with a critical eye.

8.5.2 Lagrange Interpolation (Method)

Consider Table 8.1 as ordered data that we wish to interpolate. We call the independent variable x and its tabulated values $x_i (i = 1, 2, \ldots)$, 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 $(n-1)$-degree polynomial in each interval i:

$$g_i(x) \simeq a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}, \quad (x \simeq x_i). \tag{8.32}$$

Because our fit is local, we do not assume that one $g(x)$ can fit all the data in the table but instead 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, multiple polynomials are used to span the entire table. If some care is taken, the set of polynomials so obtained will behave well enough to be used in further calculations without introducing much unwanted noise or discontinuities.

The classic interpolation formula was created by Lagrange. He figured out a closed-form one that directly fits the $(n-1)$-order polynomial (8.32) to n values of the function $g(x)$ evaluated at the points x_i. The formula is written as the sum of polynomials:

$$g(x) \simeq g_1 \lambda_1(x) + g_2 \lambda_2(x) + \cdots + g_n \lambda_n(x), \tag{8.33}$$

$$\lambda_i(x) = \prod_{j(\neq i)=1}^{n} \frac{x - x_j}{x_i - x_j} = \frac{x - x_1}{x_i - x_1} \frac{x - x_2}{x_i - x_2} \cdots \frac{x - x_n}{x_i - x_n}. \tag{8.34}$$

For three points, (8.33) 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:

$$x_{1-4} = (0, 1, 2, 4) \quad g_{1-4} = (-12, -12, -24, -60), \tag{8.35}$$

$$g(x) = \frac{(x-1)(x-2)(x-4)}{(0-1)(0-2)(0-4)}(-12) + \frac{x(x-2)(x-4)}{(1-0)(1-2)(1-4)}(-12)$$

$$+ \frac{x(x-1)(x-4)}{(2-0)(2-1)(2-4)}(-24) + \frac{x(x-1)(x-2)}{(4-0)(4-1)(4-2)}(-60),$$

$$\Rightarrow \quad g(x) = x^3 - 9x^2 + 8x - 12. \tag{8.36}$$

As a check we see that

$$g(4) = 4^3 - 9(4^2) + 32 - 12 = -60, \qquad g(0.5) = -10.125. \tag{8.37}$$

If the data contain little noise, this polynomial can be used with some confidence within the range of the data, but with risk beyond the range of the 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*

$$R_n \simeq \frac{(x-x_1)(x-x_2)\cdots(x-x_n)}{n!} g^{(n)}(\zeta), \tag{8.38}$$

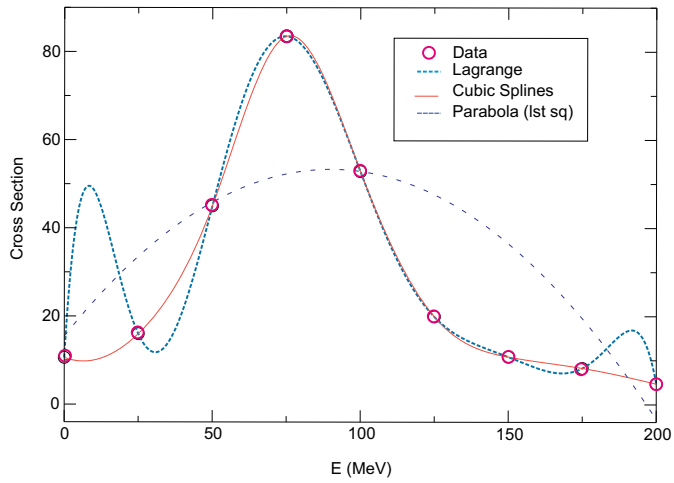
where ζ lies somewhere in the interpolation interval but is otherwise undetermined. This shows that if significant high derivatives exist in $g(x)$, then it cannot be approximated well by a polynomial. In particular, if $g(x)$ is a table of experimental data, it is likely to contain noise, and then it is a bad idea to fit a curve through all the data points.

8.5.3 Lagrange Implementation, Assessment

Consider the experimental neutron scattering data in Table 8.1. The expected theoretical functional form that describes these data is (8.31), and our empirical fits to these data are shown in Figure 8.3.

1. Write a subroutine to perform an n-point Lagrange interpolation using (8.33). Treat n as an arbitrary input parameter. (You can also do this exercise with the spline fits discussed in § 8.5.5.)
2. Use the Lagrange interpolation formula to fit the entire experimental spectrum with one polynomial. (This means that you must 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 Γ (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-sectional data in 5-MeV steps using three-point Lagrange interpolation. (Note that the end intervals may be

Figure 8.3 Three fits to cross-section data. *Short dashed line:* Lagrange interpolation using an eight-degree polynomial that passes through all the data points but has nonphysical oscillations between points; *solid line:* cubic splines fit (smooth but not necessarily good science); *dashed line:* Least-squares parabola fit (clearly not a good theory). The best approach is to do a least-squares fit of the correct theoretical function, in this case the Breit–Wigner formula (8.31).



special cases.)

This example shows how easy it is to go wrong with a high-degree-polynomial fit. Although the polynomial is guaranteed to 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 §8.7.

8.5.4 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 Table 8.1. Compare your results to the theoretical Breit–Wigner shape (8.31).

8.5.5 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. (A 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 an $n = 4$ interpolation, known as *cubic splines*, often leads to surprisingly eye-pleasing fits. In this approach (Figure 8.3), cubic polynomials are fit to the function in each interval, with the additional constraint that the first and second derivatives of the polynomials 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 spline 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 of data 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 calculation, yet easy for computers and, not surprisingly, popular in both calculations and graphics. To illustrate, the smooth curves connecting points in most "draw" programs are usually splines, as is the solid curve in Figure 8.3.

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:

$$g(x) \simeq g_i(x), \quad \text{for } x_i \le x \le x_{i+1}, \tag{8.39}$$

$$g_i(x) = g_i + g_i'(x - x_i) + \frac{1}{2} g_i''(x - x_i)^2 + \frac{1}{6} g_i'''(x - x_i)^3. \tag{8.40}$$

This representation makes it clear that the coefficients in the polynomial equal the values of $g(x)$ and its first, second, and third derivatives at the tabulated points x_i. Derivatives beyond the third vanish for a cubic. The computational chore is to determine these derivatives in terms of the N tabulated g_i values. The matching of g_i at the *nodes* that connect one interval to the next provides the equations

$$g_i(x_{i+1}) = g_{i+1}(x_{i+1}), \quad i = 1, N - 1. \tag{8.41}$$

The matching of the first *and* second derivatives at each interval's boundaries provides the equations

$$g_{i-1}'(x_i) = g_i'(x_i), \quad g_{i-1}''(x_i) = g_i''(x_i). \tag{8.42}$$

The additional equations needed to determine all constants is obtained by matching the third derivatives at adjacent nodes. Values for the third derivatives are found by approximating them in terms of the second derivatives:

$$g_i''' \simeq \frac{g_{i+1}'' - g_i''}{x_{i+1} - x_i}. \tag{8.43}$$

As discussed in Chapter 7, "Differentiation & Searching," a *central-difference approximation* would be better than a forward-difference approximation, yet (8.43) keeps the equations simpler.

It is straightforward though complicated to solve for all the parameters in (8.40). We leave that to other reference sources [Thom 92, Pres 94]. 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 the 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 the boundaries: Knowing values is of course better than approximating values, but it requires the user to input information. If the values of g'' are not known, they can be approximated by applying a forward-difference approximation to the tabulated values:

$$g''(x) \simeq \frac{[g(x_3) - g(x_2)]/[x_3 - x_2] - [g(x_2) - g(x_1)]/[x_2 - x_1]}{[x_3 - x_1]/2}. \tag{8.44}$$

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 is possible; if you have the ability to calculate the function directly for arbitrary x, Gaussian quadrature may be preferable. We know that the spline fit to g in each interval is the cubic (8.40)

$$g(x) \simeq g_i + g_i'(x - x_i) + \frac{1}{2}g_i''(x - x_i)^2 + \frac{1}{6}g_i'''(x - x_i)^3. \tag{8.45}$$

It is easy to integrate this to obtain the integral of g for this interval and then to sum over all intervals:

$$\int_{x_i}^{x_{i+1}} g(x)\,dx \simeq \left(g_i x + \frac{1}{2}g_i' x_i^2 + \frac{1}{6}g_i'' x^3 + \frac{1}{24}g_i''' x^4\right)\Bigg|_{x_i}^{x_{i+1}}, \tag{8.46}$$

$$\int_{x_j}^{x_k} g(x)\,dx = \sum_{i=j}^{k}\left(g_i x + \frac{1}{2}g_i' x_i^2 + \frac{1}{6}g_i'' x^3 + \frac{1}{24}g_i''' x^4\right)\Bigg|_{x_i}^{x_{i+1}}. \tag{8.47}$$

Making the intervals smaller does not necessarily increase precision, as subtractive cancelations in (8.46) may get large.

8.5.6 Spline Fit of Cross Section (Implementation)

Applet Fitting a series of cubics to data is a little complicated to program 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 [Pres 94] to produce the **`SplineInteract.py`** program shown in Listing 8.3 (there is also the applet). Your **problem** now is to carry out the assessment in § 8.5.3 using cubic spline interpolation rather than Lagrange interpolation.

8.6 FITTING EXPONENTIAL DECAY (PROBLEM)

Figure 8.4 presents actual experimental data on the number of decays ΔN of the π meson as a function of time [Stez 73]. Notice that the time has been "binned" into $\Delta t = 10$-ns intervals

and that the smooth curve is the theoretical exponential decay expected for very large numbers. Your **problem** is to deduce the lifetime τ of the π meson from these data (the tabulated lifetime of the pion is 2.6×10^{-8} s).

8.6.1 Theory to Fit

Applet Assume that we start with N_0 particles at time $t = 0$ that can decay to other particles.[4] If we wait a short time Δt, then a small number ΔN 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 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 ΔN in a time interval Δt is proportional to the number of particles $N(t)$ present at that time and to the time interval

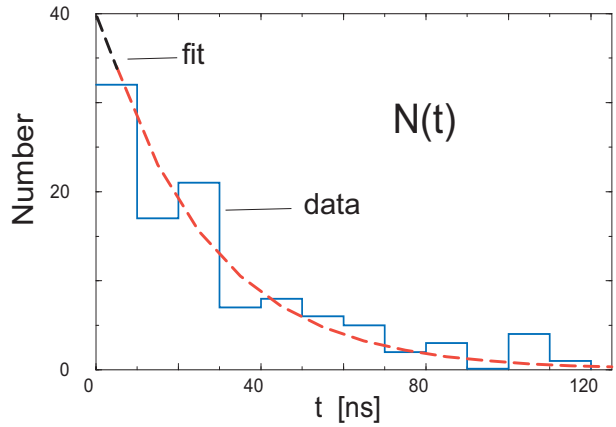
$$\Delta N(t) = -\frac{1}{\tau} N(t) \Delta t \quad \Rightarrow \quad \frac{\Delta N(t)}{\Delta t} = -\lambda N(t). \tag{8.48}$$

Listing 8.3 **SplineInteract.py** performs a cubic spline fit to data and permits interactive control. The arrays `x[]` and `y[]` are the data to fit, and the values of the fit at `Nfit` points are output.

```
# SplineInteract.py  Spline fit with slide to control number of points

from visual import *;                    from visual.graph import *;
from visual.graph import gdisplay, gcurve
from visual.controls import slider, controls, toggle

x = array([0., 0.12, 0.25, 0.37, 0.5, 0.62, 0.75, 0.87, 0.99])    # input
y = array([10.6, 16.0, 45.0, 83.5, 52.8, 19.9, 10.8, 8.25, 4.7])
n = 9;  np = 15

# Initialize
y2 = zeros( (n), float); u = zeros( (n), float)
graph1 = gdisplay(x=0,y=0,width=500, height=500,
                  title='Spline Fit', xtitle='x', ytitle='y')
funct1 = gdots(color = color.yellow)
funct2 = gdots(color = color.red)
graph1.visible = 0

def update():                                          # Nfit = 30 = output
    Nfit = int(control.value)
    for i in range(0, n):                               # Spread out points
        funct1.plot(pos = (x[i], y[i]) )
        funct1.plot(pos = (1.01*x[i], 1.01*y[i]) )
        funct1.plot(pos = (.99*x[i], .99*y[i]) )
        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])
    if (yp1 > 0.99e30):  y2[0] = 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)
    for i in range(1, n - 1):                                  # Decomp loop
        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
    if (ypn > 0.99e30):  qn = un = 0.                    # Test for natural
    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 in range(n - 2, 1, - 1):
        y2[k] = y2[k]*y2[k + 1] + u[k]
    for i in range(1, Nfit + 2):                                # Begin fit
```

[4] Spontaneous decay is discussed further and simulated in § 5.5.

Figure 8.4 A reproduction of the experimental measurement of [Stez 73] giving the number of decays of a π meson as a function of time since its creation. Measurements were made during time intervals (box sizes) of 10-ns width. The dashed curve is the result of a linear least-square fit to the $\log N(t)$.




```
        xout = x[0] + (x[n − 1] − x[0])*(i − 1)/(Nfit)
        klo = 0;     khi = n − 1                          # Bisection algor
        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
        funct2.plot(pos = (xout, yout) )
c = controls(x=300,y=0,width=200,height=200)          # Control via slider
control = slider(pos=(−50,50,0), min = 2, max = 100, action = update)
toggle(pos = (0, 35, − 5), text1 = "Number of points", height = 0)
control.value = 2
update()

while 1:
    c.interact()
    rate(50)                                          # update < 10/sec
    funct2.visible = 0
```

Applet Here $\tau = 1/\lambda$ is the *lifetime* of the particle, with λ the rate parameter. The actual decay *rate* is given by the second equation in (8.48). If the number of decays ΔN is very small compared to the number of particles N, and if we look at vanishingly small time intervals, then the difference equation (8.48) becomes the differential equation

$$\frac{dN(t)}{dt} \simeq -\lambda N(t) = \frac{1}{\tau} N(t). \tag{8.49}$$

This differential equation has an exponential solution for the number as well as for the decay rate:

$$N(t) = N_0 e^{-t/\tau}, \quad \frac{dN(t)}{dt} = -\frac{N_0}{\tau} e^{-t/\tau} = \frac{dN(0)}{dt}\, e^{-t/\tau}. \tag{8.50}$$

Equation (8.50) is the theoretical formula we wish to "fit" to the data in Figure 8.4. The output of such a fit is a "best value" for the lifetime τ.

8.7 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 [B&R 02, Pres 94, M&W 65, Thom 92]. However, we will emphasize three 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. If the theory is not an appropriate one for the data (e.g., the parabola in Figure 8.3), then its best fit to the data may not be a good fit at all. This is good, for it indicates that this is not the right theory.
3. 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 subroutine libraries. However, in §8.7.6 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:

$$(x_i, y_i \pm \sigma_i), \quad i = 1, N_D, \tag{8.51}$$

where $\pm\sigma_i$ is the uncertainty in the ith value of y. (For simplicity we assume that all the errors σ_i occur in the dependent variable, although this is hardly ever true [Thom 92]). 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 a *theory* or a *model*) can describe these data. Alternatively, if the theory contains some parameters or constants, our goal can be viewed as determining the 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 $\{a_1, a_2, \ldots, a_{M_P}\}$. 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 (8.50), the parameters are the lifetime τ and the initial decay rate $dN(0)/dt$. We indicate this as

$$g(x) = g(x; \{a_1, a_2, \ldots, a_{M_P}\}) = g(x; \{a_m\}). \tag{8.52}$$

We use the chi-square (χ^2) measure as a gauge of how well a theoretical function g reproduces data:

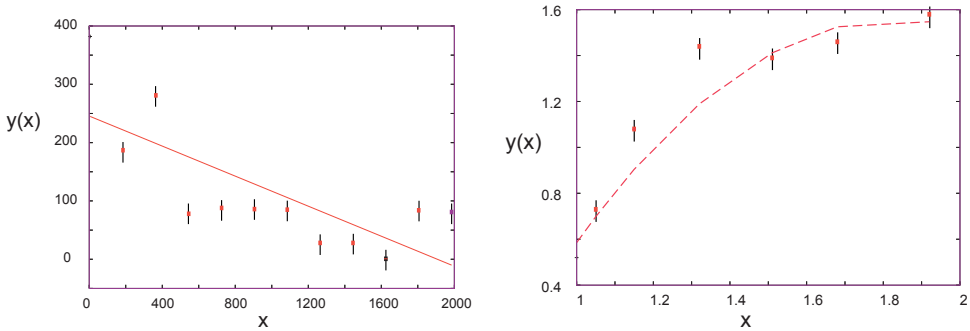
$$\chi^2 \stackrel{\text{def}}{=} \sum_{i=1}^{N_D} \left(\frac{y_i - g(x_i; \{a_m\})}{\sigma_i} \right)^2, \tag{8.53}$$

where the sum is over the N_D experimental points $(x_i, y_i \pm \sigma_i)$. The definition (8.53) is such that smaller values of χ^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[5] contribute less to χ^2.

Least-squares fitting refers to adjusting the parameters in the theory until a minimum in χ^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

[5] 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.

Figure 8.5 *Left:* A linear least-squares best fit of a straight line to data. The deviation of theory from experiment is greater than would be expected from statistics, which means that a straight line is not a good theory to describe these data. *Right:* A linear least-squares best fit of a parabola to different data. Here we see that the fit misses approximately one-third of the points, as expected from the statistics for a good fit. Although error bars are not shown, in both fits the experimental error is about 10%.



χ^2 an extremum are found by solving the M_P equations:

$$\frac{\partial \chi^2}{\partial a_m} = 0, \quad \Rightarrow \quad \sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_m} = 0, \quad (m = 1, M_P). \tag{8.54}$$

More usually, the function $g(x; \{a_m\})$ has a sufficiently complicated dependence on the a_m values for (8.54) 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 §8.7.6. To be safe, when such a search is completed, you need to check that the minimum χ^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 χ^2.

8.7.1 Least-Squares Fitting: Theory and Implementation

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 [B&R 02]. You know that your fit is good if the value of χ^2 calculated via the definition (8.53) is approximately equal to the number of degrees of freedom $\chi^2 \simeq N_D - M_P$, where N_D is the number of data points and M_P is the number of parameters in the theoretical function. If your χ^2 is much less than $N_D - M_P$, it doesn't mean that you have a "great" theory or a really precise measurement; instead, you probably have too many parameters or have assigned errors (σ_i values) that are too large. In fact, too small a χ^2 may indicate that you are fitting the random scatter in the data rather than missing approximately one-third of the error bars, as expected for a normal distribution. If your χ^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 that are not random.

The M_P simultaneous equations (8.54) can be simplified considerably if the functions $g(x; \{a_m\})$ depend *linearly* on the parameter values a_i, e.g.,

$$g\left(x; \{a_1, a_2\}\right) = a_1 + a_2 x. \tag{8.55}$$

In this case (also known as *linear regression* and shown on the left in Figure 8.5) 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, 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,

$$\frac{\partial g(x_i)}{\partial a_1} = 1, \quad \frac{\partial g(x_i)}{\partial a_2} = x_i, \tag{8.56}$$

and after substitution, the χ^2 minimization equations (8.54) can be solved [Pres 94]:

$$a_1 = \frac{S_{xx}S_y - S_x S_{xy}}{\Delta}, \qquad a_2 = \frac{S S_{xy} - S_x S_y}{\Delta}, \tag{8.57}$$

$$S = \sum_{i=1}^{N_D} \frac{1}{\sigma_i^2}, \qquad S_x = \sum_{i=1}^{N_D} \frac{x_i}{\sigma_i^2}, \quad S_y = \sum_{i=1}^{N_D} \frac{y_i}{\sigma_i^2}, \tag{8.58}$$

$$S_{xx} = \sum_{i=1}^{N_D} \frac{x_i^2}{\sigma_i^2}, \qquad S_{xy} = \sum_{i=1}^{N_D} \frac{x_i y_i}{\sigma_i^2}, \quad \Delta = S S_{xx} - S_x^2. \tag{8.59}$$

Statistics also gives you an expression for the *variance* or uncertainty in the deduced parameters:

$$\sigma_{a_1}^2 = \frac{S_{xx}}{\Delta}, \quad \sigma_{a_2}^2 = \frac{S}{\Delta}. \tag{8.60}$$

This is a measure of the uncertainties in the values of the fitted parameters arising from the uncertainties σ_i in the measured y_i values. A measure of the dependence of the parameters on each other is given by the *correlation coefficient*:

$$\rho(a_1, a_2) = \frac{\text{cov}(a_1, a_2)}{\sigma_{a_1}\sigma_{a_2}}, \quad \text{cov}(a_1, a_2) = \frac{-S_x}{\Delta}. \tag{8.61}$$

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 $-1 \le \rho \le 1$, with a positive ρ indicating that the errors in a_1 and a_2 are likely to have the same sign, and a negative ρ indicating opposite signs.

The preceding analytic solutions for the parameters are of the form found in statistics books but are not optimal for numerical calculations because subtractive cancelation can make the answers unstable. As discussed in Chapter 2, "Errors & Uncertainties in Computations," a rearrangement of the equations can decrease this type of error. For example, [Thom 92] gives improved expressions that measure the data relative to their averages:

$$a_1 = \overline{y} - a_2\overline{x}, \quad a_2 = \frac{S_{xy}}{S_{xx}}, \quad \overline{x} = \frac{1}{N}\sum_{i=1}^{N_d} x_i, \quad \overline{y} = \frac{1}{N}\sum_{i=1}^{N_d} y_i$$

$$S_{xy} = \sum_{i=1}^{N_d} \frac{(x_i - \overline{x})(y_i - \overline{y})}{\sigma_i^2}, \quad S_{xx} = \sum_{i=1}^{N_d} \frac{(x_i - \overline{x})^2}{\sigma_i^2}. \tag{8.62}$$

In **`Fit.py`** in Listing 8.4 we give a program that fits a parabola to some data. You can use it as a model for fitting a line to data, although you can use our closed-form expressions for a straight-line fit. In **`Fit.py`** in the instructor's manual we give a program for fitting to the decay data.

8.7.2 Exponential Decay Fit Assessment

Fit the exponential decay law (8.50) to the data in Figure 8.4. This means finding values for τ and $\Delta N(0)/\Delta t$ that provide 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 Figure 8.4. Because time was measured in bins, t_i should correspond to the middle of a bin.
2. Add an estimate of the error σ_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 a plot of $\ln|dN/dt|$ *versus* t to be 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 Δt as the independent variable, we can use our linear fit results. Plot $\ln|\Delta N/\Delta t|$ *versus* Δt.
4. Make a least-squares fit of a straight line to your data and use it to determine the lifetime τ of the π meson. Compare your deduction to the tabulated lifetime of 2.6×10^{-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?

8.7.3 Exercise: Fitting Heat Flow

The table below gives the temperature T along a metal rod whose ends are kept at a fixed constant temperature. 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 the data to verify the appropriateness of a linear relation
$$T(x) \simeq a + bx. \tag{8.63}$$
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 σ error band (that's what is expected for a normal distribution of errors).
5. Use your computed variance to determine the χ^2 of the fit. Comment on the value obtained.
6. Determine the variances σ_a and σ_b and check whether it makes sense to use them as the errors in the deduced values for a and b.

8.7.4 Linear Quadratic Fit (Extension)

As indicated earlier, as long as the function being fitted depends *linearly* on the unknown parameters a_i, the condition of minimum χ^2 leads to a set of simultaneous linear equations for the a's that can be solved on the computer using matrix techniques. To illustrate, suppose we want to fit the quadratic polynomial

$$g(x) = a_1 + a_2 x + a_3 x^2 \tag{8.64}$$

to the experimental measurements $(x_i, y_i, i = 1, N_D)$ (Figure 8.5 right). Because this $g(x)$ is linear in all the parameters a_i, we can still make a linear fit even though x is raised to the second power. [However, if we tried to a fit a function of the form $g(x) = (a_1 + a_2 x)\,\exp(-a_3\, x)$ to the data, then we would not be able to make a linear fit because one of the a's appears in the exponent.]

The best fit of this quadratic to the data is obtained by applying the minimum χ^2 condition (8.54) for $M_p = 3$ parameters and N_D (still arbitrary) data points. A solution represents the maximum likelihood that the deduced parameters provide a correct description of the data for the theoretical function $g(x)$. Equation (8.54) leads to the three simultaneous equations for a_1, a_2, and a_3:

$$\sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_1} = 0, \qquad \frac{\partial g}{\partial a_1} = 1, \tag{8.65}$$

$$\sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_2} = 0, \qquad \frac{\partial g}{\partial a_2} = x, \tag{8.66}$$

$$\sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_3} = 0, \qquad \frac{\partial g}{\partial a_3} = x^2. \tag{8.67}$$

Note: Because the derivatives are independent of the parameters (the a's), the a dependence arises only from the term in square brackets in the sums, and because that term has only a linear dependence on the a's, these equations are linear equations in the a's.

Exercise: Show that after some rearrangement, (8.65)–(8.67) can be written as

$$\begin{aligned} S a_1 + S_x a_2 + S_{xx} a_3 &= S_y, \\ S_x a_1 + S_{xx} a_2 + S_{xxx} a_3 &= S_{xy}, \\ S_{xx} a_1 + S_{xxx} a_2 + S_{xxxx} a_3 &= S_{xxy}. \end{aligned} \tag{8.68}$$

Here the definitions of the S's are simple extensions of those used in (8.57)–(8.59) and are programmed in `Fit.py` shown in Listing 8.4. After placing the three unknown parameters into a vector `x` and the known three RHS terms in (8.68) into a vector $\vec{b}$, these equations assume the matrix form:

$$\mathbf{A}\vec{x} = \vec{b}, \tag{8.69}$$

$$\mathbf{A} = \begin{bmatrix} S & S_x & S_{xx} \\ S_x & S_{xx} & S_{xxx} \\ S_{xx} & S_{xxx} & S_{xxxx} \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} S_y \\ S_{xy} \\ S_{xxy} \end{bmatrix}.$$

This is the exactly the matrix problem we solved in § 8.4 with the code `Fit.py` given in List-

ing 8.4. The solution for the parameter vector $\vec{a}$ is obtained by solving the matrix equations. Although for 3×3 matrices we can write out the solution in closed form, for larger problems the numerical solution requires matrix methods. ▌

Listing 8.4 **Fit.py** performs a least-squares fit of a parabola to data using the linalg package to solve the set of linear equations $\mathbf{S}\vec{a} = \vec{s}$.

```
# Fit.py     Linear least square fit; e.g. of matrix computation arrays

import pylab as p
from visual import *
from numpy import*
from numpy.linalg import inv
from numpy.linalg import solve

t = arange(1.0, 2.0, 0.1)                                  # x range curve
x = array([1., 1.1, 1.24, 1.35,  1.451, 1.5, 1.92])       # Given x values
y = array([0.52, 0.8, 0.7, 1.8, 2.9, 2.9, 3.6])           # Given y values
p.plot(x, y, 'bo' )                                      # Plot data in blue
sig = array([0.1, 0.1, 0.2, 0.3, 0.2, 0.1, 0.1])         # error bar lenghts
p.errorbar(x,y,sig)                                        # Plot error bars
p.title('Linear Least Square Fit')                             # Plot figure
p.xlabel( 'x' )                                                 # Label axes
p.ylabel( 'y' )
p.grid(True)                                                     # plot grid
Nd = 7
A = zeros( (3,3), float )                                        # Initialize
bvec = zeros( (3,1), float )
ss= sx = sxx = sy = sxxx = sxxxx = sxy = sxy = sxxy = 0.

for i in range(0, Nd):
        sig2 = sig[i] * sig[i]
        ss  += 1. / sig2;    sx   += x[i]/sig2;        sy    += y[i]/sig2
        rhl  = x[i] * x[i];  sxx  += rhl/sig2;   sxxy  += rhl * y[i]/sig2
        sxy += x[i]*y[i]/sig2; sxxx +=rhl*x[i]/sig2; sxxxx +=rhl*rhl/sig2

A    = array([ [ss,sx,sxx], [sx,sxx,sxxx], [sxx,sxxx,sxxxx] ])
bvec = array([sy, sxy, sxxy])

xvec = multiply(inv(A), bvec)                              # Invert matrix
Itest = multiply(A, inv(A))                              # Matrix multiply
print('\n x vector via inverse')
print(xvec, '\n')
print('A*inverse(A)')
print(Itest, '\n')

xvec = solve(A, bvec)                              # Solve via elimination
print('x Matrix via direct')
print(xvec, 'end= ')
print('FitParabola Final Results\n')
print('y(x) = a0 + a1 x + a2 x^2')                              # Desired fit
print('a0 = ', x[0])
print('a1 = ', x[1])
print('a2 = ', x[2], '\n')
print(' i   xi     yi    yfit   ')
for i in range(0, Nd):
    s = xvec[0] + xvec[1]*x[i] + xvec[2]*x[i]*x[i]
    print(" %d %5.3f  %5.3f  %8.7f \n"  %(i, x[i], y[i], s))
# red line is the fit, red dots the fits at y[i]
curve  = xvec[0] + xvec[1]*t + xvec[2]*t**2
points = xvec[0] + xvec[1]*x + xvec[2]*x**2
p.plot(t, curve,'r', x, points, 'ro')
p.show()
```

8.7.5 Linear Quadratic Fit Assessment

1. Fit the quadratic (8.64) to the following data sets [given as $(x_1, y_1), (x_2, y_2), \ldots$]. In each case indicate the values found for the a's, the number of degrees of freedom, *and* the value of χ^2.
   a. $(0, 1)$
   b. $(0, 1), (1, 3)$

c. $(0,1),(1,3),(2,7)$
d. $(0,1),(1,3),(2,7),(3,15)$

2. Find a fit to the last set of data to the function

$$y = Ae^{-bx^2}. \tag{8.70}$$

Hint: A judicious change of variables will permit you to convert this to a linear fit. Does a minimum χ^2 still have meaning here?

8.7.6 Nonlinear Fit of the Breit–Wigner Formula to a Cross Section

Problem: Remember how we started Unit II of this chapter by interpolating the values in Table 8.1, which gave the experimental cross section $\S igma$ 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 (8.31):

$$f(E) = \frac{f_r}{(E - E_r)^2 + \Gamma^2/4}. \tag{8.71}$$

Your **problem** is to determine what values for the parameters E_r, f_r, and Γ in (8.71) provide the best fit to the data in Table 8.1.

Because (8.71) is not a linear function of the parameters (E_r, $\S igma_0$,Γ), the three equations that result from minimizing χ^2 are not linear equations and so cannot be solved by the techniques of *linear* algebra (matrix methods). However, in our study of the masses 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. We now use this same combination of fitting, trial-and-error searching, and matrix algebra to conduct a nonlinear least-squares fit of (8.71) to the data in Table 8.1.

Recollect 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 (8.54) to solve

$$\sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_m} = 0, \quad (m = 1, M_P). \tag{8.72}$$

To find the form of these equations appropriate to our problem, we rewrite our theory function (8.71) in the notation of (8.72):

$$a_1 = f_r, \quad a_2 = E_R, \quad a_3 = \Gamma^2/4, \quad x = E, \tag{8.73}$$

$$\Rightarrow \quad g(x) = \frac{a_1}{(x - a_2)^2 + a_3}. \tag{8.74}$$

The three derivatives required in (8.72) 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 (8.72) yields three simultaneous equations in a_1, a_2, and a_3 that we need to solve in order to fit the $N_D = 9$ data points (x_i, y_i)

in Table 8.1:

$$\sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{(x_i - a_2)^2 + a_3} = 0, \quad \sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{[(x_i - a_2)^2 + a_3]^2} = 0,$$

$$\sum_{i=1}^{9} \frac{\{y_i - g(x_i, a)\}\,(x_i - a_2)}{[(x_i - a_2)^2 + a_3]^2} = 0. \tag{8.75}$$

Even without the substitution of (8.71) for $g(x, a)$, it is clear that these three equations depend on the a's in a nonlinear fashion. That's okay because in §8.2.2 we derived the N-dimensional Newton–Raphson search for the roots of

$$f_i(a_1, a_2, \ldots, a_N) = 0, \quad i = 1, N, \tag{8.76}$$

where we have made the change of variable $y_i \to a_i$ for the present problem. We use that same formalism here for the $N = 3$ equations (8.75) by writing them as

$$f_1(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{(x_i - a_2)^2 + a_3} = 0, \tag{8.77}$$

$$f_2(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{\{y_i - g(x_i, a)\}\,(x_i - a_2)}{[(x_i - a_2)^2 + a_3]^2} = 0, \tag{8.78}$$

$$f_3(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{[(x_i - a_2)^2 + a_3]^2} = 0. \tag{8.79}$$

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 guesses for the a's can be extracted from a graph of the data. To obtain the nine derivatives of the three f's with respect to the three unknown a's, we use two nested loops over i and j, along with the forward-difference approximation for the derivative

$$\frac{\partial f_i}{\partial a_j} \simeq \frac{f_i(a_j + \Delta a_j) - f_i(a_j)}{\Delta a_j}, \tag{8.80}$$

where Δa_j corresponds to a small, say $\leq 1\%$, change in the parameter value.

Nonlinear Fit Implementation

Use the Newton–Raphson algorithm as outlined in §8.7.6 to conduct a nonlinear search for the best-fit parameters of the Breit–Wigner theory (8.71) to the data in Table 8.1. Compare the deduced values of (f_r, E_R, Γ) to that obtained by inspection of the graph. ▌

Chapter Nine

Differential Equation Applications

Part of the attraction of computational problem solving is that it is easy to solve almost every differential equation. Consequently, while most traditional (read "analytic") treatments of oscillations are limited to the small displacements about equilibrium where the restoring forces are linear, we eliminate those restrictions here and reveal some interesting nonlinear physics. In Unit I we look at oscillators that may be harmonic for certain parameter values but then become anharmonic. We start with simple systems that have analytic solutions, use them to test various differential-equation solvers, and then include time-dependent forces and investigate nonlinear resonances and beating.[1] *In Unit II we examine how a differential-equation solver may be combined with a search algorithm to solve the eigenvalue problem. In Unit III we investigate how to solve the simultaneous ordinary differential equations (ODEs) that arise in scattering, projectile motion, and planetary orbits.*

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Ordinary Diff Eqs (ODEs)	pdf	9.1	ODE Algorithms	pdf	9.5
ODE Lab	pdf	9.5.4	Quantum Eigenvalues	pdf	9.10

Applets Applet

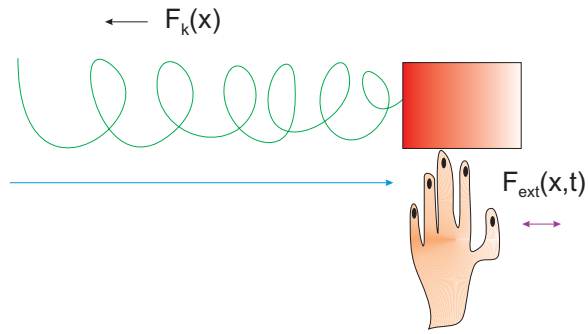
Name	*Sections*	*Name*	*Sections*
The Chaotic Pendulum	12.11–12.13	Hypersensitive Pendulums	12.11–12.13
Planetary Orbits	9.17	HearData: Sound Converter for Data	9.7
Chaotic Scattering	9.14	Relativistic Scattering	9.14
ABM Predictor Corrector	9.5.3	Visualizing with Sound	9.7
Photoelectric Effect	9.10-9.12		

9.1 UNIT I. FREE NONLINEAR OSCILLATIONS

Problem: In Figure 9.1 we show a mass m 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 the motion is constrained to one dimension.

[1] In Chapter 12, "Discrete & Continuous Nonlinear Dynamics," we make a related study of the realistic pendulum and its chaotic behavior. Some special properties of nonlinear equations are discussed in Chapter 19, "Solitons & Computational Fluid Dynamics."

Figure 9.1 A mass m (the block) attached to a spring with restoring force $F_k(x)$ driven by an external time-dependent driving force (the hand).



9.2 NONLINEAR OSCILLATORS (MODELS)

This is a problem in classical mechanics for which Newton's second law provides us with the equation of motion

$$F_k(x) + F_{\text{ext}}(x,t) = m\frac{d^2x}{dt^2}, \tag{9.1}$$

where $F_k(x)$ is the restoring force exerted by the spring and $F_{\text{ext}}(x,t)$ is the external force. Equation (9.1) is the differential equation we must solve for arbitrary forces. Because we are not told just how the spring departs 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 and also contains a perturbation that introduces a nonlinear term to the force for large x values:

$$V(x) \simeq \frac{1}{2}kx^2\left(1 - \frac{2}{3}\alpha x\right), \tag{9.2}$$

$$\Rightarrow \quad F_k(x) = -\frac{dV(x)}{dx} = -kx(1-\alpha x) = m\frac{d^2x}{dt^2}, \tag{9.3}$$

where we have omitted the time-dependent external force. Equation (9.3) 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 the curves on the left in Figure 9.2. As long as $x < 1/\alpha$, there will be a *restoring force* and the motion will be periodic (repeated exactly and indefinitely in time), even though it is harmonic (linear) only 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 will become repulsive and the mass will "roll" 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:

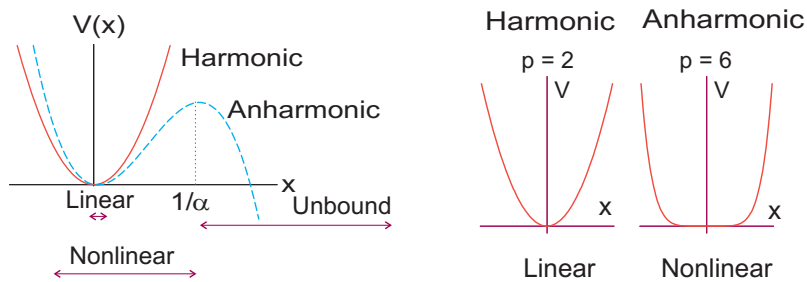
$$V(x) = \frac{1}{p}kx^p, \quad (p \text{ even}). \tag{9.4}$$

We require an even p to ensure that the force,

$$F_k(x) = -\frac{dV(x)}{dx} = -kx^{p-1}, \tag{9.5}$$

contains an odd power of p, which guarantees that it is a *restoring* force for positive or negative x values. We display some characteristics of this potential on the right in Figure 9.2. We see that $p = 2$ is the harmonic oscillator and that $p = 6$ is nearly a square well with the mass

Figure 9.2 *Left:* The potentials of an harmonic oscillator (solid curve) and of an anharmonic oscillator (dashed curve). If the amplitude becomes too large for the anharmonic oscillator, the motion becomes unbound. *Right:* The shapes of the potential energy function $V(x) \propto |x|^p$ for $p = 2$ and $p = 6$. The "linear" and "nonlinear" labels refer to the restoring force derived from these potentials.



moving almost freely until it hits the wall at $x \simeq \pm 1$. Regardless of the p value, the motion will be periodic, but it will be harmonic only for $p = 2$. Newton's law (9.1) gives the second-order ODE we need to solve:

$$F_{\text{ext}}(x,t) - kx^{p-1} = m\frac{d^2x}{dt^2}. \tag{9.6}$$

9.3 TYPES OF DIFFERENTIAL EQUATIONS (MATH)

The background material in this section is presented to avoid confusion about semantics. The well-versed reader may want to skim or skip it.

Order: A general form for a *first-order* differential equation is

$$\frac{dy}{dt} = f(t,y), \tag{9.7}$$

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

$$\frac{dy}{dt} = -3t^2y + t^9 + y^7, \tag{9.8}$$

this is still first order in the derivative. A general form for a *second-order* differential equation is

$$\frac{d^2y}{dt^2} + \lambda\frac{dy}{dt} = f\left(t, \frac{dy}{dt}, y\right). \tag{9.9}$$

The derivative function f on the RHS is arbitrary and may involve any power of the first derivative as well. To illustrate,

$$\frac{d^2y}{dt^2} + \lambda\frac{dy}{dt} = -3t^2\left(\frac{dy}{dt}\right)^4 + t^9y(t) \tag{9.10}$$

is a second-order differential equation, as is Newton's law (9.1).

In the differential equations (9.7) and (9.9), 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 often 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 (9.1) and (9.7) are *ordinary* dif-

ferential equations because they contain only *one* independent variable, in these cases t. In contrast, an equation such as the Schrödinger equation

$$i\frac{\partial\psi(\mathbf{x},t)}{\partial t} = -\frac{1}{2m}\left[\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} + \frac{\partial^2\psi}{\partial z^2}\right] + V(\mathbf{x})\psi(\mathbf{x},t) \tag{9.11}$$

(where we have set $\hbar = 1$) contains several independent variables, and this makes it a *partial differential equation* (PDE). The partial derivative symbol ∂ is used to indicate that the dependent variable ψ depends simultaneously on several independent variables. In the early parts of this book we limit ourselves to ordinary differential equations. In Chapters 17–19 we examine a variety of partial differential equations.

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,

$$\frac{dy}{dt} = g^3(t)y(t) \quad \text{(linear)}, \qquad \frac{dy}{dt} = \lambda y(t) - \lambda^2 y^2(t) \quad \text{(nonlinear)}. \tag{9.12}$$

An important property of linear equations is the *law of linear superposition* that 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 (9.12), then

$$y(t) = \alpha A(t) + \beta B(t) \tag{9.13}$$

is also a solution for arbitrary values of the constants α and β. In contrast, even if we were clever enough to guess that the solution of the nonlinear equation in (9.12) is

$$y(t) = \frac{a}{1 + be^{-\lambda t}} \tag{9.14}$$

(which you can verify by substitution), things would be amiss if we tried to obtain a more general solution by adding together two such solutions:

$$y_1(t) = \frac{a}{1 + be^{-\lambda t}} + \frac{a'}{1 + b'e^{-\lambda t}} \tag{9.15}$$

(which you can verify by substitution).

Initial and boundary conditions: The general solution of a first-order differential equation always contains one arbitrary constant. The 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 may be the position $y(t)$ at some time. For a second-order equation, the two initial conditions may be the position and velocity at some time. Regardless of how powerful the hardware and software that you employ, the mathematics remains valid, and so you must know the initial conditions in order to solve the problem uniquely.

In addition to the initial conditions, it is possible to further restrict the solutions of differential equations. One such way is by *boundary conditions* that constrain the solution to have fixed values at the boundaries of the solution space. Problems of this sort are called *eigenvalue problems*, and they are so demanding that solutions do not always exist, and even when they do exist, a trial-and-error search may be required to find them. In Unit II we discuss how to extend the techniques of the present unit to boundary-value problems.

9.4 DYNAMIC FORM FOR ODES (THEORY)

A standard form for ODEs, which has found use both in numerical analysis [Pres 94, Pres 00] and in classical dynamics [Schk 94, Tab 89, J&S 98], is to express ODEs of *any order* as N

simultaneous first-order ODEs:

$$\frac{dy^{(0)}}{dt} = f^{(0)}(t, y^{(i)}),$$

$$\frac{dy^{(1)}}{dt} = f^{(1)}(t, y^{(i)}), \tag{9.16}$$

$$\ddots \tag{9.17}$$

$$\frac{dy^{(N-1)}}{dt} = f^{(N-1)}(t, y^{(i)}), \tag{9.18}$$

where $y^{(i)}$ dependence in f is allowed but not any dependence on derivatives $dy^{(i)}/dt$. These equations can be expressed more compactly by use of the N-dimensional vectors (indicated here in **boldface**) $\mathbf{y}$ and $\mathbf{f}$:

$$d\mathbf{y}(t)/dt = \mathbf{f}(t, \mathbf{y}), \tag{9.19}$$

$$\mathbf{y} = \begin{pmatrix} y^{(0)}(t) \\ y^{(1)}(t) \\ \ddots \\ y^{(N-1)}(t) \end{pmatrix}, \quad \mathbf{f} = \begin{pmatrix} f^{(0)}(t, \mathbf{y}) \\ f^{(1)}(t, \mathbf{y}) \\ \ddots \\ f^{(N-1)}(t, \mathbf{y}) \end{pmatrix}.$$

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 (9.19) without having to worry about the individual components. To see how this works, let us convert Newton's law

$$\frac{d^2x}{dt^2} = \frac{1}{m} F\left(t, x, \frac{dx}{dt}\right) \tag{9.20}$$

to standard dynamic form. 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 $y^{(1)}$:

$$y^{(0)}(t) \stackrel{\text{def}}{=} x(t), \quad y^{(1)}(t) \stackrel{\text{def}}{=} \frac{dx}{dt} = \frac{dy^{(0)}(t)}{dt}. \tag{9.21}$$

The second-order ODE (9.20) now becomes two simultaneous first-order ODEs:

$$\frac{dy^{(0)}}{dt} = y^{(1)}(t), \quad \frac{dy^{(1)}}{dt} = \frac{1}{m} F(t, y^{(0)}, y^{(1)}). \tag{9.22}$$

This expresses the acceleration [the second derivative in (9.20)] as the first derivative of the velocity [$y^{(1)}$]. These equations are now in the standard form (9.19), with the derivative or force function $\mathbf{f}$ having the two components

$$f^{(0)} = y^{(1)}(t), \quad f^{(1)} = \frac{1}{m} F(t, y^{(0)}, y^{(1)}), \tag{9.23}$$

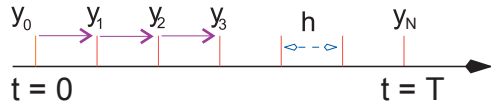
where F may be an explicit function of time as well as of position and velocity.

To be even more specific, applying these definitions to our spring problem (9.6), we obtain the coupled first-order equations

$$\frac{dy^{(0)}}{dt} = y^{(1)}(t), \quad \frac{dy^{(1)}}{dt} = \frac{1}{m}\left[F_{\text{ext}}(x, t) - k y^{(0)}(t)^{p-1}\right], \tag{9.24}$$

where $y^{(0)}(t)$ is the position of the mass at time t and $y^{(1)}(t)$ is its velocity. In the standard

Figure 9.3 A sequence of uniform steps of length h taken in solving a differential equation. The solution starts at time t = 0 and is stepped out (integrated) to t = T.



form, the components of the force function and the initial conditions are

$$f^{(0)}(t,\mathbf{y}) = y^{(1)}(t), \quad f^{(1)}(t,\mathbf{y}) = \frac{1}{m}\left[F_{\text{ext}}(x,t) - k(y^{(0)})^{p-1}\right],$$
$$y^{(0)}(0) = x_0, \qquad y^{(1)}(0) = v_0. \tag{9.25}$$

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 (9.1), into two *Hamiltonian* equations involving position and momentum:

$$\frac{dp_i}{dt} = F_i, \quad m\frac{dy_i}{dt} = p_i. \tag{9.26}$$

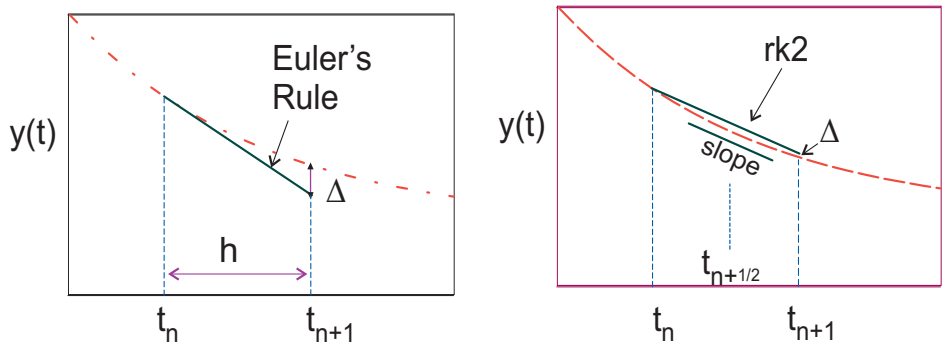
9.5 ODE ALGORITHMS

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 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 can solve the ODE for all t values by just continuing stepping to larger times one small h at a time (Figure 9.3).[2] Error is always a concern when integrating differential equations because derivatives require small differences, and small differences are prone to subtractive cancelations and round-off error accumulation. 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.

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 §9.5.2, adapt the step size by making h larger in regions where y varies slowly (this speeds up the integration and cuts down on round-off error) and making h smaller in regions where y varies rapidly.

[2] To avoid confusion, notice that $y^{(n)}$ is the nth 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.)

Figure 9.4 *Left:* Euler's algorithm for integration of a differential equation one step forward in time. This linear extrapolation with the slope evaluated at the initial point is seen to lead to an error Δ. *Right:* The `rk2` algorithm for integration of a differential equation uses a slope (bold line segment) evaluated at the interval's midpoint, and is seen to lead to a smaller error.



9.5.1 Euler's Rule

Euler's rule (Figure 9.4 left) is a simple algorithm for integrating the differential equation (9.7) by one step and is just the forward-difference algorithm for the derivative:

$$\frac{d\mathbf{y}(t)}{dt} \simeq \frac{\mathbf{y}(t_{n+1}) - \mathbf{y}(t_n)}{h} = \mathbf{f}(t, \mathbf{y}), \tag{9.27}$$

$$\Rightarrow \quad \mathbf{y}_{n+1} \simeq \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n), \tag{9.28}$$

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:

$$y_1^{(0)} = x_0 + v_0 h, \quad y_1^{(1)} = v_0 + h\frac{1}{m}\left[F_{\text{ext}}(t=0) + F_k(t=0)\right]. \tag{9.29}$$

Compare these to the projectile equations familiar from first-year physics,

$$x = x_0 + v_0 h + \frac{1}{2}ah^2, \quad v = v_0 + ah, \tag{9.30}$$

and we see that the acceleration does not contribute to the distance covered (no h^2 term), yet it does contribute to the velocity (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 round-off error, which may lead to instability.[3] Whereas we do not recommend Euler's algorithm for general use, it is commonly used to start some of the more precise algorithms.

9.5.2 Runge–Kutta Algorithm

Even though 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

[3] Instability is often a problem when you integrate a $y(t)$ that 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.

and capable of industrial-strength work. In spite of **rk4** being our recommended standard method, we derive the simpler **rk2** here and just give the results for **rk4**. The Runge–Kutta algorithms for integrating a differential equation are based upon the formal (exact) integral of our differential equation:

$$\frac{dy}{dt} = f(t,y) \quad \Rightarrow \quad y(t) = \int f(t,y)\,dt \tag{9.31}$$

$$\Rightarrow \quad y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t,y)\,dt. \tag{9.32}$$

To derive the second-order Runge–Kutta algorithm **rk2** (Figure 9.4 right and **rk2.py**), we expand $f(t,y)$ in a Taylor series about the *midpoint* of the integration interval and retain two terms:

$$f(t,y) \simeq f(t_{n+1/2}, y_{n+1/2}) \,+\, (t - t_{n+1/2})\,\frac{df}{dt}(t_{n+1/2}) + \mathcal{O}(h^2). \tag{9.33}$$

Because $(t - t_{n+1/2})$ to any odd power is equally positive and negative over the interval $t_n \le t \le t_{n+1}$, the integral of $(t - t_{n+1/2})$ in (9.32) vanishes and we obtain our algorithm:

$$\int_{t_n}^{t_{n+1}} f(t,y)\,dt \simeq f(t_{n+1/2},\ y_{n+1/2})h + \mathcal{O}(h^3), \tag{9.34}$$

$$\Rightarrow \quad y_{n+1} \simeq y_n + hf(t_{n+1/2},\ y_{n+1/2}) + \mathcal{O}(h^3) \quad \textbf{(rk2)}. \tag{9.35}$$

We see that while **rk2** contains the same number of terms as Euler's rule, it obtains a higher level of precision by taking advantage of the cancelation of the $\mathcal{O}(h)$ terms [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 middle of the time interval, $t = t_n + h/2$. And there's the rub, for we do not know the value of $y_{n+1/2}$ and cannot use this algorithm to determine it. The way out of this quandary is to use Euler's algorithm for $y_{n+1/2}$:

$$y_{n+1/2} \simeq y_n + \frac{1}{2}h\,\frac{dy}{dt} = y_n + \frac{1}{2}hf(t_n,\ y_n). \tag{9.36}$$

Putting the pieces all together gives the complete **rk2** algorithm:

$$\mathbf{y}_{n+1} \;\simeq\; \mathbf{y}_n + \mathbf{k}_2, \quad \textbf{(rk2)} \tag{9.37}$$

$$\mathbf{k}_2 = h\mathbf{f}\left(t_n + \tfrac{h}{2},\ \mathbf{y}_n + \tfrac{\mathbf{k}_1}{2}\right), \quad \mathbf{k}_1 = h\,\mathbf{f}(t_n,\ \mathbf{y}_n), \tag{9.38}$$

where we use boldface to indicate the vector nature of y and f. We see that the known derivative function $\mathbf{f}$ is evaluated at the ends and the 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 our spring problem:

$$y_1^{(0)} = y_0^{(0)} + hf^{(0)}\left(\frac{h}{2}, y_0^{(0)} + k_1\right) \simeq x_0 + h\left[v_0 + \frac{h}{2}F_k(0)\right],$$

$$y_1^{(1)} = y_0^{(1)} + hf^{(1)}\left[\frac{h}{2}, y_0 + \frac{h}{2}f(0,y_0)\right] \simeq v_0 + \frac{h}{m}\left[F_{\text{ext}}\left(\frac{h}{2}\right) + F_k\left(y_0^{(1)} + \frac{k_1}{2}\right)\right].$$

These equations say that the position $y^{(0)}$ changes because of the initial velocity and force, while the velocity changes because of 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 level of first-year physics.

The fourth-order Runge–Kutta method **rk4** (Listing 9.1) 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 step size h can be made larger. Explicitly, **rk4** requires the evaluation of four intermediate slopes, and these are approximated with the Euler algorithm to obtain[Pres 94]:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4), \tag{9.39}$$

$$\mathbf{k}_1 = h\mathbf{f}(t_n, \mathbf{y}_n), \qquad \mathbf{k}_2 = h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_1}{2}\right),$$

$$\mathbf{k}_3 = h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_2}{2}\right), \qquad \mathbf{k}_4 = h\mathbf{f}(t_n + h, \mathbf{y}_n + \mathbf{k}_3).$$

Listing 9.1 **rk4.py** solves an ODE with the 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.

```
# rk4.py 4th order Runge Kutta soltn of ODE

from visual.graph import *

#   Initialization
a = 0.;                         b = 10.;                    n = 100
ydumb = zeros((2),float);y = zeros((2),float); fReturn = zeros((2),float)
y[0] = 3. ;  y[1] = -5.;  t = a;   h = (b-a)/n;

def f( t, y, fReturn ):                     # function returns RHS, change
    fReturn[0] = y[1]                                          # 1st deriv
    fReturn[1] = -9*y[0]-2*y[1] + 9*sin(3.*t) # 2nd spring, friction, ext

def rk4Function(h, fReturn):                # Func for rk4; do not disturb
    k1 = zeros((2),float); k2 = zeros((2),float);  k3 = zeros((2),float);
    k4 = zeros((2), float)
    k1[0] = h*fReturn[0];  k1[1] = h*fReturn[1]          # Compute function
    for i in range(0, 2): 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 in range(0, 2):  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 in range(0, 2): ydumb[i] = y[i] + k3[i]
    f(t + h, ydumb, fReturn)
    k4[0] = h*fReturn[0];   k4[1] = h*fReturn[1]
    for i in range(0, 2): y[i] = y[i] + (k1[i]+2.*(k2[i]+k3[i])+k4[i])/6.

graph1 = gdisplay(x=0,y=0, width = 400, height = 400, title = 'RK4',
            xtitle = 't', ytitle = 'Y[0]',xmin=0,xmax=10,ymin=-2,ymax=3)
funct1 = gcurve(color = color.blue)
graph2 = gdisplay(x=400,y=0, width = 400, height = 400, title = 'RK4',
           xtitle = 't', ytitle = 'Y[1]',xmin=0,xmax=10,ymin=-25,ymax=18)
funct2 = gcurve(color = color.red)
funct1.plot(pos = (t, y[0]) )
funct2.plot(pos = (t, y[1]) )
while (t < b):                                                 # Time loop
    if ( (t + h) > b ): h = b - t                              # Last step
    f(t, y, fReturn)                  # Evaluate RHS's, return in fReturn
    rk4Function(h, fReturn)
    t = t + h
    rate(30)
    funct1.plot(pos = (t, y[0]))
    funct2.plot(pos = (t, y[1]))                          # End while loop
```

rk45: rk45.py in listing 9.2 is a variation of **rk4**, known as the Runge–Kutta–Fehlberg method or **rk45** [Math 02]. It 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.

Listing 9.2 **rk45.py** solves an ODE with the RHS given by the method `f()` using a fourth-order Runge–Kutta algorithm with adaptive step size.

```
# rk45.py          Adaptive step size Runge Kutta

from visual import *
from visual.graph import *

a = 0.; b = 10.                        # Error tolerance, endpoints
Tol = 1.0E-8

# Initialize
ydumb = zeros( (2), float)
y = zeros( (2), float)
fReturn = zeros( (2), float)
err = zeros( (2), float)
k1 = zeros( (2), float)
k2 = zeros( (2), float)
k3 = zeros( (2), float)
k4 = zeros( (2), float)
k5 = zeros( (2), float)
k6 = zeros( (2), float)
n = 20

y[0] = 1. ;   y[1] = 0.          # Initialize

h = (b - a)/n; hmin = h/64;   hmax = h*64     # Min and max step sizes
t = a;   j = 0
flops = 0;   Eexact = 0. ;  error = 0.
sum = 0.

def f( t, y, fReturn ):                # Return RHS's force function
    fReturn[0] = y[1]                 # RHS 1st eq
    fReturn[1] =   - 6.*pow(y[0], 5.)  # RHS 2nd

graph1 = gdisplay( width = 600, height = 600, title = 'RK 45',
                   xtitle = 't', ytitle = 'Y[0]')

funct1 = gcurve(color = color.blue)
graph2 = gdisplay( width = 500, height = 500, title = 'RK45',
                   xtitle = 't', ytitle = 'Y[1]')

funct2 = gcurve(color = color.red)
funct1.plot(pos = (t, y[0]) )
funct2.plot(pos = (t, y[1]) )

while (t < b):                              # Loop over time
    funct1.plot(pos = (t, y[0]) )
    funct2.plot(pos = (t, y[1]) )
    if ( (t + h) > b ):
        h = b - t                           # Last step
    f(t, y, fReturn)    # Evaluate RHS's, return in fReturn
    k1[0] = h*fReturn[0];     k1[1] = h*fReturn[1]
    for i in range(0, 2):
        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 in range(0, 2):
        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 in range(0, 2):
        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 in range(0, 2):
        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 in range(0, 2):
        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 in range(0, 2):
        err[i] = abs( k1[i]/360 - 128*k3[i]/4275 - 2197*k4[i]/75240 + k5[i]/50. +
            2*k6[i]/55)
    if ( err[0] < Tol or err[1] < Tol or h <= 2*hmin ): # Accept step
        for i in range(0, 2):
            y[i] = y[i] + 25*k1[i]/216. + 1408*k3[i]/2565. + 2197*k4[i]/4104. - k5[i]/5.
        t = t + h
        j = j + 1

    if ( err[0] == 0 or err[1] == 0 ):
        s = 0                                          # Trap division by 0
    else:
        s = 0.84*pow(Tol*h/err[0], 0.25)                   # Reduce step
    if ( s < 0.75 and h > 2*hmin ):
        h /= 2.                                        # Increase step
    else:
        if ( s > 1.5 and 2* h < hmax ):
            h *= 2.
    flops = flops + 1
    E = pow(y[0], 6.) + 0.5*y[1]*y[1]
    Eexact = 1.
    error = abs( (E - Eexact)/Eexact)
    sum += error

print (" < error>= ", sum/flops)
print ("flops = ", flops )
```

9.5.3 Adams–Bashforth–Moulton Predictor-Corrector

Applet Another approach for obtaining high precision in an ODE algorithm uses the solution from previous steps, say, y_{n-2} and y_{n-1}, in addition to y_n, to predict y_{n+1}. (The Euler and **rk** methods use just the previous step.) Many of these methods tend to be like a Newton's search method; we start with a guess or *prediction* for the next step and then use an algorithm such as **rk4** to check on the prediction. This yields a *correction*. As with **rk45**, one can use the difference between prediction and correction as a measure of the error and then adjust the step size to obtain improved precision [Math 92, Pres 00]. For those readers who may want to explore such methods, **ABM.py** in Listing 9.3 gives our implementation of the *Adams–Bashforth–Moulton* predictor-corrector scheme.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{1}{6}(\mathbf{k}_0 + 2\mathbf{k}_1 + 2\mathbf{k}_2 + \mathbf{k}_3),$$

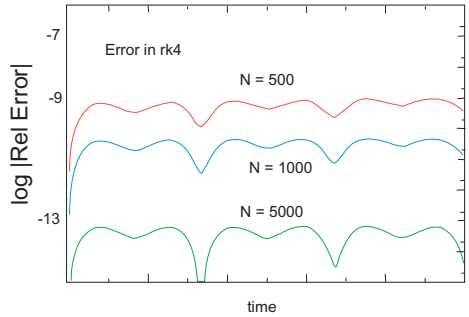
$$\mathbf{k}_0 = h\mathbf{f}(t_n,\ \mathbf{y}_n), \qquad \mathbf{k}_1 = h\mathbf{f}(t_n + \frac{h}{2},\ \mathbf{y}_n + \frac{\mathbf{k}_1}{2}),$$

$$\mathbf{k}_2 = h\mathbf{f}(t_n + \frac{h}{2},\ \mathbf{y}_n + \frac{\mathbf{k}_2}{2}), \qquad \mathbf{k}_3 = h\mathbf{f}(t_n + h,\ \mathbf{y}_n + \mathbf{k}_3).$$

Listing 9.3 **ABM.py** solves an ODE with the RHS given by the method `f()` using the ABM predictor-corrector algorithm.

```
# ABM.py: ABM method to integrate ODE
# Solves y' = (t - y)/2, with y[0] = 1 over [0, 3]

from visual import *
from visual.graph import *

numgr = gdisplay(x = 0, y = 0, width = 600, height = 300,
     title = "Numerical Solution", xtitle = 't', ytitle = 'y', xmax = 3.0,
    xmin = 0.0, ymax = 2.0, ymin = 0.9)

numsol = gcurve(color = color.yellow, display = numgr)
exactgr = gdisplay(x = 0, y = 300, width = 600, height = 300,
        title = "Exact solution", xtitle = 't', ytitle = 'y', xmax = 3.0,
        xmin = 0.0, ymax = 2.0, ymin = 0.9)

exsol =  gcurve(color = color.cyan, display = exactgr)
```

Figure 9.5 The logarithm of the relative error in the solution of an ODE obtained with `rk4` using a differing number N of time steps over a fixed time interval. The logarithm approximately equals the negative of the number of places of precision. Increasing the number of steps used for a fixed interval is seen to lead to smaller error.




```
N = 24                                                  # Number of steps > 3
a = 0; b = 3.                                       # Endpoints, inital value
t = zeros( (500), float)
y = zeros( (500), float)

def f(t, y):                                    # Return RHS's force function
    return  (t - y)/2.0

def rk4(t, y, h1):                                  # function for rk4 init
    for i in xrange(0, 100):
        t  = h1 * i
        k0 = h1 * f( t, y[i] )
        k1 = h1 * f( t + h1/2., y[i] + k0/2. )
        k2 = h1 * f( t + h1/2., y[i] + k1/2. )
        k3 = h1 * f( t + h1, y[i] + k2 )
        y[i + 1] = y[i] + (1./6.) * (k0 + 2.*k1 + 2.*k2 + k3)
    return y[3]

# Compute 3 additional starting values using rk
h = (b  -  a) / N
t[0] = a;y[0] = 1.00
F0   = f(t[0], y[0])

for k in xrange(1, 4):
    t[k] = a  +  k * h

y[1]  = rk4(t[1], y, h)
y[2]  = rk4(t[2], y, h)
y[3] = rk4(t[3], y, h)
F1 = f(t[1], y[1])
F2 = f(t[2], y[2])
F3 = f(t[3], y[3])
h2 = h / 24.00

for k in xrange(3, N):                                          # Predictor
    p = y[k]  +  h2 * (- 9.*F0  +  37.*F1  -59.*F2  +  55.*F3)
    t[k + 1] = a  +  h * (k + 1)                                # Next abscissa
    F4 = f(t[k + 1], p)                                     # Evaluate f(t, y)
    y[k + 1] = y[k]  +  h2 * (F1 - 5.*F2 + 19.*F3 + 9.*F4)  # Corrector
    F0 = F1                                                  # Update values
    F1 = F2
    F2 = F3
    F3 = f(t[k + 1], y[k + 1])

print("  k "  "    t " "     Y numerical "  "    Y exact")
for k in xrange( 0, N + 1 ):
    print(" %3d  %5.3f  %12.11f  %12.11f " %( k,t[k],y[k],(3*exp(-t[k]/2.)-2+t[k])) )
    numsol.plot(pos = (t[k], y[k]) )
    exsol.plot(pos = (t[k], 3.*exp(- t[k]/2)-2 + t[k]))
```

Table 9.1 Comparison of ODE Solvers for Different Equations

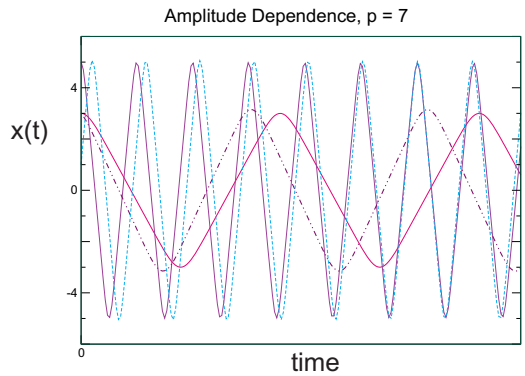
Eqtn No.	*Method*	*Initial* h	*No. of Flops*	*Time* (ms)	*Relative Error*
(9.40)	**rk4**	0.01	1000	5.2	2.2×10^{-8}
	rk45	1.00	72	1.5	1.8×10^{-8}
(9.41)	**rk4**	0.01	227	8.9	1.8×10^{-8}
	rk45	0.1	3143	36.7	5.7×10^{-11}

9.5.4 Assessment: `rk2` *versus* `rk4` *versus* `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. Regardless, we 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 give **rk4**, and **rk45** codes.

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** solver in a program that solves the equation of motion (9.6) or (9.24). Use double precision to help control subtractive cancelation 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 that you know what h values to use.
   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 with which to compare:
   $$x(t) = A\,\sin(\omega_0 t + \phi), \quad v = \omega_0 A\,\cos(\omega_0 t + \phi), \quad \omega_0 = \sqrt{\frac{k}{m}}.$$
   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 until the solution looks smooth, has a period that remains constant for a large number of cycles, and 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 a bad solution turn good.
   d. Make sure that you have exactly the same initial conditions for the analytic and numerical 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 are approximately two places of agreement.
   e. Try different initial velocities and verify that a *harmonic* oscillator is *isochronous*, that is, that 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.
5. Make a table of comparisons similar to Table 9.1. There we compare **rk4** and **rk45** for the two equations
$$2yy'' + y^2 - y'^2 = 0, \tag{9.40}$$
$$y'' + 6y^5 = 0, \tag{9.41}$$

Figure 9.6 The position versus time for oscillations within the potential $V \propto x^7$ for four different initial amplitudes. Each is seen to have a different period.



with initial conditions $([y(0), y'(0)] = (1, 1)$. Although nonlinear, (9.40) does have the analytic solution $y(t) = 1 + \sin t$. But *be warned*, the **rk** procedures may be inaccurate for this equation if integrated through $y(t) = 0$; the two terms in the equation proportional to y vanish there, leaving $y'^2 = 0$, which is problematic. A different methods, such as predictor-corrector, might then be better.[4] Equation (9.41) corresponds to our standard potential (9.4), 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** (Figure 9.5) but that it requires ∼10 times more floating-point operations and takes ∼5 times longer.

9.6 SOLUTION FOR NONLINEAR OSCILLATIONS (ASSESSMENT)

Use your **rk4** program to study anharmonic oscillations by trying powers in the range $p = 2$–12 or anharmonic strengths in the range $0 \le \alpha x \le 2$. Do *not* include any explicit 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 the solution remains periodic with constant amplitude and period for a given initial condition and value of p or α regardless of how nonlinear you make the force. In particular, check that the maximum speed occurs at $x = 0$ and that the minimum speed occurs at maximum x. The latter is a consequence of energy conservation.
2. Verify that nonharmonic oscillators are *nonisochronous*, that is, that different initial conditions (amplitudes) lead to different periods (Figure 9.6).
3. Explain why the shapes of the oscillations change for different p's or α's.
4. Devise an algorithm to determine the period T of the oscillation by recording times at which the mass passes through the origin. Note that because the motion may be asymmetric, you must record at least three times.
5. Construct a graph of the deduced period as a function of initial amplitude.
6. Verify that the motion is oscillatory but not harmonic as the initial energy approaches $k/6\alpha^2$ or for $p > 6$.
7. Verify that for the anharmonic oscillator with $E = k/6\alpha^2$, the motion changes from

[4] We thank Guenter Schneider for pointing out this possibility.

oscillatory to translational. See how close you can get to the *separatrix* where a single oscillation takes an infinite time. (There is no separatrix for the power-law potential.)

9.6.1 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 the form of the equations of motion that energy must be a constant for all values of p or α. That being the case, the constancy of energy is a demanding test of the numerics.

1. Plot the potential energy $\text{PE}(t) = V[x(t)]$, the kinetic energy $\text{KE}(t) = mv^2(t)/2$, and the total energy $E(t) = \text{KE}(t) + \text{PE}(t)$, for 50 periods. Comment on the correlation between $\text{PE}(t)$ and $\text{KE}(t)$ and how it depends on the potential's parameters.
2. Check the long-term *stability* of your solution by plotting
$$-\log_{10}\left|\frac{E(t)-E(t=0)}{E(t=0)}\right| \simeq \text{number of places of precision}$$
for a large number of periods (Figure 9.5). Because $E(t)$ 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}). If you cannot achieve 11 or more places, then you need to decrease the value of h or debug.
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 the physics behind the Virial theorem for a power-law potential:
$$\langle \text{KE} \rangle = \frac{p}{2}\langle \text{PE} \rangle. \tag{9.42}$$
Verify that your solution satisfies the Virial theorem. (Those readers who have worked the perturbed oscillator problem can use this relation to deduce an effective p value, which should be between 2 and 3.)

9.7 EXTENSIONS: NONLINEAR RESONANCES, BEATS, FRICTION

Problem: So far our oscillations have been rather simple. We have ignored friction and assumed that there are no external forces (hands) influencing 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.
3. How introducing friction affects resonances.

9.7.1 Friction (Model)

The world is full of friction, and it is not all bad. For while friction may make it harder to pedal a bike through the wind, it also tends to stabilize motion. The simplest models for frictional

force are called *static*, *kinetic*, and *viscous* friction:

$$F_f^{(\text{static})} \leq -\mu_s N, \quad F_f^{(\text{kinetic})} = -\mu_k N \frac{v}{|v|}, \quad F_f^{(\text{viscous})} = -bv. \tag{9.43}$$

Here N is the *normal force* on the object under consideration, μ and b are parameters, and v is the velocity. This model for static friction is appropriate for objects at rest, while the model for kinetic friction is appropriate for an object sliding on a dry surface. If the surface is lubricated, or if the object is moving through a viscous medium, then a frictional force proportional to velocity is a better model.[5]

1. Extend your harmonic oscillator code to include the three types of friction in (9.43) and observe how the motion differs for each.
2. *Hint:* For the simulation with static plus 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:

Underdamped:	$b < 2m\omega_0$	Oscillate within decaying envelope
Critically:	$b = 2m\omega_0$	Nonoscillatory, finite decay time
Over damped:	$b > 2m\omega_0$	Nonoscillatory, infinite decay time

9.7.2 Resonances & Beats: Model, Implementation

All stable physical systems will oscillate if displaced slightly from their rest positions. The frequency ω_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 the natural frequency ω_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, the amplitude will increase continuously 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 phenomena, known as *beating*, may occur. In this situation there is interference between the natural amplitude that is independent of the driving force plus an amplitude due to the external force. If the frequency of the driver is very close to the natural frequency, then the resulting motion,

$$x \simeq x_0 \sin \omega t + x_0 \sin \omega_0 t = \left(2x_0 \cos \frac{\omega - \omega_0}{2} t\right) \sin \frac{\omega + \omega_0}{2} t, \tag{9.44}$$

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* $\frac{\omega-\omega_0}{2}$.

[5] The effect of air resistance on projectile motion is studied in Unit III of this chapter.

9.8 EXTENSION: TIME-DEPENDENT FORCES

To extend our simulation to include an external force,

$$F_{\text{ext}}(t) = F_0 \sin \omega t, \tag{9.45}$$

we need to include some time dependence in the force function $\mathbf{f}(t, \mathbf{y})$ that occurs in our ODE solver.

1. Add the sinusoidal time-dependent external force (9.45) 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. Verify that for the harmonic oscillator, 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, where $\omega \simeq \omega_0$.
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 nonlinear systems resonate. If the nonlinear system is close to being harmonic, you should get beating in place 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 the amplitude decreases; with the decrease in amplitude, the frequency of the oscillator returns to its natural frequency, the driver and oscillator get back in phase, and the entire cycle repeats.
8. Investigate now how the inclusion of viscous friction modifies the curve of amplitude *versus* driver frequency. You should find that friction broadens the curve.
9. Explain 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, and so the driver is less effective.

9.9 UNIT II. BINDING A QUANTUM PARTICLE

Problem: In this unit is we want to determine whether the rules of quantum mechanics are applicable inside a nucleus. More specifically, you are told that nuclei contain neutrons and protons (nucleons) with mass $mc^2 \simeq 940\,\text{MeV}$ and that a nucleus has a size of about 2 fm.[6] 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.

This problem requires us to solve the bound-state eigenvalue problem for the 1-D, time-independent 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

[6] A fermi (fm) equals 10^{-13} cm $= 10^{-15}$ m, and $\hbar c \simeq 197.32\,\text{MeV fm}$.

makes this an eigenvalue problem. Specifically, the bound-state requirement imposes boundary conditions on the form of the solution, which in turn means that a solution exists 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 gain experience with the technique of conducting a numerical search for the eigenvalue in conjunction with solving an ODE numerically. This is how one solves the numerical ODE eigenvalue problem. In §20.2.1, we discuss how to solve the equivalent, but more advanced, momentum-space eigenvalue problem as a matrix problem. In Chapter 18, PDE Waves: String, Quantum Packet, *and 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 [Schd 00, Koon 86].*

9.10 THEORY: QUANTUM EIGENVALUE PROBLEM

Quantum mechanics describes phenomena that occur on atomic or subatomic scales (an elementary particle is subatomic). It is a statistical theory in which the probability that a particle is located in a region dx around the point x is $\mathcal{P} = |\psi(x)|^2\,dx$, where $\psi(x)$ is called the *wave function*. If a particle of definite energy E moving in one dimension experiences a potential $V(x)$, its wave function is determined by an ordinary differential equation (or a partial differential equation for more than one dimension) known as the time-independent Schrödinger equation[7]:

$$\frac{-\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x). \tag{9.46}$$

Although we say we are solving for the energy E, in practice we solve for the wave vector κ. The energy is negative for bound states, and so we relate the two by

$$\kappa^2 = -\frac{2m}{\hbar^2}E = \frac{2m}{\hbar^2}|E|. \tag{9.47}$$

The Schrödinger equation then takes the form

$$\frac{d^2\psi(x)}{dx^2} - \frac{2m}{\hbar^2}V(x)\psi(x) = \kappa^2\psi(x). \tag{9.48}$$

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 only way to have a $\psi(x)$ with a finite integral is to have it decay exponentially as $x \to \pm\infty$ (where the potential vanishes):

$$\psi(x) \to \begin{cases} e^{-\kappa x}, & \text{for } x \to +\infty, \\ e^{+\kappa x}, & \text{for } x \to -\infty. \end{cases} \tag{9.49}$$

In summary, although it is straightforward to solve the ODE (9.46) with the techniques we have learned so far, we must also require that the solution $\psi(x)$ simultaneously satisfies the boundary conditions (9.49). This extra condition turns the ODE problem into an *eigenvalue problem* that has solutions (*eigenvalues*) for only 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).

[7] The time-dependent equation requires the solution of a partial differential equation, as discussed in Chapter 18, "PDE Waves: String, Quantum Packet, and E&M."

9.10.1 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 and to permit some analytic checking, we will use a simple model in which the potential $V(x)$ in (9.46) is a finite square well (Figure 9.7):

$$V(x) = \begin{cases} -V_0 = -83\,\text{MeV}, & \text{for } |x| \le a = 2\,\text{fm}, \\ 0, & \text{for } |x| > a = 2\,\text{fm}, \end{cases} \tag{9.50}$$

where values of 83 MeV for the depth and 2 fm for the radius are typical for nuclei (these are the units in which we solve the problem). With this potential the Schrödinger equation (9.48) becomes

$$\frac{d^2\psi(x)}{dx^2} + \left(\frac{2m}{\hbar^2}V_0 - \kappa^2\right)\psi(x) = 0, \quad \text{for } |x| \le a, \tag{9.51}$$

$$\frac{d^2\psi(x)}{dx^2} - \kappa^2\psi(x) = 0, \quad \text{for } |x| > a. \tag{9.52}$$

To evaluate the ratio of constants here, we insert c^2, the speed of light squared, into both the numerator and the denominator [L 96, Appendix A.1]:

$$\frac{2m}{\hbar^2} = \frac{2mc^2}{(\hbar c)^2} \simeq \frac{2 \times 940\,\text{MeV}}{(197.32\,\text{MeV fm})^2} = 0.0483\,\text{MeV}^{-1}\,\text{fm}^{-2}. \tag{9.53}$$

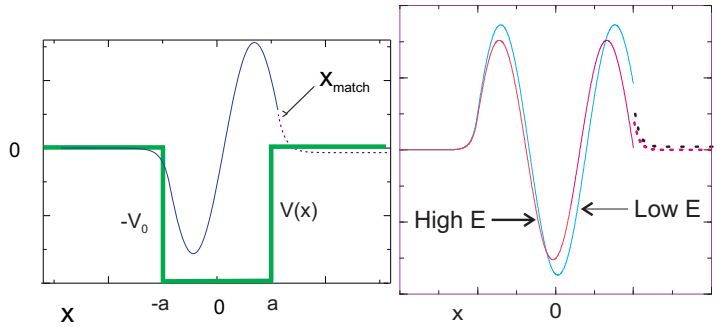
9.11 DUAL ALGORITHMS: EIGENVALUES VIA ODE SOLVER + SEARCH

The solution of the eigenvalue problem combines the numerical solution of the ordinary differential equation (9.48) with a trial-and-error search for a wave function that satisfies the boundary conditions (9.49). This is done in several steps:[8]

1. Start on the very far *left* at $x = -X_{\text{max}} \simeq -\infty$, where $X_{\text{max}} \gg a$. Since the potential $V = 0$ in this region, the analytic solution here is $e^{\pm\kappa x}$. Accordingly, assume that the wave function there satisfies the left-hand boundary condition:
$$\psi_L(x = -X_{\text{max}}) = e^{+\kappa x} = e^{-\kappa X_{\text{max}}}.$$
2. Use your favorite ODE solver to step $\psi_L(x)$ in toward the origin (to the right) from $x = -X_{\text{max}}$ until you reach the *matching radius* x_{match}. The exact value of this matching radius is not important, and our final solution should be independent of it. On the left in Figure 9.7, we show a sample solution with $x_{\text{match}} = -a$; that is, we match at the left edge of the potential well. In the middle and on the right in Figure 9.7 we see some guesses that do not match.
3. Start on the very far *right*, that is, at $x = +X_{\text{max}} \simeq +\infty$, with a wave function that satisfies the right-hand boundary condition:
$$\psi_R(x = +\kappa X_{\text{max}}) = e^{-\kappa x} = e^{-\kappa X_{\text{max}}}.$$
4. Use your favorite ODE solver (e.g., **`rk4`**) to step $\psi_R(x)$ in toward the origin (to the left) from $x = +X_{\text{max}}$ until you reach the *matching radius* x_{match}. This means that we have stepped through the potential well (Figure 9.7).

[8] The procedure outlined here is for a general potential that falls off gradually. For a square well with sharp cutoffs the analytic solution is valid right up unto the walls, and we could start integrating inwards from there in this special case. In contrast, if we were working with a Coulomb potential, its very slow falloff would does not match onto a simple exponential, even at infinity [L 96].

Figure 9.7 *Left:* Computed wave function and the square-well potential (the potential is in bold lines). The wave function computed by integration in from the left is matched to the one computed by integration in from the right (dashed curve) at a point near the right edge of the well. Note how the wave function decays rapidly outside the well. *Right:* A first guess at a wave function with energy E that is 0.5% too low (dotted curve). We see that the left wave function does not vary rapidly enough to match the right one at $x = 500$. The solid curve shows a second guess at a wave function with energy E that is 0.5% too high. We see that now the left wave function varies too rapidly.



5. In order for probability and current to be continuous at $x = x_{\text{match}}$, $\psi(x)$ and $\psi'(x)$ must be continuous there. 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 ψ's normalization.
6. Even though we do not know ahead of time which energies E or κ values are eigenvalues, we still need a starting value for the energy in order to use our ODE solver. Such being the case, we start the solution with a guess for the energy. A good guess for ground-state energy would be a value somewhat up from that at the bottom of the well, $E > -V_0$.
7. Because it is unlikely that any guess will be correct, the left- and right-wave functions will not quite match at $x = x_{\text{match}}$ (Figure 9.7). This is okay because we can use the amount of mismatch to improve the next guess. We measure how well the right and left wave functions match by calculating the difference

$$\Delta(E,x) = \left.\frac{\psi_L'(x)/\psi_L(x) - \psi_R'(x)/\psi_R(x)}{\psi_L'(x)/\psi_L(x) + \psi_R'(x)/\psi_R(x)}\right|_{x=x_{\text{match}}}, \tag{9.54}$$

where the denominator is used to avoid overly large or small numbers. Next we try a different energy, note how much $\Delta(E)$ has changed, and use this to deduce an intelligent guess at the next energy. The search continues until the left- and right-wave ψ'/ψ match within some tolerance.

9.11.1 Numerov Algorithm for Schrödinger ODE ⊙

We generally recommend the fourth-order Runge–Kutta method for solving ODEs, and its combination with a search routine for solving the eigenvalue problem. In this section we present the Numerov method, an algorithm that is specialized for ODEs not containing any first derivatives (such as our Schrödinger equation). While this algorithm is not as general as `rk4`*, it is of $\mathcal{O}(h^6)$ and thus speeds up the calculation by providing additional precision.*

We start by rewriting the Schrödinger equation (9.48) in the compact form,

$$\frac{d^2\psi}{dx^2} + k^2(x)\psi = 0, \quad k^2(x) \stackrel{\text{def}}{=} \frac{2m}{\hbar^2}\begin{cases} E + V_0, & \text{for } |x| < a, \\ E, & \text{for } |x| > a, \end{cases} \tag{9.55}$$

where $k^2 = -\kappa^2$ in potential-free regions. Observe that although (9.55) is specialized to a square well, other potentials would have $V(x)$ in place of $-V_0$. The trick in the Numerov method is to get extra precision in the second derivative by taking advantage of there being no first derivative $d\psi/dx$ in (9.55). We start with the Taylor expansions of the wave functions,

$$\psi(x+h) \simeq \psi(x) + h\psi^{(1)}(x) + \frac{h^2}{2}\psi^{(2)}(x) + \frac{h^3}{3!}\psi^{(3)}(x) + \frac{h^4}{4!}\psi^{(4)}(x) + \cdots$$

$$\psi(x-h) \simeq \psi(x) - h\psi^{(1)}(x) + \frac{h^2}{2}\psi^{(2)}(x) - \frac{h^3}{3!}\psi^{(3)}(x) + \frac{h^4}{4!}\psi^{(4)}(x) + \cdots,$$

where $\psi^{(n)}$ signifies the nth derivative $d^n\psi/dx^n$. Because the expansion of $\psi(x-h)$ has odd powers of h appearing with negative signs, all odd powers cancel when we add $\psi(x+h)$ and $\psi(x-h)$ together:

$$\begin{aligned}\psi(x+h) + \psi(x-h) \simeq & 2\psi(x) + h^2\psi^{(2)}(x) + \frac{h^4}{12}\psi^{(4)}(x) + \mathcal{O}(h^6),\\ \Rightarrow \quad \psi^{(2)}(x) \simeq & \frac{\psi(x+h) + \psi(x-h) - 2\psi(x)}{h^2}\\ & - \frac{h^2}{12}\psi^{(4)}(x) + \mathcal{O}(h^4).\end{aligned} \tag{9.56}$$

To obtain an algorithm for the second derivative we eliminate the fourth-derivative term by applying the operator $1 + \frac{h^2}{12}\frac{d^2}{dx^2}$ to the Schrödinger equation (9.55):

$$\psi^{(2)}(x) + \frac{h^2}{12}\psi^{(4)}(x) + k^2(x)\psi + \frac{h^2}{12}\frac{d^2}{dx^2}[k^2(x)\psi^{(4)}(x)] = 0.$$

We eliminate the $\psi^{(4)}$ terms by substituting the derived expression for the $\psi^{(2)}$:

$$\frac{\psi(x+h) + \psi(x-h) - 2\psi(x)}{h^2} + k^2(x)\psi(x) + \frac{h^2}{12}\frac{d^2}{dx^2}[k^2(x)\psi(x)] \simeq 0.$$

Now we use a central-difference approximation for the second derivative of $k^2(x)\psi(x)$

$$h^2\frac{d^2[k^2(x)\psi(x)]}{dx^2} \simeq [(k^2\psi)_{x+h} - (k^2\psi)_x] + [(k^2\psi)_{x-h} - (k^2\psi)_x].$$

After this substitution we obtain the Numerov algorithm:

$$\psi(x+h) \simeq \frac{2[1 - \frac{5}{12}h^2k^2(x)]\psi(x) - [1 + \frac{h^2}{12}k^2(x-h)]\psi(x-h)}{1 + h^2k^2(x+h)/12}. \tag{9.57}$$

We see that the Numerov algorithm uses the values of ψ at the two previous steps x and $x-h$ to move ψ forward to $x+h$. To step backward in x, we need only to reverse the sign of h. Our implementation of this algorithm, **`Numerov.py`**, is given in Listing 9.4.

Listing 9.4 **QuantumNumerov.py** solves the 1-D time-independent Scrödinger equation for bound-state energies using a Numerov method (rk4 also works, as we show in Listing 9.5).

```
#  QuantumNumerov:  Quantum BS via Numerov ODE solver + search

from visual import *

psigr  = display(x=0, y=0, width=600,height=300,title='R & L Wave Funcs')
psi    = curve(x=list(range(0,1000)), display=psigr, color=color.yellow)
psi2gr = display(x=0,y=300,width=600,height=200,title='Wave func^2')
psio   = curve(x=list(range(0,1000)),color=color.magenta, display=psi2gr)
energr = display(x=0, y=500, width=600,height=200,title='Potential & E')
poten  = curve(x=list(range(0,1000)), color=color.cyan, display=energr)
autoen = curve(x=list(range(0,1000)), display=energr)
```

```
dl    = 1e-6                        # very small interval to stop bisection
ul    = zeros([1501], float)
ur    = zeros([1501], float)
k2l   = zeros([1501], float)      # k**2 in Schroedinger eq. left wavefunc
k2r   = zeros([1501], float)     # k**2 in Schroedinger eq. right wavefunc
n     = 1501
m     = 5                                       # to plot every 5 points
imax  = 100
x10   = -1000;   xr0   =  1000       # leftmost, rightmost x wave function
h     = (1.0*(xr0-x10)/(n-1.))              # 1.0 to make it a float number
amin  = -0.001; amax  = -0.00085              # root between amin and amax
e     = amin                                    # Initial guess for energy
de    = 0.01
ul[0] = 0.0; ul[1] = 0.00001; ur[0] = 0.0; ur[1] = 0.00001
im = 500                                     # match point left and right wv
nl = im+2;   nr = n-im+1                                # for left, right wv
istep=0

def V(x):                                     # Potential finite square well
    if (abs(x)<=500):   v = -0.001                            # well depth
    else:               v = 0
    return v

def setk2():                                # sets k2l=(sqrt(e-V))^2 and k2r
    for i in range(0,n):
       xl = xl0+i*h
       xr = xr0-i*h
       k2l[i] = e-V(xl)
       k2r[i] = e-V(xr)

def numerov (n,h,k2,u):                  # Numerov algorithm can be used for
    b=(h**2)/12.0                            # left and right wave functions
    for i in range(1, n-1):
     u[i+1] = (2*u[i]*(1-5*b*k2[i])-(1.+b*k2[i-1])*u[i-1])/(1+b*k2[i+1])

setk2()
numerov (nl, h, k2l, ul)                          # finds left wave function
numerov (nr, h, k2r, ur)                         # finds right wave function
fact= ur[nr-2]/ul[im]                                  # to Rescale  solution
for i  in range (0,nl): ul[i] = fact*ul[i]
f0 = (ur[nr-1]+ul[nl-1]-ur[nr-3]-ul[nl-3])/(2*h*ur[nr-2])    #  Log deriv

def normalize():
    asum = 0
    for i in range( 0,n):                       # to normalize wave function
        if i > im :
            ul[i] = ur[n-i-1]
            asum = asum+ul[i]*ul[i]
    asum        = sqrt(h*asum);
    elabel      = label(pos=(700, 500), text='e=', box=0,display=psigr)
    elabel.text = 'e=%10.8f' %e
    ilabel      = label(pos=(700,400),text='istep=',box=0,display=psigr)
    ilabel.text = 'istep=%4s' %istep
    poten.pos   = [(-1500,200),(-1000,200),(-1000,-200),
                            (0,-200),(0,200),(1000,200)]
    autoen.pos = [(-1000,e*400000.0+200),(0,e*400000.0+200)]
    label(pos=(-1150,-240), text='0.001', box=0, display=energr)
    label(pos=(-1000,300),  text='0',     box=0, display=energr)
    label(pos=(-900,180),   text='-500',  box=0, display=energr)
    label(pos=(-100,180),   text='500',   box=0, display=energr)
    label(pos=(-500,180),   text='0',     box=0, display=energr)
    label(pos=(900,120),    text='r',     box=0, display=energr)

    j=0
    for i in range(0,n,m):
        xl        = xl0 + i*h
        ul[i]     = ul[i]/asum                    # wave function normalized
        psi.x[j]  = xl - 500                                     # plot psi
        psi.y[j]  = 10000.0*ul[i]        # vertical line for match of wvfs
        line      = curve(pos=[(-830,-500),(-830,500)],
                    color=color.red,display=psigr)
        psio.x[j] = xl-500                                       # plot psi
        psio.y[j] = 1.0e5*ul[i]**2
        j +=1

while abs(de) > dl and istep < imax :        # bisection algorithm begins
    rate(2)                                       # to slowdown  animation
    e1 = e                                                  # guessed root
    e  = (amin+amax)/2                                     # half interval
    for i in range(0,n):
        k2l[i] = k2l[i] + e-e1
        k2r[i] = k2r[i] + e-e1
```

```
    im = 500;
    nl = im+2
    nr = n-im+1;
    numerov (nl,h,k2l,ul)              # Find wavefuntions for new k2l,k2r
    numerov (nr,h,k2r,ur)
    fact = ur[nr-2]/ul[im]
    for i in range(0,nl):  ul[i] = fact*ul[i]
    f1 = (ur[nr-1]+ul[nl-1]-ur[nr-3]-ul[nl-3])/(2*h*ur[nr-2]) # Log deriv
    rate(2)
    if f0*f1 < 0:                              # bisection localize root
        amax = e
        de = amax - amin
    else:
         amin = e
         de = amax - amin
         f0 = f1
    normalize()
    istep = istep + 1
```

9.11.2 Implementation: Eigenvalues via ODE Solver + Bisection Algorithm

1. Combine your bisection algorithm search program with your **`rk4`** or Numerov ODE solver program to create an eigenvalue solver. Start with a step size $h = 0.04$.
2. Write a subroutine that calculates the matching function $\Delta(E, x)$ 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 $\Delta(E, x = 2)$ vanishes.
3. As a first guess, take $E \simeq 65\,\text{MeV}$.
4. Search until $\Delta(E, x)$ changes in only the fourth decimal place. We do this in the code **`QuantumEigen.py`** shown in Listing 9.5.
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, 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 odd (the ground state must be even).
9. Include in your version of Figure 9.7 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.
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-fermi well. ▌

Listing 9.5 **QuantumEigen.py** solves the 1-D time-independent Schrödinger equation for bound-state energies using the rk4 algorithm.

```
# QuantumEigen.py: Finds E and psi via rk4 + bisection

# mass/((hbar*c)**2)= 940MeV/(197.33MeV-fm)**2 =0.4829, well width=20.0 fm
# well depth 10 MeV, Wave function not normalized. Potential not to scale.

from visual import *
```

```
psigr = display(x=0,y=0,width=600,height=300, title='R & L Wavefunc')
Lwf = curve(x=list(range(502)),color=color.red)
Rwf = curve(x=list(range(997)),color=color.yellow)
eps       = 1E-3                                  # variables,  precision
n_steps   = 501
E         = -17.0                            # try this energy as a guess
h         = 0.04
count_max = 100
Emax      = 1.1*E                           # between these Emax and Emin
Emin      = E/1.1                                     # is the eigenenerg

def f(x, y, F,E):
    F[0] = y[1]
    F[1] = -(0.4829)*(E-V(x))*y[0]

def V(x):
    if (abs(x) < 10.):  return (-16.0)             # Potential well depth
    else:               return (0.)

def rk4(t, y,h,Neqs,E):
    F  = zeros((Neqs),float)
    ydumb     = zeros((Neqs),float)
    k1 = zeros((Neqs),float)
    k2 = zeros((Neqs),float)
    k3 = zeros((Neqs),float)
    k4 = zeros((Neqs),float)
    f(t, y, F,E)
    for i in range(0,Neqs):
        k1[i] = h*F[i]
        ydumb[i] = y[i] + k1[i]/2.
    f(t + h/2., ydumb, F,E)
    for i in range(0,Neqs):
        k2[i] = h*F[i]
        ydumb[i] = y[i] + k2[i]/2.
    f(t + h/2., ydumb, F,E)
    for i in range(0,Neqs):
        k3[i]=  h*F[i]
        ydumb[i] = y[i] + k3[i]
    f(t + h, ydumb, F,E);
    for i in range(0,Neqs):
        k4[i]=h*F[i]
        y[i]=y[i]+(k1[i]+2*(k2[i]+k3[i])+k4[i])/6.0

def diff(E, h):
    y = zeros((2),float)
    i_match = n_steps//3                                  # Matching radius
    nL = i_match + 1
    y[0] = 1.E-15;                                  # Initial wf on left
    y[1] = y[0]*sqrt(-E*0.4829)
    for ix in range(0,nL + 1):
        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]*sqrt(-E*0.4829)                          # Initialize R wf
    for ix in range( n_steps,nL+1,-1):
        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) )

def plot(E, h):                                # Repeat integrations for plot
    x = 0.
    n_steps = 1501                            # Total no integration steps
    y = zeros((2),float)
    yL = zeros((2,505),float)
    i_match = 500                                          # Matching point
    nL = i_match + 1;
    y[0] = 1.E-40                                  # Initial wf on the left
    y[1] = -sqrt(-E*0.4829) *y[0]
    for ix in range(0,nL+1):                          # left wave function
        yL[0][ix] = y[0]
        yL[1][ix] = y[1]
        x = h * (ix -n_steps/2)
        rk4(x, y, h, 2, E)
    y[0] = -1.E-15                          # - slope: even;  reverse for odd
    y[1] = -sqrt(-E*0.4829)*y[0]
    j=0
    for ix in range(n_steps -1,nL + 2,-1):           # right wave function
        x = h * (ix + 1 -n_steps/2)                          # Integrate in
        rk4(x, y, -h, 2, E)
        Rwf.x[j] = 2.*(ix + 1 -n_steps/2)-500.0
        Rwf.y[j] = y[0]*35e-9 +200
```

```
        j +=1
    x = x-h
    normL = y[0]/yL[0][nL]
    j=0
    # Renormalize L wf & derivative
    for ix in range(0,nL+1):
        x = h * (ix-n_steps/2 + 1)
        y[0] = yL[0][ix]*normL
        y[1] = yL[1][ix]*normL
        Lwf.x[j] = 2.*(ix  -n_steps/2+1)-500.0
        Lwf.y[j] = y[0]*35e-9+200                # factor to  reduce scale
        j +=1
for count in range(0,count_max+1):
    rate(1)                                  # slowest rate to show changes
    # Iteration loop
    E = (Emax + Emin)/2.                                  # Divide E range
    Diff = diff(E, h)
    if (diff(Emax, h)*Diff > 0):  Emax = E          # Bisection algorithm
    else:                         Emin = E
    if ( abs(Diff)  <  eps ):     break
    if count >3:        # first iterates are very irregular
        rate(4)
        plot(E, h)
    elabel      = label(pos=(700, 400), text='E=', box=0)
    elabel.text = 'E=%13.10f' %E
    ilabel      = label(pos=(700, 600), text='istep=', box=0)
    ilabel.text = 'istep=%4s' %count
elabel      = label(pos=(700, 400), text='E=', box=0)    # last iteration
elabel.text = 'E=%13.10f' %E
ilabel      = label(pos=(700, 600), text='istep=', box=0)
ilabel.text = 'istep=%4s' %count

print("Final eigenvalue E = ",E)
print("iterations, max = ",count)
```

9.12 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 $E \geq -V_0$, 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 several bound states. 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. $\odot$ Conduct some explorations in which you discover different combinations of (V_0, a) that give the same ground-state energies (discrete ambiguities). The existence of several different combinations means that a knowledge of 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 smaller 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, using both the Numerov and `rk4` methods.
9. **Newton–Raphson extension:** Extend the eigenvalue search by using the Newton–Raphson method in place of the bisection algorithm. Determine how much faster and more precise it is.

9.13 UNIT III. SCATTERING, PROJECTILES & ORBITS

9.14 PROBLEM 1: CLASSICAL CHAOTIC SCATTERING

Applet **Problem:** One expects the classical scattering of a projectile from a barrier to be a continuous process. Yet it has been observed in experiments conducted on pinball machines (Figure 9.8 left) that for certain conditions, the projectile undergoes multiple internal scatterings and ends up with a final trajectory that is apparently unrelated to the initial one. Your **problem** is to determine if this process can be modeled as scattering from a static potential or if there must be active mechanisms built into the pinball machines that cause chaotic scattering.

Although this problem is easy to solve on the computer, the results have some chaotic features that are surprising (chaos is discussed further in Chapter 12, "Discrete & Continuous Nonlinear Dynamics"*). In fact, the applet* `Disper2e.html` *(created by Jaime Zuluaga) that simulates this problem continues to be a source of wonderment for readers as well as authors.*

9.14.1 Model and Theory

Our model for balls bouncing off the electrically activated bumpers in pinball machines is a point particle scattering from the stationary 2-D potential [Bleh 90]

$$V(x, y) = \pm x^2 y^2 e^{-(x^2+y^2)}. \tag{9.58}$$

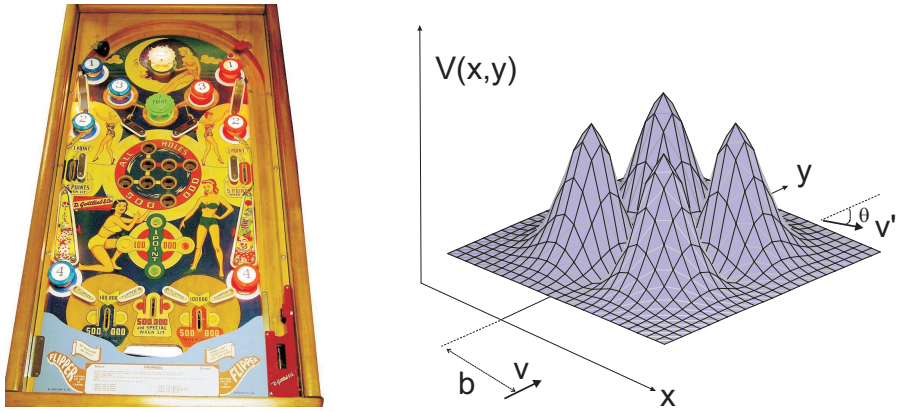
This potential has four circularly symmetric peaks in the xy plane (Figure 9.8 right). The two signs correspond to repulsive and attractive potentials, respectively (the pinball machine contains only repulsive interactions). Because there are four peaks in this potential, we suspect that it may be possible to have multiple scatterings in which the projectile bounces back and forth among the peaks, somewhat as in a pinball machine.

The **theory** for this problem is classical dynamics. Visualize a scattering experiment in which a projectile starting out at an infinite distance from a target with a definite velocity $\mathbf{v}$ and an impact parameter b (Figure 9.8 right) is incident on a target. After interacting with the target and moving a nearly infinite distance from it, the scattered particle is observed at the scattering angle θ. Because the potential cannot recoil, the speed of the projectile does not change, but its direction does. An experiment typically measures the number of particles scattered and then converts this to a function, the differential cross section $\sigma(\theta)$, which is independent of the details of the experimental apparatus:

$$\sigma(\theta) = \lim \frac{N_{\text{scatt}}(\theta)/\Delta\Omega}{N_{\text{in}}/\Delta A_{\text{in}}}. \tag{9.59}$$

Here $N_{\text{scatt}}(\theta)$ is the number of particles per unit time scattered into the detector at angle θ subtending a solid angle $\Delta\Omega$, N_{in} is the number of particle per unit time incident on the target of cross-sectional area ΔA_{in}, and the limit in (9.59) is for infinitesimal detector and area sizes.

Figure 9.8 *Left:* A photograph of a pinball machine in which multiple scatterings occur from the bumpers. *Right:* Scattering from the potential $V(x,y) = x^2y^2e^{-(x^2+y^2)}$ which in some ways models the pinball machine. The incident velocity v is in the y direction, and the impact parameter (x value) is b. The velocity of the scattered particle is v' and its scattering angle is θ.



The definition (9.59) for the cross section is the one that experimentalists use to convert their measurements to a function that can be calculated by theory. We as theorists solve for the trajectories of particles scattered from the potential (9.58) and from them deduce the scattering angle θ. Once we have the scattering angle, we predict the differential cross section from the dependence of the scattering angle upon the classical impact parameter b [M&T 03]:

$$\sigma(\theta) = \left|\frac{d\theta}{db}\right| \frac{b}{\sin\theta(b)}. \tag{9.60}$$

The surprise you should find in the simulation is that for certain parameters, $d\theta/db$ can get to be very large or discontinuous, and this accordingly leads to large and discontinuous cross sections.

The dynamical equations to solve are just Newton's law for the x and y motions with the potential (9.58):

$$\begin{aligned} \mathbf{F} &= m\mathbf{a} \\ -\frac{\partial V}{\partial x}\hat{\imath} - \frac{\partial V}{\partial y}\hat{\jmath} &= m\frac{d^2\mathbf{x}}{dt^2}, \\ \mp 2xye^{-(x^2+y^2)}\left[y(1-x^2)\hat{\imath} + x(1-y^2)\hat{\jmath}\right] &= m\frac{d^2x}{dt^2}\hat{\imath} + m\frac{d^2y}{dt^2}\hat{\jmath}. \end{aligned} \tag{9.61}$$

The equations for the x and y motions are simultaneous second-order ODEs:

$$m\frac{d^2x}{dt^2} = \mp 2y^2x(1-x^2)e^{-(x^2+y^2)}, \tag{9.62}$$

$$m\frac{d^2y}{dt^2} = \mp 2x^2y(1-y^2)e^{-(x^2+y^2)}. \tag{9.63}$$

Because the force vanishes at the peaks in Figure 9.8, these equations tell us that the peaks are at $x = \pm 1$ and $y = \pm 1$. Substituting these values into the potential (9.58) yields $V_{\max} = \pm e^{-2}$, which sets the energy scale for the problem.

9.14.2 Implementation

In §9.16.1 we will also describe how to express simultaneous ODEs such as (9.62) and (9.63) in the standard **rk4** form. Even though both equations are independent, we solve them simultaneously to determine the scattering trajectory $[x(t), y(t)]$. We use the same **rk4** algorithm we used for a single second-order ODE, only now the arrays will be 4-D rather than 2-D:

$$\frac{d\mathbf{y}(t)}{dt} = \mathbf{f}(t, \mathbf{y}), \tag{9.64}$$

$$y^{(0)} \stackrel{\text{def}}{=} x(t), \quad y^{(1)} \stackrel{\text{def}}{=} y(t), \tag{9.65}$$

$$y^{(2)} \stackrel{\text{def}}{=} \frac{dx}{dt}, \quad y^{(3)} \stackrel{\text{def}}{=} \frac{dy}{dt}, \tag{9.66}$$

where the order in which the $y^{(i)}$s are assigned is arbitrary. With these definitions and equations (9.62) and (9.63), we can assign values for the force function:

$$f^{(0)} = y^{(2)}, \qquad f^{(1)} = y^{(3)}, \tag{9.67}$$

$$f^{(2)} = \frac{\mp 1}{m} 2y^2 x(1-x^2)e^{-(x^2+y^2)} = \frac{\mp 1}{m} 2{y^{(1)}}^2 y^{(0)}(1-{y^{(0)}}^2)e^{-({y^{(0)}}^2+{y^{(1)}}^2)},$$

$$f^{(3)} = \frac{\mp 1}{m} 2x^2 y(1-y^2)e^{-(x^2+y^2)} = \frac{\mp 1}{m} 2{y^{(0)}}^2 y^{(1)}(1-{y^{(1)}}^2)e^{-({y^{(0)}}^2+{y^{(1)}}^2)}.$$

To deduce the scattering angle from our simulation, we need to examine the trajectory of the scattered particle at an "infinite" separation from the target. To approximate that, we wait until the scattered particle no longer feels the potential (say $|\text{PE}|/\text{KE} \leq 10^{-10}$) and call this infinity. The scattering angle is then deduced from the components of velocity,

$$\theta = \tan^{-1}\left(\frac{v_y}{v_x}\right) = \texttt{atan2(Vx, Vy)}. \tag{9.68}$$

Here **atan2** is a function in most computer languages that computes the arctangent in the correct quadrant without requiring any explicit divisions (that can blow up).

Listing 9.6 **ProjectileAir.py** solves for projectile motion with air resistance as well as analytically for the frictionless case.

```
# ProjectileAir solves numerically for projectile motion with drag;

from visual.graph import *
v0 = 22.
angle = 34.
g = 9.8
kf = 0.9
N = 25
v0x = v0*cos(angle*pi/180.)
v0y = v0*sin(angle*pi/180.)
T = 2.*v0y/g
H = v0y*v0y/(2.*g)
R = 2.*v0x*v0y/g
graph1 = gdisplay( title='Projectile with (red)/without (yellow) Air Resistance',
          xtitle='x', ytitle='y', xmax=R, xmin=-R/20.,ymax=8,ymin=-6.0)
funct = gcurve(color=color.red)
funct1 = gcurve(color=color.yellow)
print('Frictionless T = ',T)
print('Frictionless H = ',H)
print('Frictionless R = ',R)
def plotNumeric(k):
 vx = v0*cos(angle*pi/180.)
 vy = v0*sin(angle*pi/180.)
 x = 0.0
 y = 0.0
 dt =  vy/g/N/2.
 print("       x              y")
 for i in range(3*N):
    rate(30)
```

```
    vx = vx - k*vx*dt
    vy = vy - g*dt - k*vy*dt
    x = x + vx*dt
    y = y + vy*dt
    funct.plot(pos=(x,y))
    print(" %13.10f  %13.10f "%(x,y))

def plotAnalytic():
    v0x = v0*cos(angle*pi/180.)
    v0y = v0*sin(angle*pi/180.)
    dt = 2.*v0y/g/N
    print("       FRICTIONLESS  ")
    print("        x            y")
    for i in range(N):
        rate(30)
        t = i*dt
        x = v0x*t
        y = v0y*t -g*t*t/2.
        funct1.plot(pos=(x,y))
        print(" %13.10f  %13.10f"%(x ,y))

plotNumeric(kf)
plotAnalytic()
```

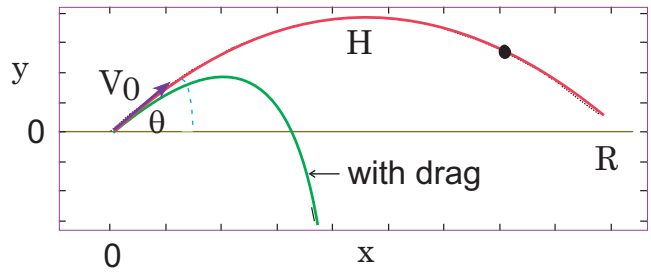
9.14.3 Assessment

1. Apply the **`rk4`** method to solve the simultaneous second-order ODEs (9.62) and (9.63) with a 4-D force function.
2. The **initial conditions** are (a) an incident particle with only a y component of velocity and (b) an impact parameter b (the initial x value). You do not need to vary the initial y, but it should be large enough such that $\mathrm{PE}/\mathrm{KE} \leq 10^{-10}$, which means that the $\mathrm{KE} \simeq E$.
3. Good parameters are $m = 0.5$, $v_y(0) = 0.5$, $v_x(0) = 0.0$, $\Delta b = 0.05$, $-1 \leq b \leq 1$. You may want to lower the energy and use a finer step size once you have found regions of rapid variation.
4. Plot a number of trajectories $[x(t), y(t)]$ that show usual and unusual behaviors. In particular, plot those for which backward scattering occurs, and consequently for which there is much multiple scattering.
5. Plot a number of phase space trajectories $[x(t), \dot{x}(t)]$ and $[y(t), \dot{y}(t)]$. How do these differ from those of bound states?
6. Determine the scattering angle $\theta =$ **`atan2(Vx,Vy)`** by determining the velocity of the scattered particle after it has left the interaction region, that is, $\mathrm{PE}/\mathrm{KE} \leq 10^{-10}$.
7. Identify which characteristics of a trajectory lead to discontinuities in $d\theta/db$ and thus $\sigma(\theta)$.
8. Run the simulations for both attractive and repulsive potentials and for a range of energies less than and greater than $V_{\max} = \exp(-2)$.
9. **Time delay:** Another way to find unusual behavior in scattering is to compute the *time delay* $T(b)$ as a function of the impact parameter b. The time delay is the increase in the time it takes a particle to travel through the interaction region after the interaction is turned on. Look for highly oscillatory regions in the semilog plot of $T(b)$, and once you find some, repeat the simulation at a finer scale by setting $b \simeq b/10$ (the structures are fractals, see Chapter 13, "Fractals & Statistical Growth").

9.15 PROBLEM 2: BALLS FALLING OUT OF THE SKY

Golf and baseball players claim that hit balls appear to fall straight down out of the sky at the end of their trajectories (the solid curve in Figure 9.9). 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

Figure 9.9 The trajectories of a projectile fired with initial velocity V_0 in the θ direction. The lower curve includes air resistance.



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.

9.16 THEORY: PROJECTILE MOTION WITH DRAG

Figure 9.9 shows the initial velocity V_0 and inclination θ for a projectile launched from the origin. 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

$$x(t) = V_{0x}t, \qquad y(t) = V_{0y}t - \tfrac{1}{2}gt^2, \tag{9.69}$$

$$v_x(t) = V_{0x}, \qquad v_y(t) = V_{0y} - gt, \tag{9.70}$$

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 show that the trajectory is a parabola:

$$y = \frac{V_{0y}}{V_{0x}}x - \frac{g}{2V_{0x}^2}x^2. \tag{9.71}$$

Likewise, it is easy to show (dashed curve in Figure 9.9) that without friction the range $R = 2V_0^2\,\sin\theta\cos\theta/g$ and the maximum height $H = \frac{1}{2}V_0^2\sin^2\theta/g$.

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 Figure 9.9). 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 $-mg\hat{\mathbf{e}}_y$:

$$\mathbf{F}^{(f)} - mg\hat{\mathbf{e}}_y = m\frac{d^2\mathbf{x}(t)}{dt^2}, \tag{9.72}$$

$$\Rightarrow \quad F_x^{(f)} = m\frac{d^2x}{dt^2}, F_y^{(f)} - mg = m\frac{d^2y}{dt^2}, \tag{9.73}$$

where the bold symbols indicate vector quantities.

The frictional force $\mathbf{F}^{(f)}$ is not a basic force of nature but rather a simple model of a complicated phenomenon. We do know that friction always opposes motion, which means it is in the direction opposite to velocity. One model assumes that the frictional force is proportional to a power n of the projectile's speed [M&T 03]:

$$\mathbf{F}^{(f)} = -k\,m\,|v|^n\,\frac{\mathbf{v}}{|v|}, \tag{9.74}$$

where the $-\mathbf{v}/|v|$ factor ensures that the frictional force is always in a direction opposite that of the velocity. Physical measurements indicate that the power n is noninteger and varies with velocity, and so a more accurate model would be a numerical one that uses the empirical velocity dependence $n(v)$. With a constant power law for friction, the equations of motion are

$$\frac{d^2x}{dt^2} = -k\,v_x^n\,\frac{v_x}{|v|}, \quad \frac{d^2y}{dt^2} = -g - k\,v_y^n\,\frac{v_y}{|v|}, \quad |v| = \sqrt{v_x^2 + v_y^2}. \tag{9.75}$$

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) $n = \frac{3}{2}$, for medium velocities; and (3) $n = 2$ for high velocities.

9.16.1 Simultaneous Second-Order ODEs

Even though (9.75) are simultaneous second-order ODEs, we can still use our regular ODE solver on them after expressing them in standard form

$$\frac{d\mathbf{y}}{dt} = \mathbf{y}(t, \mathbf{y}) \quad \text{(standard form)}. \tag{9.76}$$

We pick $\mathbf{y}$ to be the 4-D vector of dependent variables:

$$y^{(0)} = x(t), \quad y^{(1)} = \frac{dx}{dt}, \quad y^{(2)} = y(t), \quad y^{(3)} = \frac{dy}{dt}. \tag{9.77}$$

We express the equations of motion in terms of $\mathbf{y}$ to obtain the standard form:

$$\frac{dy^{(0)}}{dt}\left(\equiv \frac{dx}{dt}\right) = y^{(1)}, \quad \frac{dy^{(1)}}{dt}\left(\equiv \frac{d^2x}{dt^2}\right) = \frac{1}{m}F_x^{(f)}(\mathbf{y})$$

$$\frac{dy^{(2)}}{dt}\left(\equiv \frac{dy}{dt}\right) = y^{(3)}, \quad \frac{dy^{(3)}}{dt}\left(\equiv \frac{d^2y}{dt^2}\right) = \frac{1}{m}F_y^{(f)}(\mathbf{y}) - g.$$

And now we just read off the components of the force function $\mathbf{f}(t, \mathbf{y})$:

$$f^{(0)} = y^{(1)}, \quad f^{(1)} = \frac{1}{m}F_x^{(f)}, \quad f^{(2)} = y^{(3)}, \quad f^{(3)} = \frac{1}{m}F_y^{(f)} - g.$$

9.16.2 Assessment

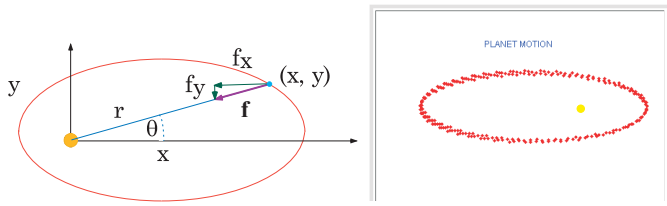
Applet

1. Modify your **`rk4`** program so that it solves the simultaneous ODEs for projectile motion (9.75) with friction ($n = 1$).
2. Check that you obtain graphs similar to those in Figure 9.9.
3. The model (9.74) with $n = 1$ is okay for low velocities. Now modify your program to handle $n = \frac{3}{2}$ (medium-velocity friction) and $n = 2$ (high-velocity friction). Adjust the value of k for the latter two cases such that the initial force of friction $k\,v_0^n$ is the same for all three cases.
4. What is your conclusion about balls falling out of the sky?

9.17 PROBLEM 3: 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 to predict periods of the motion accurately. All

Figure 9.10 *Left:* The gravitational force on a planet a distance r from the sun. The x and y components of the force are indicated. *Right:* Output from the applet `Planet` showing the precession of a planet's orbit when the gravitational force $\propto 1/r^4$ (successive orbits do not lie on top of each other).



Newton needed to postulate was that the force between a planet of mass m and the sun of mass M is

$$F^{(g)} = -\frac{GmM}{r^2}. \tag{9.78}$$

Here r is the planet-CM distance, G is the universal gravitational constant, and the attractive force lies along the line connecting the planet and the sun (Figure 9.10 left). The hard part for Newton was solving the resulting differential equations because he had to invent calculus to do it and then had go through numerous analytic manipulations. The numerical solution is straightforward since even for planets the equation of motion is still

$$\mathbf{f} = m\mathbf{a} = m\frac{d^2\mathbf{x}}{dt^2}, \tag{9.79}$$

with the force (9.78) having components (Figure 9.10)

$$f_x = F^{(g)} \cos\,\theta = F^{(g)}\frac{x}{r}, \tag{9.80}$$

$$f_y = F^{(g)} \sin\,\theta = F^{(g)}\frac{y}{r}, \tag{9.81}$$

$$r = \sqrt{x^2 + y^2}. \tag{9.82}$$

The equation of motion yields two simultaneous second-order ODEs:

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^3}, \quad \frac{d^2y}{dt^2} = -GM\frac{y}{r^3}. \tag{9.83}$$

9.17.1 Implementation: Planetary Motion

1. Assume units such that $GM = 1$ and use the initial conditions

$$x(0) = 0.5, \quad y(0) = 0, \quad v_x(0) = 0.0, \quad v_y(0) = 1.63.$$

2. Modify your ODE solver program to solve (9.83).
3. You may need to make the time step small enough so that the elliptical orbit closes upon itself, as it should, and the number of steps large enough such that the orbits just repeat.
4. Experiment with the initial conditions until you find the ones that produce a circular orbit (a special case of an ellipse).
5. Once you have obtained good precision, note the effect of progressively increasing the initial velocity until the orbits open up and become hyperbolic.
6. Using the same initial conditions that produced the ellipse, investigate the effect of the power in (9.78) being $1/r^4$ rather than $1/r^2$. You should find that the orbital ellipse now rotates or precesses (Figure 9.10). In fact, as you should verify, even a slight variation

from an inverse square power law (as arises from general relativity) causes the orbit to precess.

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 xy plane and that the units are such that $G = 1$.

Chapter Ten

Fourier Analysis; Signals and Filters

In Unit I of this chapter we examine Fourier series and Fourier transforms, two standard tools for decomposing periodic and nonperiodic motions. We find that as implemented for numerical computation, both the series and the integral become a discrete Fourier transform (DFT), which is simple to program. In Unit II we discuss the subject of signal filtering and see that various Fourier tools can be used to reduce noise in measured or simulated signals. In Unit III we present a discussion of the fast Fourier transform (FFT), a technique that is so efficient that it permits evaluations of DFTs in real time.

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Fourier Analysis	pdf	10.1	Discrete Fourier Transform I	pdf	10.4
DFT II	pdf	10.4	Filters	pdf	10.5
DFT Aliasing	pdf	10.4.2	Fast Fourier Transform	pdf	10.8

10.1 UNIT I. FOURIER ANALYSIS OF NONLINEAR OSCILLATIONS

Consider a particle oscillating in the nonharmonic potential of equation (9.4):

$$V(x) = \frac{1}{p}k|x|^p, \tag{10.1}$$

for $p \neq 2$, or for the perturbed harmonic oscillator (9.2),

$$V(x) = \frac{1}{2}kx^2\left(1 - \frac{2}{3}\alpha x\right). \tag{10.2}$$

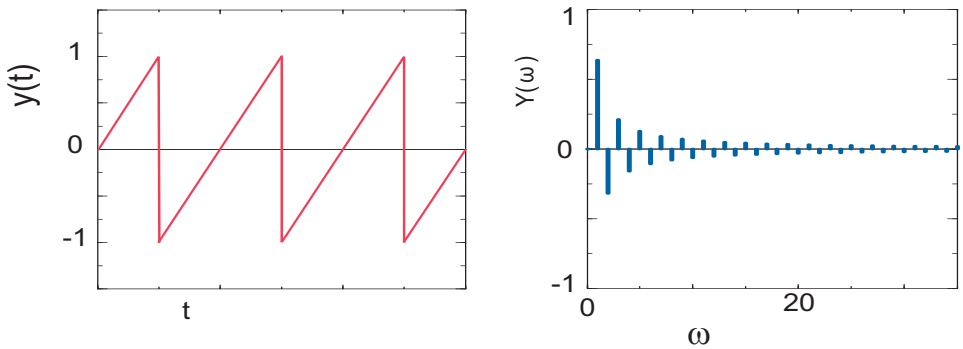
While free oscillations in these potentials are always periodic, they are not truly sinusoidal. Your **problem** is to take the solution of one of these nonlinear oscillators and relate it to the solution

$$x(t) = A_0 \sin(\omega t + \phi_0) \tag{10.3}$$

of the linear harmonic oscillator. If your oscillator is sufficiently nonlinear to behave like the sawtooth function (Figure 10.1 left), then the Fourier spectrum you obtain should be similar to that shown on the right in Figure 10.1.

In general, when we 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 out. It is easy to identify just what the initial transient is for linear systems but may be less so for nonlinear systems in which the "steady state" jumps among a number of configurations.

Figure 10.1 *Left:* A sawtooth function that repeats infinitely in time. *Right:* The Fourier spectrum of frequencies contained in this function (natural units).



10.2 FOURIER SERIES (MATH)

Part of our interest in nonlinear oscillations arises from their lack of study in traditional physics courses even though linear oscillations are just the first approximation to a naturally oscillatory 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 $p \simeq 10$ that produces an $x(t)$ looking like a series of pyramids; this motion is periodic but not harmonic.

In numerical analysis there really is no distinction between a Fourier integral and a Fourier series because the integral is always approximated as a finite series. We will illustrate both methods. 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 by perturbation theory [L&L,M 76]. 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 linear *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 in the series

$$y(t) = b_0 \sin \omega_0 t + b_1 \sin 2\omega_0 t + \cdots . \tag{10.4}$$

Anharmonic oscillators vibrate at a single frequency (which may vary with amplitude) but not with a sinusoidal waveform. Expanding the oscillations in a Fourier series does not imply that the individual harmonics can be excited (played).

You may recall from classical mechanics that the general solution for a vibrating 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 $y_1(t)$ and $y_2(t)$ are solutions of some linear equation, then $\alpha_1 y_1(t) + \alpha_2 y_2(t)$ 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 with frequencies that are integer multiples of the true frequency of the nonlinear oscillator.[1] 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

$$y(t + T) = y(t). \tag{10.5}$$

[1] We remind the reader that every periodic system by definition has a period T and consequently a true frequency ω. Nonetheless, this does not imply that the system behaves like $\sin \omega t$. Only harmonic oscillators do that.

This tells us the *true frequency* ω:

$$\omega \equiv \omega_1 = \frac{2\pi}{T}. \tag{10.6}$$

A periodic function (usually called the *signal*) can be expanded as a series of harmonic functions with frequencies that are multiples of the true frequency:

$$y(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos n\omega t + b_n \sin n\omega t\right). \tag{10.7}$$

This equation represents the signal $y(t)$ as the simultaneous sum of pure tones of frequency $n\omega$. The coefficients a_n and b_n 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 is proportional to $a_n^2 + b_n^2$.

The Fourier series (10.7) is a best fit in the least-squares sense of Chapter 8, "Solving Systems of Equations with Matrices; Data Fitting," because it minimizes $\sum_i [y(t_i) - y_i]^2$, where i denotes different measurements of the signal. 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 low-accuracy reproduction 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, multiply both sides of (10.7) by $\cos n\omega t$ or $\sin n\omega t$, integrate over one period, and project a single a_n or b_n:

$$\begin{pmatrix} a_n \\ b_n \end{pmatrix} = \frac{2}{T} \int_0^T dt \begin{pmatrix} \cos n\omega t \\ \sin n\omega t \end{pmatrix} y(t), \quad \omega \stackrel{\text{def}}{=} \frac{2\pi}{T}. \tag{10.8}$$

As seen in the b_n coefficients (Figure 10.1 right), these coefficients usually decrease in magnitude as the frequency increases and can occur with a negative sign, the negative sign indicating relative 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:

$$a_0 = 2\left\langle y(t) \right\rangle. \tag{10.9}$$

- For an *odd function*, that is, one for which $y(-t) = -y(t)$, all of the coefficients $a_n \equiv 0$ and only half of the integration range is needed to determine b_n:

$$b_n = \frac{4}{T} \int_0^{T/2} dt\, y(t) \sin n\omega t. \tag{10.10}$$

  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:

$$a_n = \frac{4}{T} \int_0^{T/2} dt\, y(t) \cos n\omega t. \tag{10.11}$$

10.2.1 Example 1: Sawtooth Function

The sawtooth function (Figure 10.1) is described mathematically as

$$y(t) = \begin{cases} \frac{t}{T/2}, & \text{for } 0 \le t \le \frac{T}{2}, \\ \frac{t-T}{T/2}, & \text{for } \frac{T}{2} \le t \le T. \end{cases} \tag{10.12}$$

It is clearly periodic, nonharmonic, and discontinuous. Yet it is also odd and so can be represented more simply by shifting the signal to the left:

$$y(t) = \frac{t}{T/2}, \quad -\frac{T}{2} \le t \le \frac{T}{2}. \tag{10.13}$$

Even though the general shape of this function can be reproduced with only a few terms of the Fourier components, many components are needed to reproduce the sharp corners. Because the function is odd, the Fourier series is a sine series and (10.8) determines the values

$$b_n = \frac{2}{T}\int_{-T/2}^{+T/2} dt\, \sin n\omega t \frac{t}{T/2} = \frac{\omega}{\pi}\int_{-\pi/\omega}^{+\pi/\omega} dt\, \sin n\omega t \frac{\omega t}{\pi} = \frac{2}{n\pi}(-1)^{n+1},$$

$$\Rightarrow \quad y(t) = \frac{2}{\pi}\left[\sin\omega t - \frac{1}{2}\sin 2\omega t + \frac{1}{3}\sin 3\omega t - \cdots\right]. \tag{10.14}$$

10.2.2 Example 2: Half-wave Function

The half-wave function

$$y(t) = \begin{cases} \sin\omega t, & \text{for } 0 < t < T/2, \\ 0, & \text{for } T/2 < t < T, \end{cases} \tag{10.15}$$

is periodic, nonharmonic (the upper half of a sine wave), and continuous, but with discontinuous derivatives. Because it lacks the sharp corners of the sawtooth function, it is easier to reproduce with a finite Fourier series. Equation (10.8) determines

$$a_n = \begin{cases} \frac{-2}{\pi(n^2-1)}, & n \text{ even or } 0, \\ 0, & n \text{ odd}, \end{cases} \qquad b_n = \begin{cases} \frac{1}{2}, & n = 1, \\ 0, & n \ne 1, \end{cases}$$

$$\Rightarrow \quad y(t) = \frac{1}{2}\sin\omega t + \frac{1}{\pi} - \frac{2}{3\pi}\cos 2\omega t - \frac{2}{15\pi}\cos 4\omega t + . \tag{10.16}$$

10.3 EXERCISE: SUMMATION OF FOURIER SERIES

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?)

10.4 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 convert from series to transform by imagining a system described by a continuum of "fundamental" frequencies. We thereby deal with *wave packets* containing continuous rather than discrete frequencies.[2] While the difference between series and transform methods may appear clear mathematically, when we approximate the Fourier integral as a finite sum, the two become equivalent.

By analogy with (10.7), we now imagine our function or signal $y(t)$ expressed in terms of a continuous series of harmonics (*inverse Fourier transform*):

$$y(t) = \int_{-\infty}^{+\infty} d\omega\, Y(\omega)\, \frac{e^{i\omega t}}{\sqrt{2\pi}}, \tag{10.17}$$

where for compactness we use a complex exponential function.[3] The expansion amplitude $Y(\omega)$ is analogous to the Fourier coefficients (a_n, b_n) and is called the *Fourier transform* of $y(t)$. The integral (10.17) is the inverse transform since it converts the transform to the signal. The *Fourier transform* converts $y(t)$ to its transform $Y(\omega)$:

$$Y(\omega) = \int_{-\infty}^{+\infty} dt\, \frac{e^{-i\omega t}}{\sqrt{2\pi}} y(t). \tag{10.18}$$

The $1/\sqrt{2\pi}$ factor in both these integrals is a common normalization in quantum mechanics but maybe not in engineering where only a single $1/2\pi$ factor is used. Likewise, the signs in the exponents are also conventions that do not matter as long as you maintain consistency.

If $y(t)$ is the measured response of a system (signal) as a function of time, then $Y(\omega)$ is the *spectral function* that measures the amount of frequency ω present in the signal. [However, some experiments may measure $Y(\omega)$ directly, in which case an inverse transform is needed to obtain $y(t)$.] In many cases $Y(\omega)$ is a complex function with positive and negative values and with significant variation in magnitude. Accordingly, it is customary to eliminate some of the complexity of $Y(\omega)$ by making a semilog plot of the squared modulus $|Y(\omega)|^2$ *versus* ω. This is called a *power spectrum* and provides you with an immediate view of the amount of power or strength in each component.

If the Fourier transform and its inverse are consistent with each other, we should be able to substitute (10.17) into (10.18) and obtain an identity:

$$Y(\omega) = \int_{-\infty}^{+\infty} dt\, \frac{e^{-i\omega t}}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} d\omega'\, \frac{e^{i\omega' t}}{\sqrt{2\pi}} Y(\omega') = \int_{-\infty}^{+\infty} d\omega' \left\{ \int_{-\infty}^{+\infty} dt\, \frac{e^{i(\omega'-\omega)t}}{2\pi} \right\} Y(\omega').$$

For this to be an identity, the term in braces must be the *Dirac delta function*:

$$\int_{-\infty}^{+\infty} dt\, e^{i(\omega'-\omega)t} = 2\pi\delta(\omega' - \omega). \tag{10.19}$$

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 misbehaves terribly in a computational sense. While it is possible to create numerical approximations to $\delta(\omega' - \omega)$, they may well

[2] We follow convention and consider time t the function's variable and frequency ω the transform's variable. Nonetheless, these can be reversed or other variables such as position x and wave vector k may also be used.

[3] Recollect 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.

be borderline pathological. It is certainly better for you to do the delta function part of an integration analytically and leave the nonsingular leftovers to the computer.

10.4.1 Discrete Fourier Transform Algorithm

If $y(t)$ or $Y(\omega)$ is known analytically or numerically, the integral (10.17) or (10.18) can be evaluated using the integration techniques studied earlier. In practice, the signal $y(t)$ is measured at just a finite number N of times t, and these are what we must use to approximate the transform. The resultant *discrete Fourier transform* is an approximation both because the signal is not known for all times and because we integrate numerically.[4] Once we have a discrete set of transforms, they can be used to reconstruct the signal for any value of the time. In this way the DFT can be thought of as a technique for interpolating, compressing, and extrapolating data.

We assume that the signal $y(t)$ is sampled at $(N+1)$ discrete times (N time intervals), with a constant spacing h between times:

$$y_k \stackrel{\text{def}}{=} y(t_k), \qquad k = 0, 1, 2, \ldots, N, \tag{10.20}$$

$$t_k \stackrel{\text{def}}{=} kh, \qquad h = \Delta t. \tag{10.21}$$

In other words, we measure the signal $y(t)$ once every hth of a second for a total time T. This corresponds to period T and *sampling rate* s:

$$T \stackrel{\text{def}}{=} Nh, \quad s = \frac{N}{T} = \frac{1}{h}. \tag{10.22}$$

Regardless of the actual periodicity of the signal, when we choose a period T over which to sample, the mathematics produces a $y(t)$ that is periodic with period T,

$$y(t+T) = y(t). \tag{10.23}$$

We recognize this periodicity, and ensure that there are only N independent measurements used in the transform, by requiring the first and last y's to be the same:

$$y_0 = y_N. \tag{10.24}$$

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 the variation of $y(t)$. Consequently, it determines the lowest frequency,

$$\omega_1 = \frac{2\pi}{T}, \tag{10.25}$$

contained in our Fourier representation of $y(t)$. The frequencies ω_n are determined by the number of samples taken and by the total sampling time $T = Nh$ as

$$\omega_n = n\omega_1 = n\frac{2\pi}{Nh}, \quad n = 0, 1, \ldots, N. \tag{10.26}$$

Here $\omega_0 = 0$ corresponds to the zero-frequency or DC component.

[4] More discussion can be found in [B&H 95], which is devoted to just this topic.

The DFT algorithm results from (1) evaluating the integral in (10.18) not from $-\infty$ to $+\infty$ but rather from time 0 to time T over which the signal is measured, and from (2) using the trapezoid rule for the integration[5]

$$Y(\omega_n) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} dt\, \frac{e^{-i\omega_n t}}{\sqrt{2\pi}} y(t) \simeq \int_0^T dt\, \frac{e^{-i\omega_n t}}{\sqrt{2\pi}} y(t), \tag{10.27}$$

$$\simeq \sum_{k=1}^{N} h\, y(t_k) \frac{e^{-i\omega_n t_k}}{\sqrt{2\pi}} = h \sum_{k=1}^{N} y_k \frac{e^{-2\pi i k n/N}}{\sqrt{2\pi}}. \tag{10.28}$$

To keep the final notation more symmetric, the step size h is factored from the transform Y and a discrete function Y_n is defined:

$$Y_n \stackrel{\text{def}}{=} \frac{1}{h} Y(\omega_n) = \sum_{k=1}^{N} y_k \frac{e^{-2\pi i k n/N}}{\sqrt{2\pi}}. \tag{10.29}$$

With this same care in accounting, and with $d\omega \to 2\pi/Nh$, we invert the Y_n's:

$$y(t) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} d\omega\, \frac{e^{i\omega t}}{\sqrt{2\pi}} Y(\omega) \simeq \sum_{n=1}^{N} \frac{2\pi}{Nh} \frac{e^{i\omega_n t}}{\sqrt{2\pi}} Y(\omega_n). \tag{10.30}$$

Once we know the N values of the transform, we can use (10.30) to evaluate $y(t)$ for any time t. There is nothing illegal about evaluating 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:

$$y(t_{k+N}) = y(t_k), \quad Y(\omega_{n+N}) = Y(\omega_n). \tag{10.31}$$

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.

We see from (10.26) that the larger we make the time $T = Nh$ over which we sample the function, the smaller will be the frequency steps or resolution.[6] Accordingly, if you want a smooth frequency spectrum, you need to have a small frequency step $2\pi/T$. This means you need a large value for the total observation time T. While the best approach would be to measure the input signal for longer times, in practice a measured signal $y(t)$ is often extended in time ("padded") by adding zeros for times beyond the last measured signal, which thereby increases the value of T. Although this does not add new information to the analysis, it does build in the experimentalist's belief that the signal has no existence at times after the measurements are stopped.

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. 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 DFT is an excellent way of obtaining Fourier 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, correspondingly, for the lowest frequencies.

The discrete Fourier transform and its inverse can be written in a concise and insightful way, and be evaluated efficiently, by introducing a complex variable Z for the exponential and

[5] 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 $h/2$ terms have been added to produce one h term.

[6] See also §10.4.2 where we discuss the related phenomenon of aliasing.

then raising Z to various powers:

$$y_k = \frac{\sqrt{2\pi}}{N} \sum_{n=1}^{N} Z^{-nk} Y_n, \quad Z = e^{-2\pi i/N}, \tag{10.32}$$

$$Y_n = \frac{1}{\sqrt{2\pi}} \sum_{k=1}^{N} Z^{nk} y_k, \quad n = 0, 1, \ldots, N, \tag{10.33}$$

where $Z^{nk} \equiv [(Z)^n]^k$. With this formulation, the computer needs to compute only powers of Z. We give our DFT code in Listing 10.1. If your preference is to avoid complex numbers, we can rewrite (10.32) in terms of separate real and imaginary parts by applying Euler's theorem:

$$Z = e^{-i\theta}, \quad \Rightarrow \quad Z^{\pm nk} = e^{\mp ink\theta} = \cos nk\theta \mp i \sin nk\theta, \tag{10.34}$$

where $\theta \stackrel{\text{def}}{=} 2\pi/N$. In terms of the explicit real and imaginary parts,

$$\begin{aligned} Y_n = \frac{1}{\sqrt{2\pi}} \sum_{k=1}^{N} & [(\cos(nk\theta)\text{Re}\, y_k + \sin(nk\theta)\, \text{Im}\, y_k \\ & + \mathbf{i}(\cos(nk\theta)\, \text{Im}\, y_k - \sin(nk\theta)\text{Re}\, y_k)]\,, \end{aligned} \tag{10.35}$$

$$\begin{aligned} y_k = \frac{\sqrt{2\pi}}{N} \sum_{n=1}^{N} & [(\cos(nk\theta)\, \text{Re}\, Y_n - \sin(nk\theta)\text{Im}\, Y_n \\ & + \mathbf{i}(\cos(nk\theta)\text{Im}\, Y_n + \sin(nk\theta)\, \text{Re}\, Y_n\,)]\,. \end{aligned} \tag{10.36}$$

Readers new to DFTs are often surprised when they apply these equations to practical situations and end up with transforms Y having imaginary parts, even though the signal y is real. Equation (10.35) shows that a real signal ($\text{Im}\, y_k \equiv 0$) will yield an imaginary transform unless $\sum_{k=1}^{N} \sin(nk\theta)\, \text{Re}\, y_k = 0$. This occurs only if $y(t)$ is an *even* function over $-\infty \leq t \leq +\infty$ *and* we integrate exactly. Because neither condition holds, the DFTs of real, even functions may have small imaginary parts. This is not due to an error in programming and in fact is a good measure of the approximation error in the entire procedure.

The computation time for a discrete Fourier transform can be reduced even further by use of the *fast Fourier transform* algorithm. An examination of (10.32) shows that the DFT is evaluated as a matrix multiplication of a vector of length N containing the Z values by a vector of length N of y value. The time for this DFT scales like N^2, while the time for the FFT algorithm scales as $N \log_2 N$. Although this may not seem like much of a difference, for $N = 10^{2-3}$, the difference of 10^{3-5} is the difference between a minute and a week. For this reason, FFT is often used for on-line analysis of data. We discuss FFT techniques in §10.8.

Listing 10.1 **DFTcomplex.py** uses the built-in complex numbers of Python to compute the discrete Fourier transform for the signal in method `f( )`.

```
# DFTcomplex.py:  Discrete Fourier Transform, using built in complex

from visual.graph import *
import cmath                                    # for complex math functions

signgr = gdisplay(x=0, y=0, width=600, height=250, title ='Signal',\
              xtitle='x', ytitle = 'signal', xmax = 2.*math.pi, xmin = 0,\
              ymax = 30, ymin = 0)
sigfig = gcurve(color=color.yellow, display=signgr)
imagr = gdisplay(x=0,y=250,width=600,height=250,title ='Imag Fourier TF',
        xtitle = 'x',ytitle='TF.Imag',xmax=10.,xmin=-1,ymax=100,ymin=-0.2)
impart = gvbars(delta = 0.05, color = color.red, display = imagr)  # thin
```

```
N = 50;                    Np = N                          # Number points
signal = zeros( (N+1), float )
twopi  = 2.*pi;          sq2pi = 1./sqrt(twopi);          h = twopi/N
dftz   = zeros( (Np), complex )                  # sequence complex elements

def f(signal):                                            # signal function
    step = twopi/N;          x = 0.
    for i in range(0, N+1):
       signal[i] = 30*cos(x*x*x*x)
       sigfig.plot(pos = (x, signal[i]))                     # plot signal
       x += step

def fourier(dftz):                                                  # DFT
    for n in range(0, Np):
      zsum = complex(0.0, 0.0)                     # real and imag parts = zero
      for  k in range(0, N):                                  # loop for sums
          zexpo = complex(0, twopi*k*n/N)                 # complex exponent
          zsum += signal[k]*exp(-zexpo)             # Fourier transform core
      dftz[n] = zsum * sq2pi                                        # factor
      if dftz[n].imag != 0:                        # plot if not too small imag
          impart.plot(pos = (n, dftz[n].imag) )                  # plot bars

f(signal);        fourier(dftz)                     # call signal, transform
```

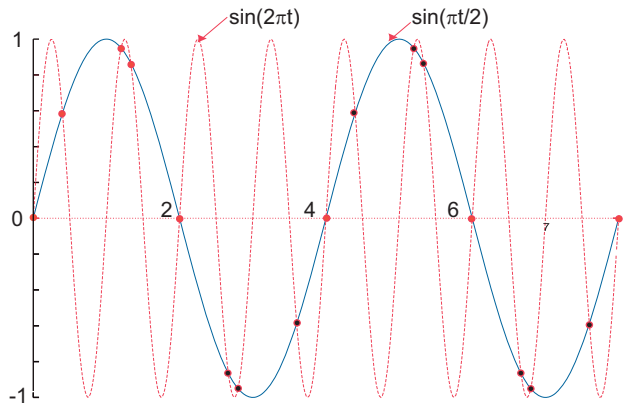
10.4.2 Aliasing(Assessment) ⊙

The sampling of a signal by DFT for only a finite number of times limits the accuracy of the deduced high-frequency components present in the signal. Clearly, good information about very high frequencies requires sampling the signal with small time steps so that all the wiggles can be included. While a poor deduction of the high-frequency components may be tolerable if all we care about are the low-frequency ones, the 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 moir pattern distortion in digital images.

As an example, consider Figure 10.2 showing the two functions $\sin(\pi t/2)$ and $\sin(2\pi t)$ for $0 \le t \le 8$, with their points of overlap in bold. 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 $y \equiv 0$ and assume that there was no signal at all. However, if we were unfortunate enough to measure the signal at the filled dots in Figure 10.2 where $\sin(\pi t/2) = \sin(2\pi t)$, specifically, $t = 0, \frac{12}{10}, \frac{4}{3}, \ldots$, then our Fourier analysis would completely miss the high-frequency component. In DFT jargon, we would say that the high-frequency component 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 separate the high-frequency component properly. In this case some high-frequency signals would be included spuriously as part of the low-frequency spectrum, and this would lead 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 $s \le f/2$. 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, some applications avoid the effects of aliasing by filtering out the high frequencies from the signal and then analyzing the remaining low-frequency part. (The low-frequency *sinc filter* discussed in §10.7.1 is often used for this.) Even though this approach eliminates some high-frequency information, it lessens the distortion of the low-frequency components and so may lead to

Figure 10.2 A plot of the functions $\sin(\pi t/2)$ and $\sin(2\pi t)$. If the sampling rate is not high enough, these signals may appear indistinguishable in a Fourier decomposition. If the sample rate is too low and if both signals are present in a sample, the deduced low-frequency component may be contaminated by the higher-frequency component signal.



improved reproduction of the signal.

If accurate values for the high frequencies are required, then we will need to increase the sampling rate s by increasing the number N of samples taken within the 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.

If we vary the the total time sampling time $T = Nh$ but not the sampling rate $s = N/T = 1/h$, we make ω_1 smaller because the discrete frequencies

$$\omega_n = n\omega_1 = n\frac{2\pi}{T} \tag{10.37}$$

are measured in steps of ω_1. This leads to a smoother frequency spectrum. However, to keep the time step the same and thus not lose high-frequency information, we would also need to increase the number of N samples. And as we said, this is often done, after the fact, by padding the end of the data set with zeros.

Listing 10.2 **DFTreal.py** computes the discrete Fourier transform for the signal in method `f( )` using real numbers.

```
# DFTreal.py:  Discrete Fourier Transform using real numbers

from visual.graph import *

# for the original signal
signgr = gdisplay(x=0,y=0,width=600,height=250,
          title='Original signal y(t)= 3 cos(wt)+2 cos(3wt)+ cos(5wt) ',\
       xtitle='x', ytitle='signal',xmax=2.*math.pi,xmin=0,ymax=7,ymin=-7)
sigfig = gcurve(color=color.yellow,display=signgr)
# For the imaginary part of the transform
imagr = gdisplay(x=0,y=250,width=600,height=250,\
                 title='Fourier transform imaginary part',xtitle='x',\
                 ytitle='Transf.Imag',xmax=10.0,xmin=-1,ymax=20,ymin=-70)
impart = gvbars(delta=0.05,color=color.red,display=imagr)      # thin bars
# for the real part of the transform
# # for you to do
N = 10                                    # points for signal and transform
Np = N                                                  # global constants
signal = zeros((N+1),float)
twopi = 2.*pi
sq2pi = 1./sqrt(twopi)
h = twopi/N
dftimag = zeros((Np),float)                # contains im. part of transform
```

```
def f(signal):                                          # initial function
    step = twopi/N
    t= 0.
    for  i in range(0,N+1):
       signal[i] = 3*sin(t*t*t)
       sigfig.plot(pos=(t,signal[i]))                       # plot function
       t += step

def fourier(dftimag):                            # Discrete Fourier Transform
    for n in range(0,Np):                                  # over frequency
      imag = 0.                                          # reset  variables
      for  k in range(0, N):                                # loop for sums
        imag += signal[k]*sin((twopi*k*n)/N)
      dftimag[n] = -imag*sq2pi                        # imag. part transform
      if dftimag[n] !=0:                   # to plot if not too small trnsf
          impart.plot(pos=(n,dftimag[n]))                        # plot bars

f(signal)
fourier(dftimag)
print ("hola")
for i in range(0,N):
        if abs(dftimag[i])>5:
            print ( "i=",i , "dftimag ", dftimag[i])
```

10.4.3 Fourier Series DFT (Algorithm)

For simplicity let us consider the Fourier cosine series:

$$y(t) = \sum_{n=0}^{\infty} a_n \cos(n\omega t), \quad a_k = \frac{2}{T}\int_0^T dt \cos(k\omega t) y(t). \tag{10.38}$$

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 a small amplitude). We assume that the function $y(t)$ is sampled for a discrete set of times

$$y(t = t_k) \equiv y_k, \quad k = 0, 1, \ldots, N. \tag{10.39}$$

Because we are analyzing a periodic function, we 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:

$$y_0 = y_N. \tag{10.40}$$

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 (10.38), we determine the N independent Fourier components as

$$a_n \simeq \frac{2h}{T}\sum_{k=1}^{N} \cos(n\omega t_k)\, y(t_k) = \frac{2}{N}\sum_{k=1}^{N} \cos\left(\frac{2\pi nk}{N}\right) y_k, \quad n = 0, \ldots, N. \tag{10.41}$$

Because we can determine only N Fourier components from N independent $y(t)$ values, our Fourier series for the $y(t)$ must be in terms of only these components:

$$y(t) \simeq \sum_{n=0}^{N} a_n \cos(n\omega t) = \sum_{n=0}^{N} a_n \cos\left(\frac{2\pi nt}{Nh}\right). \tag{10.42}$$

In summary, we sample the function $y(t)$ at N times, $t_1, \ldots, t_N$. We see that all the 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 wavelet analysis in Chapter 11,

"Wavelet Analysis & Data Compression," the theory is reformulated so that additional samplings determine higher Fourier components without affecting lower ones.

10.4.4 Assessments

Simple analytic input:It is always good to do these simple checks 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; then check that they are essentially real and in the ratio 3:2:1 (or 9:4:1 for the power spectrum), that the frequencies have the expected values (not just ratios), and that the components resum to give the input signal.
2. Experiment on the separate effects of picking different values of the step size h and of enlarging the measurement period $T = Nh$.
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; then check that they are essentially imaginary and in the ratio 1:2:3 (or 1:4:9 if a power spectrum is plotted) and that they resum to give the input signal.
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; then check that there are three of them in the ratio 5:2:1 (or 25:4:1 if a power spectrum is plotted) and that they resum to give the input signal.
5. Sample the signal
$$y(t) = 5 + 10\sin(t + 2).$$
Compare and explain the results obtained by sampling (a) without the 5, (b) as given but without the 2, and (c) without the 5 and without the 2.
6. In our discussion of aliasing, we examined Figure 10.2 showing the functions $\sin(\pi t/2)$ and $\sin(2\pi t)$. Sample the function
$$y(t) = \sin\left(\frac{\pi}{2}t\right) + \sin(2\pi t)$$
and explore how aliasing occurs. Explicitly, we know that the true transform contains peaks at $\omega = \pi/2$ and $\omega = 2\pi$. Sample the signal at a rate that leads to aliasing, as well as at a higher sampling rate at which there is no aliasing. Compare the resulting DFTs in each case and check if your conclusions agree with the Nyquist criterion.

Highly nonlinear oscillator: Recall the numerical solution for oscillations of a spring with power $p = 12$ [see (10.1)]. 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 $|b_n/b_1| < 0.1$. Check that resuming the components reproduces the signal.

Nonlinearly perturbed oscillator: Remember the harmonic oscillator with a nonlinear perturbation (9.2):

$$V(x) = \frac{1}{2}kx^2(1 - \frac{2}{3}\alpha x), \quad F(x) = -kx(1 - \alpha x). \tag{10.43}$$

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 the signal to contain "approximately 10% nonlinearity." This being the case, fix your value of α 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 α fixed.
2. Decompose your numerical solution into a discrete Fourier spectrum.
3. Plot a graph of the percentage of 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 both even and odd components are present, Y_n should be complex. Because a 10% effect in amplitude becomes a 1% effect in power, make sure that you make a semilog plot of the power spectrum.
4. As always, check that resumations of your transforms reproduce the signal.

(*Warning:* The ω you use in your series must correspond to the *true* frequency of the system, not just the ω of small oscillations.)

10.4.5 Nonperiodic Function DFT (Exploration)

Consider a simple model (a wave packet) of a "localized" electron moving through space and time. A good model for an electron initially localized around $x = 5$ is a Gaussian multiplying a plane wave:

$$\psi(x, t = 0) = \exp\left[-\frac{1}{2}\left(\frac{x - 5.0}{\sigma_0}\right)^2\right] e^{ik_0 x}. \tag{10.44}$$

This wave packet is not an eigenstate of the momentum operator[7] $p = id/dx$ and in fact contains a spread of momenta. Your **problem** is to evaluate the Fourier transform,

$$\psi(p) = \int_{-\infty}^{+\infty} dx \frac{e^{ipx}}{\sqrt{2\pi}} \psi(x), \tag{10.45}$$

as a way of determining the momenta components in (10.44).

10.5 UNIT II. FILTERING NOISY SIGNALS

You measure a signal $y(t)$ that obviously contains noise. Your **problem** is to determine the frequencies that would be present in the signal if it did not contain noise. Of course, once you have a Fourier transform from which the noise has been removed, you can transform it to obtain a signal $s(t)$ with no noise.

In the process of solving this problem we examine two simple approaches: the use of autocorrelation functions and the use of filters. Both approaches find wide applications in science, with our discussion not doing the subjects justice. However, we will see filters again in the discussion of wavelets in Chapter 11, "Wavelet Analysis & Data Compression."

[7] We use natural units in which $\hbar = 1$.

10.6 NOISE REDUCTION VIA AUTOCORRELATION (THEORY)

We assume that the measured signal is the sum of the true signal $s(t)$, which we wish to determine, plus the *noise* $n(t)$:

$$y(t) = s(t) + n(t). \tag{10.46}$$

Our first approach to separating the signal from the noise relies on that fact that noise is a random process and thus should not be correlated with the signal. Yet what do we mean when we say that two functions are *correlated?* Well, if the two tend to oscillate with their nodes and peaks in much the same places, then the two functions are clearly correlated. An analytic measure of the correlation of two arbitrary functions $y(t)$ and $x(t)$ is the *correlation function*

$$c(\tau) = \int_{-\infty}^{+\infty} dt\, y^*(t)\, x(t+\tau) \equiv \int_{-\infty}^{+\infty} dt\, y^*(t-\tau)\, x(t), \tag{10.47}$$

where τ, the *lag time*, is a variable. Even if the two signals have different magnitudes, if they have similar time dependences except for one lagging or leading the other, then for certain values of τ the integrand in (10.47) will be positive for all values of t. In this case the two signals interfere constructively and produce a large value for the correlation function. In contrast, if both functions oscillate independently, then it is just as likely for the integrand to be positive as to be negative, in which case the two signals interfere destructively and produce a small value for the integral.

Before we apply the correlation function to our problem, let us study some of its properties. We use (10.17) to express c, y^*, and x in terms of their Fourier transforms:

$$c(\tau) = \int_{-\infty}^{+\infty} d\omega''\, C(\omega'') \frac{e^{i\omega'' t}}{\sqrt{2\pi}}, \quad y^*(t) = \int_{-\infty}^{+\infty} d\omega\, Y^*(\omega) \frac{e^{-i\omega t}}{\sqrt{2\pi}},$$

$$x(t+\tau) = \int_{-\infty}^{+\infty} d\omega'\, X(\omega') \frac{e^{+i\omega t}}{\sqrt{2\pi}}. \tag{10.48}$$

Because ω, ω', and ω'' are dummy variables, other names may be used for these variables without changing any results. When we substitute these representations into the definition (10.47) and assume that the resulting integrals converge well enough to be rearranged, we obtain

$$\int_{-\infty}^{+\infty} d\omega C(\omega) e^{i\omega t} = \int_{-\infty}^{+\infty} \frac{d\omega}{2\pi} \int_{-\infty}^{+\infty} d\omega'\, Y^*(\omega) X(\omega') e^{i\omega\tau} 2\pi\delta(\omega' - \omega)$$

$$= \int_{-\infty}^{+\infty} d\omega Y^*(\omega) X(\omega) e^{i\omega\tau},$$

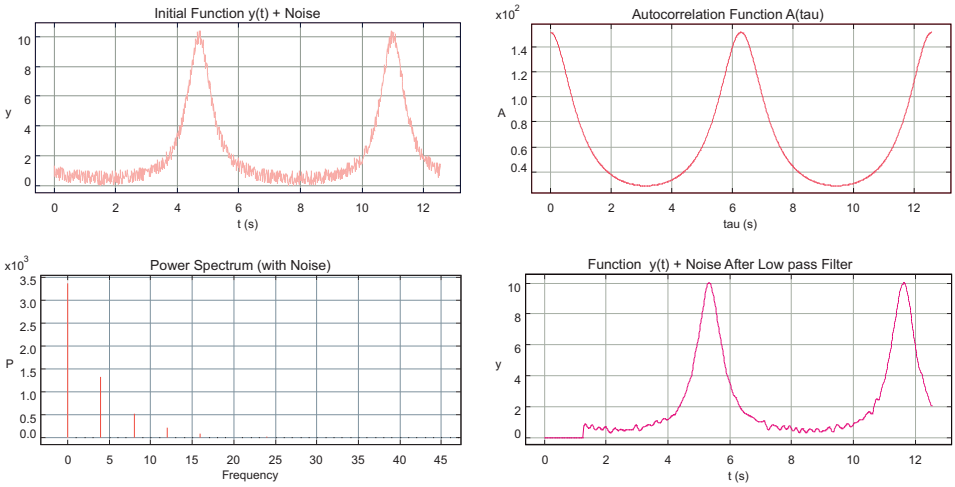
$$\Rightarrow \quad C(\omega) = \sqrt{2\pi}\, Y^*(\omega) X(\omega), \tag{10.49}$$

where the last line follows because ω'' and ω are equivalent dummy variables. Equation (10.49) says that the Fourier transform of the correlation function between two signals is proportional to the product of the transform of one signal and the complex conjugate of the transform of the other. (We shall see a related convolution theorem for filters.)

A special case of the correlation function $c(\tau)$ is the *autocorrelation function* $A(\tau)$. It measures the correlation of a time signal with itself:

$$A(\tau) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} dt\, y^*(t)\, y(t+\tau) \equiv \int_{-\infty}^{+\infty} dt\, y(t)\, y^*(t-\tau). \tag{10.50}$$

Figure 10.3 *From bottom left to right:* A function that is a signal plus noise $s(t) + n(t)$; the autocorrelation function *versus* time deduced by processing this signal; the power spectrum obtained from autocorrelation function; the signal plus noise after passage through a lowpass filter.



This function is computed by taking a signal $y(t)$ that has been measured over some time period and then averaging it over time using $y(t+\tau)$ as a weighting function. This process is also called *folding* a function onto itself (as might be done with dough) or a *convolution*. To see how this folding removes noise from a signal, we go back to the measured signal (10.46), which was the sum of pure signal plus noise $s(t) + n(t)$. As an example, on the upper left in Figure 10.3 we show a signal that was constructed by adding random noise to a smooth signal. When we compute the autocorrelation function for this signal, we obtain a function (upper right in Figure 10.3) that looks like a broadened, smoothed version of the signal $y(t)$. We can understand how the noise is removed by taking the Fourier transform of $s(t) + n(t)$ to obtain a simple sum of transforms:

$$Y(\omega) = S(\omega) + N(\omega), \tag{10.51}$$

$$\left\{ \begin{array}{c} S(\omega) \\ N(\omega) \end{array} \right\} = \int_{-\infty}^{+\infty} dt \left\{ \begin{array}{c} s(t) \\ n(t) \end{array} \right\} \frac{e^{-i\omega t}}{\sqrt{2\pi}}. \tag{10.52}$$

Because the autocorrelation function (10.50) for $y(t) = s(t) + n(t)$ involves the second power of y, is not a linear function, that is, $A_y \neq A_s + A_n$, but instead,

$$A_y(\tau) = \int_{-\infty}^{+\infty} dt \left[s(t)s(t+\tau) + s(t)n(t+\tau) + n(t)n(t+\tau) \right]. \tag{10.53}$$

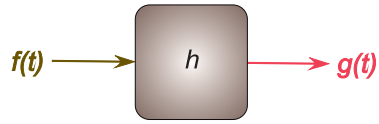
If we assume that the noise $n(t)$ in the measured signal is truly random, then it should average to zero over long times and be uncorrelated at times t and $t+\tau$. This being the case, both integrals involving the noise vanish, and so

$$A_y(\tau) \simeq \int_{-\infty}^{+\infty} dt\, s(t)\, s(t+\tau) = A_s(\tau). \tag{10.54}$$

Thus, the part of the noise that is random tends to be averaged out of the autocorrelation function, and we are left with the autocorrelation function of approximately the pure signal.

This is all very interesting but is not the transform $S(\omega)$ of the pure signal that we need to solve our problem. However, application of (10.49) with $Y(\omega) = X(\omega) = S(\omega)$ tells us

Figure 10.4 A schematic of an input signal *f(t)* passing through a filter *h* that outputs the function *g(t)*.



that the Fourier transform $A(\omega)$ of the autocorrelation function is proportional to $|S(\omega)|^2$:

$$A(\omega) = \sqrt{2\pi}\,|S(\omega)|^2. \tag{10.55}$$

The function $|S(\omega)|^2$ is the *power spectrum* we discussed in §10.4. For practical purposes, knowing the power spectrum is often all that is needed and is easier to understand than a complex $S(\omega)$; in any case it is all that we can calculate.

As a procedure for analyzing data, we (1) start with the noisy measured signal and (2) compute its autocorrelation function $A(t)$ via the integral (10.50). Because this is just folding the signal onto itself, no additional functions or input is needed. We then (3) perform a DFT on the autocorrelation function $A(t)$ to obtain the power spectrum. For example, in Figure 10.3 we see a noisy signal (lower left), the autocorrelation function (lower right), which clearly is smoother than the signal, and finally, the deduced power spectrum (upper left). Notice that the broadband high-frequency components characteristic of noise are absent from the power spectrum. You can easily modify the sample program **`DFTcomplex.py`** in Listing 10.1 to compute the autocorrelation function and then the power spectrum from $A(\tau)$. We present a program **`NoiseSincFilter.py`** in the instructor's manuel that does this.

10.6.1 Autocorrelation Function Exercises

1. Imagine that you have sampled the pure signal

$$s(t) = \frac{1}{1 - 0.9\sin t}. \tag{10.56}$$

Although there is just a single sine function in the denominator, there is an infinite number of overtones as follows from the expansion

$$s(t) \simeq 1 + 0.9\sin t + (0.9\sin t)^2 + (0.9\sin t)^3 + \cdots. \tag{10.57}$$

   a. Compute the DFT $S(\omega)$. Make sure to sample just one period but to cover the entire period. Make sure to sample at enough times (fine scale) to obtain good sensitivity to the high-frequency components.
   b. Make a semilog plot of the power spectrum $|S(\omega)|^2$.
   c. Take your input signal $s(t)$ and compute its autocorrelation function $A(\tau)$ for a full range of τ values (an analytic solution is okay too).
   d. Compute the power spectrum indirectly by performing a DFT on the autocorrelation function. Compare your results to the spectrum obtained by computing $|S(\omega)|^2$ directly.

2. Add some random noise to the signal using a random number generator:

$$y(t_i) = s(t_i) + \alpha(2r_i - 1), \quad 0 \le r_i \le 1, \tag{10.58}$$

where α is an adjustable parameter. Try several values of α, from small values that just add some fuzz to the signal to large values that nearly hide the signal.

   a. Plot your noisy data, their Fourier transform, and their power spectrum obtained directly from the transform with noise.

b. Compute the autocorrelation function $A(t)$ and its Fourier transform.
c. Compare the DFT of $A(\tau)$ to the power spectrum and comment on the effectiveness of reducing noise by use of the autocorrelation function.
d. For what value of α do you essentially lose all the information in the input?

10.7 FILTERING WITH TRANSFORMS (THEORY)

A filter (Figure 10.4) is a device that converts an input signal $f(t)$ to an output signal $g(t)$ with some specific property for the latter. More specifically, an *analog filter* is defined [Hart 98] as integration over the input function:

$$g(t) = \int_{-\infty}^{+\infty} d\tau\, f(\tau)\, h(t-\tau) \stackrel{\text{def}}{=} f(t) * h(t). \tag{10.59}$$

The operation indicated in (10.59) occurs often enough that it is given the name *convolution* and is denoted by an asterisk $*$. The function $h(t)$ is called the *response* or *transfer function* of the filter because it is the response of the filter to a unit impulse:

$$h(t) = \int_{-\infty}^{+\infty} d\tau\, \delta(\tau)\, h(t-\tau). \tag{10.60}$$

Such being the case, $h(t)$ is also called the *unit impulse response function* or *Green's function*. Equation (10.59) states that the output $g(t)$ of a filter equals the input $f(t)$ convoluted with the transfer function $h(t-\tau)$. Because the argument of the response function is delayed by a time τ relative to that of the signal in the integral (10.59), τ is called the *lag time*. While the integration is over all times, the response of a good detector usually peaks around zero time. In any case, the response must equal zero for $\tau > t$ because events in the future cannot affect the present (causality).

The *convolution theorem* states that the Fourier transform of the convolution $g(t)$ is proportional to the product of the transforms of $f(t)$ and $h(t)$:

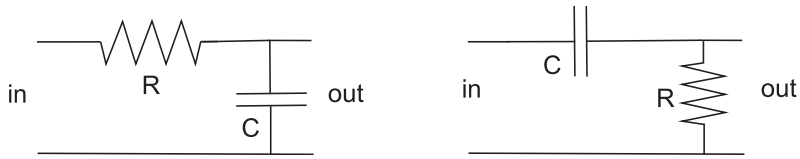
$$G(\omega) = \sqrt{2\pi}\, F(\omega)\, H(\omega). \tag{10.61}$$

The theorem results from expressing the functions in (10.59) by their transforms and using the resulting Dirac delta function to evaluate an integral (essentially what we did in our discussion of correlation functions). This is an example of how some relations are simpler in transform space than in time space.

Regardless of the domain used, filtering as we have defined it is a linear process involving just the first powers of f. This means that the output at one frequency is proportional to the input at that frequency. The constant of proportionality between the two may change with frequency and thus suppress specific frequencies relative to others, but that constant remains fixed in time. Because the law of linear superposition is valid for filters, if the input to a filter is represented as the sum of various functions, then the transform of the output will be the sum of the functions' Fourier transforms. Because the transfer function may be complex, $H(\omega) = |H(\omega)| \exp[i\phi(\omega)]$, the filter may also shift the phase of the input at frequency ω by an amount ϕ.

Filters that remove or decrease high-frequency components more than they do low-frequency components, are called *lowpass* filters. Those that filter out the low frequencies are called *highpass filters*. A simple lowpass filter is the RC circuit on the left in Figure 10.5,

Figure 10.5 *Left:* An *RC* circuit arranged as a lowpass filter. *Right:* An *RC* circuit arranged as a highpass filter.



and it produces the transfer function

$$H(\omega) = \frac{1}{1 + i\omega\tau} = \frac{1 - i\omega\tau}{1 + \omega^2\tau^2}, \tag{10.62}$$

where $\tau = RC$ is the time constant. The ω^2 in the denominator leads to a decrease in the response at high frequencies and therefore makes this a lowpass filter (the $i\omega$ affects only the phase). A simple highpass filter is the RC circuit on the right in Figure 10.5, and it produces the transfer function

$$H(\omega) = \frac{i\omega\tau}{1 + i\omega\tau} = \frac{i\omega\tau + \omega^2\tau^2}{1 + \omega^2\tau^2}. \tag{10.63}$$

$H = 1$ at large ω, yet H vanishes as $\omega \to 0$, which makes this a highpass filter.

Filters composed of resistors and capacitors are fine for analog signal processing. For digital processing we want a *digital filter* that has a specific response function for each frequency range. A physical model for a digital filter may be constructed from a delay line with taps at various spacing along the line (Figure 10.6) [Hart 98]. The signal read from tap n is just the input signal delayed by time $n\tau$, where the delay time τ is a characteristic of the particular filter. The output from each tap is described by the transfer function $\delta(t - n\tau)$, possibly with scaling factor c_n. As represented by the triangle on the right in Figure 10.6, the signals from all taps are ultimately summed together to form the total response function:

$$h(t) = \sum_{n=0}^{N} c_n\, \delta(t - n\tau). \tag{10.64}$$

In the frequency domain, the Fourier transform of a delta function is an exponential, and so (10.64) results in the transfer function

$$H(\omega) = \sum_{n=0}^{N} c_n\, e^{-i\, n\omega\tau}, \tag{10.65}$$

where the exponential indicates the phase shift from each tap.

If a digital filter is given a continuous time signal $f(t)$ as input, its output will be the discrete sum

$$g(t) = \int_{-\infty}^{+\infty} dt'\, f(t') \sum_{n=0}^{N} c_n\, \delta(t - t' - n\tau) = \sum_{n=0}^{N} c_n\, f(t - n\tau). \tag{10.66}$$

And of course, if the signal's input is a discrete sum, its output will remain a discrete sum. (We restrict ourselves to nonrecursive filters [Pres 94].) In either case, we see that knowledge of the filter coefficients c_i provides us with all we need to know about a digital filter. If we look back at our work on the discrete Fourier transform in §10.4.1, we can view a digital filter (10.66) as a Fourier transform in which we use an N-point approximation to the Fourier integral. The c_n's then contain both the integration weights and the values of the response function at the integration points. The transform itself can be viewed as a filter of the signal into specific frequencies.

Figure 10.6 A delay-line filter in which the signal at different times is scaled by different amounts c_i.

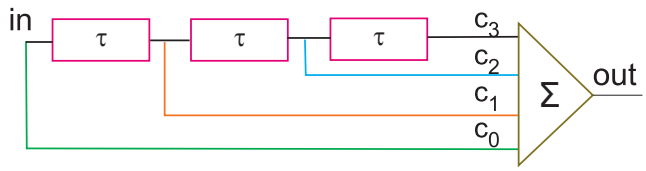
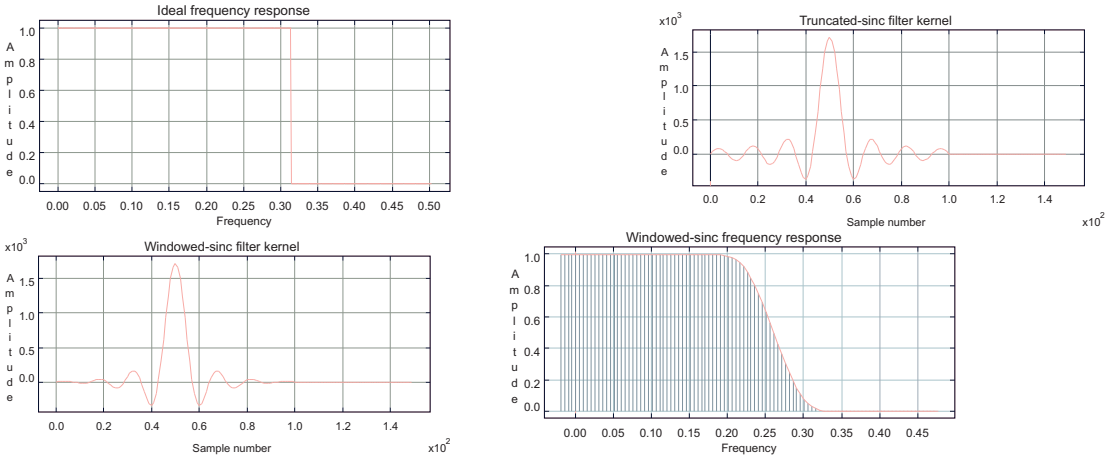


Figure 10.7 *Upper left:* Frequency response of an ideal lowpass filter; only high frequencies are eliminated. *Upper right:* Windowed-sinc filter kernel (frequency domain). *Lower left:* Truncated-sinc filter kernel (time domain). *Lower right:* Frequency response of a windowed-sinc filter is almost an ideal lowpass filterFour .



10.7.1 Digital Filters: Windowed Sinc Filters (Exploration) ⊙

Problem: Construct digital versions of highpass and lowpass filters and determine which filter works better at removing noise from a signal.

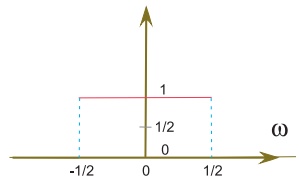
A popular way to separate the bands of frequencies in a signal is with a *windowed sinc filter* [Smi 99]. This filter is based on the observation that an ideal *lowpass* filter passes all frequencies below a cutoff frequency ω_c and blocks all frequencies above this frequency. And because there tends to be more noise at high frequencies than at low frequencies, removing the high frequencies tends to remove more noise than signal, although some signal is inevitably lost. One use for windowed sinc filters is in reducing aliasing by removing the high-frequency component of a signal before determining its Fourier components. The graph on the lower right in Figure 10.7 was obtained by passing our noisy signal through a sinc filter (using the program **`NoiseSincFilter.py`**).

If both positive and negative frequencies are included, an ideal low-frequency filter will look like the rectangular pulse in frequency space:

$$H(\omega, \omega_c) = \text{rect}\left(\frac{\omega}{2\omega_c}\right) \qquad \text{rect}(\omega) = \begin{cases} 1, & \text{if } |\omega| \le \frac{1}{2}, \\ 0, & \text{otherwise.} \end{cases} \tag{10.67}$$

Here $\text{rect}(\omega)$ is the rectangular function (Figure 10.8). Although maybe not obvious, a rectangular pulse in the frequency domain has a Fourier transform that is proportional to the *sinc*

Figure 10.8 The rectangle function rect(ω) that is constant for a finite frequency interval. The Fourier transform of this function is sinc(t).



function in the time domain [Smi 91, Wiki]

$$\int_{-\infty}^{+\infty} d\omega\, e^{-i\omega t}\text{rect}(\omega) = \text{sinc}\left(\frac{t}{2}\right) \stackrel{\text{def}}{=} \frac{\sin(\pi t/2)}{\pi t/2}, \tag{10.68}$$

where the π's are sometimes omitted. Consequently, we can filter out the high-frequency components of a signal by convoluting it with $\sin(\omega_c t)/(\omega_c t)$, a technique also known as the *Nyquist–Shannon* interpolation formula. In terms of discrete transforms, the time-domain representation of the sinc filter is

$$h[i] = \frac{\sin(\omega_c i)}{i\pi}. \tag{10.69}$$

All frequencies below the cutoff frequency ω_c are passed with unit amplitude, while all higher frequencies are blocked.

In practice, there are a number of problems in using this function as the filter. First, as formulated, the filter is *noncausal*; that is, there are coefficients at negative times, which is nonphysical because we do not start measuring the signal until $t = 0$. Second, in order to produce a perfect rectangular response, we would have to sample the signal at an infinite number of times. In practice, we sample at $(M + 1)$ points (M even) placed symmetrically around the main lobe of $\sin(\pi t)/\pi t$ and then shift times to purely positive values,

$$h[i] = \frac{\sin[2\pi\omega_c(i - M/2)]}{i - M/2}, \quad 0 \le t \le M. \tag{10.70}$$

As might be expected, a penalty is incurred for making the filter discrete; instead of the ideal rectangular response, we obtain a *Gibbs overshoot*, with rounded corners and oscillations beyond the corner.

There are two ways to reduce the departures from the ideal filter. The first is to increase the length of times for which the filter is sampled, which inevitably leads to longer compute times. The other way is to smooth out the truncation of the sinc function by multiplying it with a smoothly tapered curve like the *Hamming window function*:

$$w[i] = 0.54 - 0.46\ \cos(2\pi i/M). \tag{10.71}$$

In this way the filter's kernel becomes

$$h[i] = \frac{\sin[2\pi\omega_c(i - M/2)]}{i - M/2}\left[0.54 - 0.46\ \cos(\frac{2\pi i}{M})\right]. \tag{10.72}$$

The cutoff frequency ω_c should be a fraction of the sampling rate. The time length M determines the *bandwidth* over which the filter changes from 1 to 0.

Exercise: Repeat the exercise that added random noise to a known signal, this time using the sinc filter to reduce the noise. See how small you can make the signal and still be able to separate it from the noise. ▌

10.8 UNIT II. FAST FOURIER TRANSFORM ALGORITHM ⊙

We have seen in (10.32) that a discrete Fourier transform can be written in the compact form

$$Y_n = \frac{1}{\sqrt{2\pi}} \sum_{k=1}^{N} Z^{nk}\, y_k, \quad Z = e^{-2\pi i/N}, \quad n = 0, 1, \ldots, N-1. \tag{10.73}$$

Even if the signal elements y_k to be transformed are real, Z is always complex, and therefore we must process both real and imaginary parts when computing transforms. Because both n and k range over N integer values, the $(Z^n)^k\, y_k$ multiplications in (10.73) require some N^2 multiplications and additions of complex numbers. As N gets large, as happens in realistic applications, this geometric increase in the number of steps slows down the algorithm.

In 1965, Cooley and Tukey discovered an algorithm[8] that reduces the number of operations necessary to perform a DFT from N^2 to roughly $N \log_2 N$ [Co,65, Donn 05]. Even though this may not seem like such a big difference, it represents a 100-fold speedup for 1000 data points, which changes a full day of processing into 15 min of work. Because of its widespread use (including cell phones), the fast Fourier transform algorithm is considered one of the 10 most important algorithms of all time.

The idea behind the FFT is to utilize the periodicity inherent in the definition of the DFT (10.73) to reduce the total number of computational steps. Essentially, the algorithm divides the input data into two equal groups and transforms only one group, which requires $\sim (N/2)^2$ multiplications. It then divides the remaining (nontransformed) group of data in half and transforms them, continuing the process until all the data have been transformed. The total number of multiplications required with this approach is approximately $N \log_2 N$.

Specifically, the FFT's time economy arises from the computationally expensive complex factor $Z^{nk}[= ((Z)^n)^k]$ being equal to the same cyclically repeated value as the integers n and k vary sequentially. For instance, for $N = 8$,

$$\begin{aligned}
Y_0 &= Z^0 y_0 + Z^0 y_1 + Z^0\, y_2 + Z^0\, y_3 + Z^0\, y_4 + Z^0\, y_5 + Z^0\, y_6 + Z^0\, y_7,\\
Y_1 &= Z^0 y_0 + Z^1 y_1 + Z^2 y_2 + Z^3\, y_3 + Z^4\, y_4 + Z^5\, y_5 + Z^6\, y_6 + Z^7\, y_7,\\
Y_2 &= Z^0 y_0 + Z^2 y_1 + Z^4\, y_2 + Z^6\, y_3 + Z^8\, y_4 + Z^{10} y_5 + Z^{12} y_6 + Z^{14} y_7,\\
Y_3 &= Z^0 y_0 + Z^3 y_1 + Z^6\, y_2 + Z^9\, y_3 + Z^{12} y_4 + Z^{15} y_5 + Z^{18} y_6 + Z^{21} y_7,\\
Y_4 &= Z^0 y_0 + Z^4 y_1 + Z^8\, y_2 + Z^{12} y_3 + Z^{16} y_4 + Z^{20} y_5 + Z^{24} y_6 + Z^{28} y_7,\\
Y_5 &= Z^0 y_0 + Z^5 y_1 + Z^{10} y_2 + Z^{15} y_3 + Z^{20} y_4 + Z^{25} y_5 + Z^{30} y_6 + Z^{35} y_7,\\
Y_6 &= Z^0 y_0 + Z^6 y_1 + Z^{12} y_2 + Z^{18} y_3 + Z^{24} y_4 + Z^{30} y_5 + Z^{36} y_6 + Z^{42} y_7,\\
Y_7 &= Z^0 y_0 + Z^7 y_1 + Z^{14} y_2 + Z^{21} y_3 + Z^{28} y_4 + Z^{35} y_5 + Z^{42} y_6 + Z^{49} y_7,
\end{aligned}$$

where we include $Z^0(\equiv 1)$ for clarity. When we actually evaluate these powers of Z, we find

[8] Actually, this algorithm has been discovered a number of times, for instance, in 1942 by Danielson and Lanczos [Da,42], as well as much earlier by Gauss.

only four independent values:

$$
\begin{aligned}
&Z^0 = \exp(0) = +1, && Z^1 = \exp(-\tfrac{2\pi}{8}i) = +\tfrac{\sqrt{2}}{2} - i\,\tfrac{\sqrt{2}}{2},\\
&Z^2 = \exp(-\tfrac{2\pi}{8}2i) = -i, && Z^3 = \exp(-\tfrac{2\pi}{8}3i) = -\tfrac{\sqrt{2}}{2} - i\,\tfrac{\sqrt{2}}{2},\\
&Z^4 = \exp(-\tfrac{2\pi}{8}4i) = -Z^0, && Z^5 = \exp(-\tfrac{2\pi}{8}5i) = -Z^1,\\
&Z^6 = \exp(-\tfrac{2\pi}{8}6i) = -Z^2, && Z^7 = \exp(-\tfrac{2\pi}{8}7i) = -Z^3,\\
&Z^8 = \exp(-\tfrac{2\pi}{8}8i) = +Z^0, && Z^9 = \exp(-\tfrac{2\pi}{8}9i) = +Z^1,\\
&Z^{10} = \exp(-\tfrac{2\pi}{8}10i) = +Z^2, && Z^{11} = \exp(-\tfrac{2\pi}{8}11i) = +Z^3,\\
&Z^{12} = \exp(-\tfrac{2\pi}{8}11i) = -Z^0, && \cdots .
\end{aligned}
\tag{10.74}
$$

When substituted into the definitions of the transforms, we obtain

$$
\begin{aligned}
Y_0 &= Z^0y_0 + Z^0y_1 + Z^0y_2 + Z^0y_3 + Z^0y_4 + Z^0y_5 + Z^0y_6 + Z^0y_7,\\
Y_1 &= Z^0y_0 + Z^1y_1 + Z^2y_2 + Z^3y_3 - Z^0y_4 - Z^1y_5 - Z^2y_6 - Z^3y_7,\\
Y_2 &= Z^0y_0 + Z^2y_1 - Z^0y_2 - Z^2y_3 + Z^0y_4 + Z^2y_5 - Z^0y_6 - Z^2y_7,\\
Y_3 &= Z^0y_0 + Z^3y_1 - Z^2y_2 + Z^1y_3 - Z^0y_4 - Z^3y_5 + Z^2y_6 - Z^1y_7,\\
Y_4 &= Z^0y_0 - Z^0y_1 + Z^0y_2 - Z^0y_3 + Z^0y_4 - Z^0y_5 + Z^0y_6 - Z^0y_7,\\
Y_5 &= Z^0y_0 - Z^1y_1 + Z^2y_2 - Z^3y_3 - Z^0y_4 + Z^1y_5 - Z^2y_6 + Z^3y_7,\\
Y_6 &= Z^0y_0 - Z^2y_1 - Z^0y_2 + Z^2y_3 + Z^0y_4 - Z^2y_5 - Z^0y_6 + Z^2y_7,\\
Y_7 &= Z^0y_0 - Z^3y_1 - Z^2y_2 - Z^1y_3 - Z^0y_4 + Z^3y_5 + Z^2y_6 + Z^1y_7,\\
Y_8 &= Y_0.
\end{aligned}
$$

We see that these transforms now require $8 \times 8 = 64$ multiplications of complex numbers, in addition to some less time-consuming additions. We place these equations in an appropriate form for computing by regrouping the terms into sums and differences of the y's:

$$
\begin{aligned}
Y_0 &= Z^0(y_0 + y_4) + Z^0(y_1 + y_5) + Z^0(y_2 + y_6) + Z^0(y_3 + y_7),\\
Y_1 &= Z^0(y_0 - y_4) + Z^1(y_1 - y_5) + Z^2(y_2 - y_6) + Z^3(y_3 - y_7),\\
Y_2 &= Z^0(y_0 + y_4) + Z^2(y_1 + y_5) - Z^0(y_2 + y_6) - Z^2(y_3 + y_7),\\
Y_3 &= Z^0(y_0 - y_4) + Z^3(y_1 - y_5) - Z^2(y_2 - y_6) + Z^1(y_3 - y_7),\\
Y_4 &= Z^0(y_0 + y_4) - Z^0(y_1 + y_5) + Z^0(y_2 + y_6) - Z^0(y_3 + y_7),\\
Y_5 &= Z^0(y_0 - y_4) - Z^1(y_1 - y_5) + Z^2(y_2 - y_6) - Z^3(y_3 - y_7),\\
Y_6 &= Z^0(y_0 + y_4) - Z^2(y_1 + y_5) - Z^0(y_2 + y_6) + Z^2(y_3 + y_7),\\
Y_7 &= Z^0(y_0 - y_4) - Z^3(y_1 - y_5) - Z^2(y_2 - y_6) - Z^1(y_3 - y_7),\\
Y_8 &= Y_0.
\end{aligned}
$$

Note the repeating factors inside the parentheses, with combinations of the form $y_p \pm y_q$. These symmetries are systematized by introducing the *butterfly operation* (Figure 10.9). This operation takes the y_p and y_q data elements from the left wing and converts them to the $y_p + Zy_q$ elements in the upper- and lower-right wings. In Figure 10.10 we show what happens when we apply the butterfly operations to an entire FFT process, specifically to the pairs (y_0, y_4), (y_1, y_5), (y_2, y_6), and (y_3, y_7). Notice how the number of multiplications of complex numbers has been reduced: For the first butterfly operation there are 8 multiplications by Z^0; for the

Figure 10.9 The basic butterfly operation in which elements y_p and y_q on the left are transformed into $y_p + Z\,y_q$ and $y_p - Z\,y_q$ on the right.

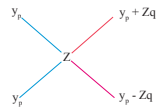
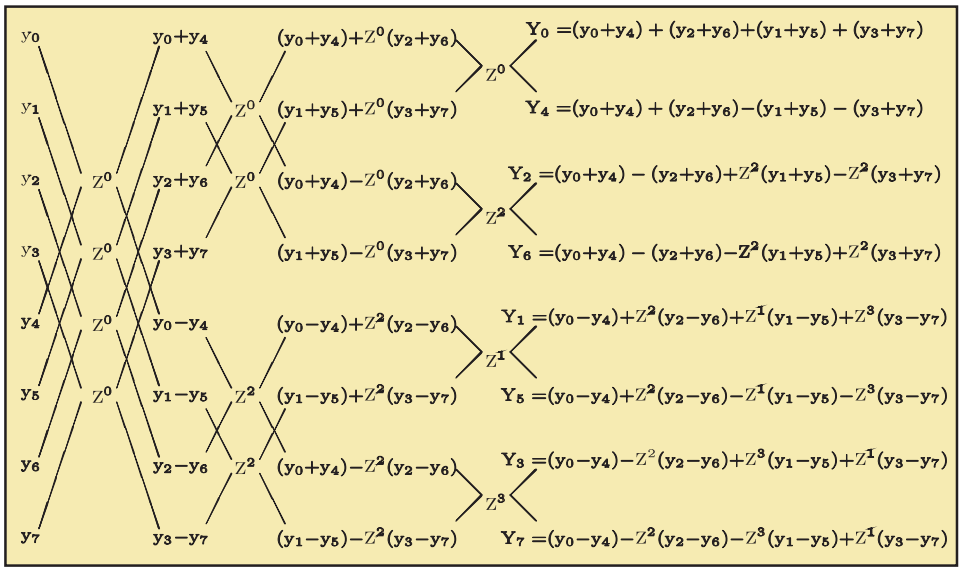


Figure 10.10 The butterfly operations performing a FFT on the eight data on the left leading to eight transforms on the right. The transforms are different linear combinations of the input data.



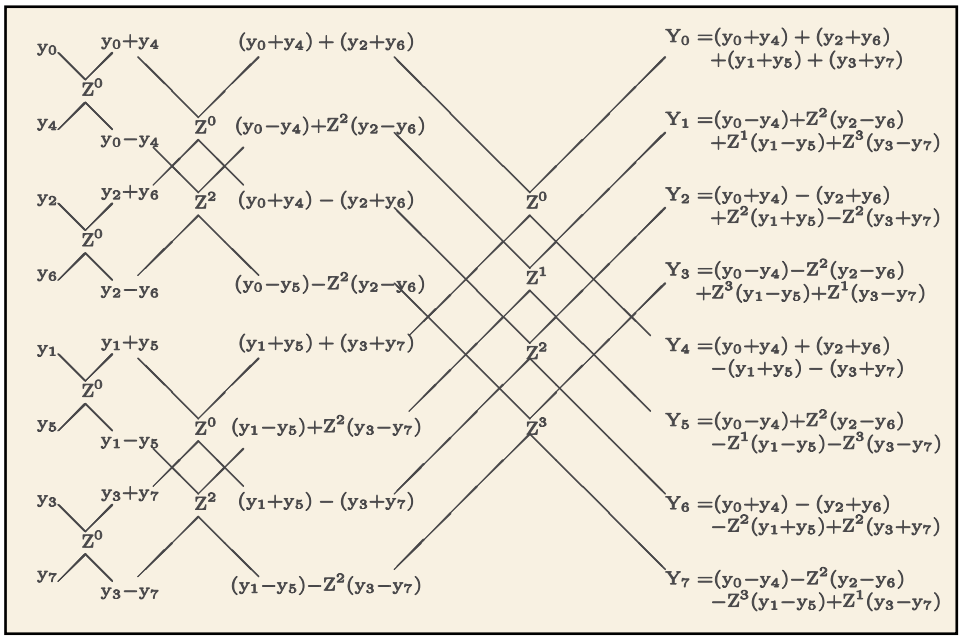
second butterfly operation there are 8 multiplications, and so forth, until a total of 24 multiplications is made in four butterflies. In contrast, 64 multiplications are required in the original DFT (10.8).

10.8.1 Bit Reversal

The reader may have observed that in Figure 10.10 we started with 8 data elements in the order 0–7 and that after three butterfly operators we obtained transforms in the order 0, 4, 2, 6, 1, 5, 3, 7. The astute reader may may also have observed that these numbers correspond to the bit-reversed order of 0–7. Let us look into this further. We need 3 bits to give the order of each of the 8 input data elements (the numbers 0–7). Explicitly, on the left in Table 10.1 we give the binary representation for decimal numbers 0–7, their bit reversals, and the corresponding decimal numbers. On the right we give the ordering for 16 input data elements, where we need 4 bits to enumerate their order. Notice that the order of the first 8 elements differs in the two cases because the number of bits being reversed differs. Notice too that after the reordering, the first half of the numbers are all even and the second half are all odd.

The fact that the Fourier transforms are produced in an order corresponding to the bit-reversed order of the numbers 0–7 suggests that if we process the data in the bit-reversed order 0, 4, 2, 6, 1, 5, 3, 7, then the output Fourier transforms will be ordered (see Table 10.1). We demonstrate this conjecture in Figure 10.11, where we see that to obtain the Fourier transform for the 8 input data, the butterfly operation had to be applied 3 times. The number 3 occurs here because it is the power of 2 that gives the number of data; that is, $2^3 = 8$. In general, in

Figure 10.11 A modified FFT in which the eight input data on the left are transformed into eight transforms on the right. The results are the same as in the previous figure, but now the output transforms are in numerical order whereas in the previous figure the input signals were in numerical order.



order for a FFT algorithm to produce transforms in the proper order, it must reshuffle the input data into bit-reversed order. As a case in point, our sample program starts by reordering the 16 (2^4) data elements given in Table 10.2. Now the 4 butterfly operations produce sequentially ordered output.

10.9 FFT IMPLEMENTATION

The first FFT program we are aware of was written in 1967 in Fortran IV by Norman Brenner at MIT's Lincoln Laboratory [Hi,76] and was hard for us to follow. Our (easier-to-follow) Java version of it is in Listing 10.3. Its input is $N = 2^n$ data to be transformed (FFTs always require 2^N input data). If the number of your input data is not a power of 2, then you can make it so by concatenating some of the initial data to the end of your input until a power of 2 is obtained; since a DFT is always periodic, this just starts the period a little earlier. This program assigns complex numbers at the 16 data points

$$y_m = m + mi, \quad m = 0, \ldots, 15, \tag{10.75}$$

reorders the data via bit reversal, and then makes four butterfly operations. The data are stored in the array **`dtr[max][2]`**, with the second subscript denoting real and imaginary parts. We increase speed further by using the 1-D array **`data`** to make memory access more direct:

$$\texttt{data}[1] = \texttt{dtr}[0][1], \quad \texttt{data}[2] = \texttt{dtr}[1][1], \quad \texttt{data}[3] = \texttt{dtr}[1][0], \ldots ,$$

which also provides storage for the output. The FFT transforms **`data`** using the butterfly operation and stores the results back in **`dtr[][]`**, where the input data were originally.

10.10 FFT ASSESSMENT

1. Compile and execute **`FFT.py`**. Make sure you understand the output.

Table 10.1 Binary-Reversed 0–7.

Decimal	*Binary*	*Reversal*	*Decimal Reversal*
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Binary-Reversed 0–16			
Decimal	*Binary*	*Reversal*	*Decimal Reversal*
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

Table 10.2 Reordering for 16 Data Complex Points

Order	Input Data	New Order	Order	Input Data	New Order
0	$0.0 + 0.0i$	$0.0 + 0.0i$	8	$8.0 + 8.0i$	$1.0 + 1.0i$
1	$1.0 + 1.0i$	$8.0 + 8.0i$	9	$9.0 + 9.0i$	$9.0 + 9.0i$
2	$2.0 + 2.0i$	$4.0 + 4.0i$	10	$10.0 + 10.i$	$5.0 + 5.0i$
3	$3.0 + 3.0i$	$12.0 + 12.0i$	11	$11.0 + 11.0i$	$13.0 + 13.0i$
4	$4.0 + 4.0i$	$2.0 + 2.0i$	12	$12.0 + 12.0i$	$3.0 + 3.0i$
5	$5.0 + 5.0i$	$10.0 + 10.i$	13	$13.0 + 13.0i$	$11.0 + 11.0i$
6	$6.0 + 6.0i$	$6.0 + 6.0i$	14	$14.0 + 14.i$	$7.0 + 7.0i$
7	$7.0 + 7.0i$	$14.0 + 14.0i$	15	$15.0 + 15.0i$	$15.0 + 15.0i$

2. Take the output from **FFT.py**, inverse-transform it back to signal space, and compare it to your input. [Checking that the double transform is proportional to itself is adequate, although the normalization factors in (10.32) should make the two equal.]
3. Compare the transforms obtained with a FFT to those obtained with a DFT (you may choose any of the functions studied before). Make sure to compare both precision and execution times.

Listing 10.3 **FFT.py** computes the FFT or inverse transform depending upon the sign of `isign`.

```
# FFT.py:  FFT for complex numbers in dtr[][2], returned in dtr

from numpy import *
from sys import version
if int(version[0]) > 2:                    # raw_input deprecated in Python 3
    raw_input=input
max = 2100
points = 1026                                          # Can be increased
data = zeros((max), float)
dtr  = zeros((points,2), float)

def fft(nn,isign):                                      # FFT of dtr[n,2]
    n = 2*nn
    for i in range(0,nn+1):                   # Original data in dtr to data
        j = 2*i+1
        data[j] = dtr[i,0]                            # Real dtr, odd data[j]
        data[j+1] = dtr[i,1]                        # Imag dtr, even data[j+1]
    j = 1                                     # Place data in bit reverse order
    for i in range(1,n+2, 2):
        if (i-j) < 0 :                       # Reorder equivalent to bit reverse
            tempr = data[j]
            tempi = data[j+1]
            data[j] = data[i]
            data[j+1] = data[i+1]
            data[i] = tempr
            data[i+1] = tempi
        m = n/2;
        while (m-2 > 0):
            if  (j-m) <= 0 :
                break
            j = j-m
            m = m/2
        j = j+m;

    print(" Bit-reversed data ")

    for i in range(1, n+1, 2):
        print("%2d  data[%2d]  %9.5f "%(i,i,data[i]))    # To see reorder
    mmax = 2
    while (mmax-n) < 0 :                                    # Begin transform
        istep = 2*mmax
        theta = 6.2831853/(1.0*isign*mmax)
        sinth = math.sin(theta/2.0)
        wstpr = -2.0*sinth**2
        wstpi = math.sin(theta)
        wr = 1.0
        wi = 0.0
        for m in range(1,mmax +1,2):
            for i in range(m,n+1,istep):
                j = i+mmax
                tempr = wr*data[j]   -wi *data[j+1]
                tempi = wr*data[j+1] +wi *data[j]
                data[j]   = data[i]   -tempr
                data[j+1] = data[i+1] -tempi
                data[i]   = data[i]   +tempr
                data[i+1] = data[i+1] +tempi
            tempr = wr
            wr = wr*wstpr - wi*wstpi + wr
            wi = wi*wstpr + tempr*wstpi + wi;
        mmax = istep
    for i in range(0,nn):
        j = 2*i+1
        dtr[i,0] = data[j]
        dtr[i,1] = data[j+1]
nn = 16                                                         # Power of 2
isign = -1                              # -1 transform, +1 inverse transform
print('        INPUT')
print("  i   Re part   Im  part")
for i in range(0,nn ):                                          # Form array
```

```
    dtr[i,0] = 1.0*i                                        # Real part
    dtr[i,1] = 1.0*i                                          # Im part
    print(" %2d %9.5f %9.5f" %(i,dtr[i,0],dtr[i,1]))
fft(nn, isign)                              # Call FFT, use global dtr[][]
print('    Fourier transform')
print("  i      Re      Im    ")
for i in range(0,nn):
    print(" %2d  %9.5f  %9.5f "%(i,dtr[i,0],dtr[i,1]))
print("Enter and return any character to quit")
s = raw_input()
```

Chapter Eleven

Wavelet Analysis & Data Compression

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Wavelets I	pdf	11.1	Wavelets II, Continuous	pdf	11.4
Wavelet III, Discrete	pdf	11.5			

PROBLEM

Problem: You have sampled the signal in Figure 11.1 that seems to contain an increasing number of frequencies as time increases. Your problem is to undertake a spectral analysis of this signal that tells you, in the most compact way possible, how much of each frequency is present at each instant in time. *Hint:* Although we want the method to be general enough to work with numerical data, for pedagogical purposes it is useful to know that the signal is

$$y(t) = \begin{cases} \sin 2\pi t, & \text{for } 0 \le t \le 2, \\ 5\sin 2\pi t + 10\sin 4\pi t, & \text{for } 2 \le t \le 8, \\ 2.5\sin 2\pi t + 6\sin 4\pi t + 10\sin 6\pi t, & \text{for } 8 \le t \le 12. \end{cases} \tag{11.1}$$

11.1 UNIT I. WAVELET BASICS

The Fourier analysis we used in §10.4.1 reveals the amount of the harmonic functions $\sin(\omega t)$ and $\cos(\omega t)$ and their overtones that are present in a signal. An expansion in periodic functions is fine for *stationary* signals (those whose forms do not change in time) but has shortcomings for the variable form of our **problem** signal (11.1). One such problem is that the Fourier

Figure 11.1 The input time signal (11.1) we wish to analyze. The signal is seen to contain additional frequencies as time increases. The boxes are possible placements of windows for short-time Fourier transforms.

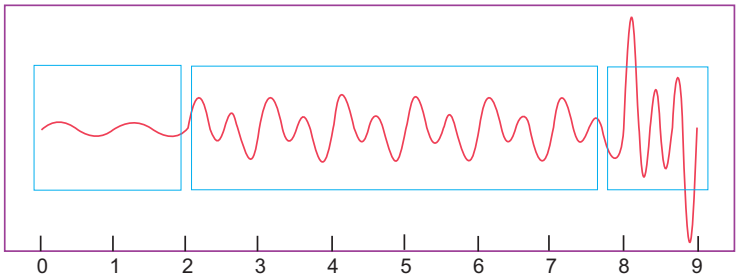
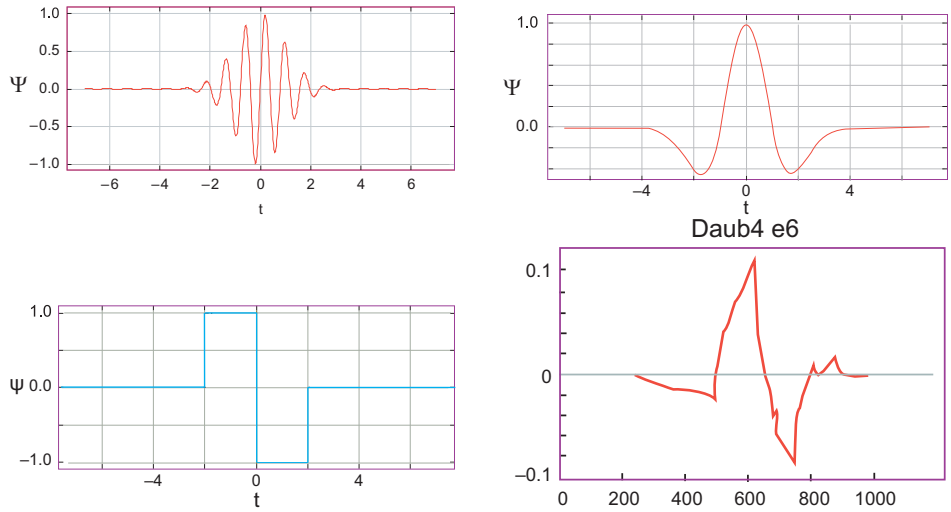


Figure 11.2 Four possible mother wavelets that can be used to generate entire sets of daughter wavelets. *Clockwise from top:* Morlet (real part), Mexican hat, Daub4 e6 (explained later), and Haar. The daughter wavelets are generated by scaling and translating these mother wavelets.



reconstruction has all constituent frequencies occurring simultaneously and so does not contain *time resolution* information indicating when each frequency occurs. Another shortcoming is that all the Fourier components are correlated, which results in more information being stored than may be needed and no convenient way to remove the excess storage.

There are a number of techniques that extend simple Fourier analysis to nonstationary signals. In this chapter we include an introduction to *wavelet analysis*, a field that has seen extensive development and application in the last decade in areas as diverse as brain waves and gravitational waves. The idea behind wavelet analysis is to expand a signal in a complete set of functions (wavelets), each of which oscillates for a finite period of time and each of which is centered at a different time. To give you a preview before we get into the details, we show four sample wavelets in Figure 11.2. Because each wavelet is local in time, it is a wave packet[1] containing a range of frequencies. These wave packets are called wavelets because they are small and do not extend for long times.

Although wavelets are required to oscillate in time, they are not restricted to a particular functional form [Add 02]. As a case in point, they may be oscillating Gaussians (Morlet: top left in Figure 11.2),

$$\Psi(t) = e^{2\pi i t} e^{-t^2/2\sigma^2} = (\cos 2\pi t + i \sin 2\pi t) e^{-t^2/2\sigma^2} \quad \text{(Morlet)}, \tag{11.2}$$

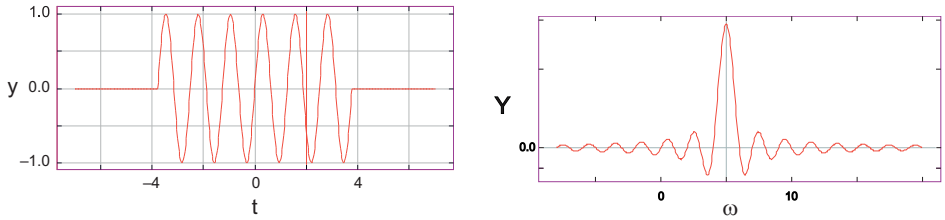
the second derivative of a Gaussian (Mexican hat, top right),

$$\Psi(t) = -\sigma^2 \frac{d^2}{dt^2} e^{-t^2/2\sigma^2} = \left(1 - \frac{t^2}{\sigma^2}\right) e^{-t^2/2\sigma^2}, \tag{11.3}$$

an up-and-down step function (lower left), or a fractal shape (bottom right). Such wavelets are *localized* in both time and frequency; that is, they are large for just a finite time and contain a finite range of frequencies. As we shall see, translating and scaling one of these *mother wavelets* generates an entire set of *child wavelet* basis functions, with each individual function covering a different frequency range at a different time.

[1] We discuss wave packets further in §11.2.

Figure 11.3 *Left:* A wave packet in time corresponding to the functional form (11.4) with $\omega_0 = 5$ and $N = 6$. *Right:* The Fourier transform in frequency of this same wave packet.



11.2 WAVE PACKETS AND UNCERTAINTY PRINCIPLE (THEORY)

A *wave packet* or *wave train* is a collection of waves added together in such a way as to produce a pulse of width Δt. As we shall see, the Fourier transform of a wave packet is a pulse in the frequency domain of width $\Delta\omega$. We will first study such wave packets analytically and then use others numerically. An example of a simple wave packet is just a sine wave that oscillates at frequency ω_0 for N periods (Figure 11.3 left) [A&W 01]

$$y(t) = \begin{cases} \sin\omega_0 t, & \text{for} \quad |t| < N\frac{\pi}{\omega_0} \equiv N\frac{T}{2}, \\ 0, & \text{for} \quad |t| > N\frac{\pi}{\omega_0} \equiv N\frac{T}{2}, \end{cases} \tag{11.4}$$

where we relate the frequency to the period via the usual $\omega_0 = 2\pi/T$. In terms of these parameters, the width of the wave packet is

$$\Delta t = NT = N\frac{2\pi}{\omega_0}. \tag{11.5}$$

The Fourier transform of the wave packet (11.4) is a straight-forward application of the transform formula (10.18):

$$Y(\omega) = \int_{-\infty}^{+\infty} dt\, \frac{e^{-i\omega t}}{\sqrt{2\pi}} y(t) = \frac{-i}{\sqrt{2\pi}} \int_0^{N\pi/\omega_0} dt\, \sin\omega_0 t\, \sin\omega t \tag{11.6}$$

$$= \frac{(\omega_0+\omega)\sin\left[(\omega_0-\omega)\frac{N\pi}{\omega_0}\right] - (\omega_0-\omega)\sin\left[(\omega_0+\omega)\frac{N\pi}{\omega_0}\right]}{\sqrt{2\pi}(\omega_0^2-\omega^2)},$$

where we have dropped a factor of $-i$ that affects only the phase. While at first glance (11.6) appears to be singular at $\omega = \omega_0$, it just peaks there (Figure 11.3 right), reflecting the predominance of frequency ω_0. However, there are sharp corners in the signal $y(t)$ (Figure 11.3 left), and these give $Y(\omega)$ a width $\Delta\omega$.

There is a fundamental relation between the widths Δt and $\Delta\omega$ of a wave packet. Although we use a specific example to determine that relation, it is true in general. While there may not be a precise definition of "width" for all functions, one can usually deduce a good measure of the width (say, within 25%). To illustrate, if we look at the right of Figure 11.3, it makes sense to use the distance between the first zeros of the transform $Y(\omega)$ (11.6) as the width $\Delta\omega$. The zeros occur at

$$\frac{\omega-\omega_0}{\omega_0} = \pm\frac{1}{N} \Rightarrow \quad \Delta\omega \simeq \omega - \omega_0 = \frac{\omega_0}{N}, \tag{11.7}$$

where N is the number of cycles in our original wave packet. Because the wave packet in time makes N oscillations each of period T, a reasonable measure of the width Δt of the signal $y(t)$

is

$$\Delta t = NT = N\frac{2\pi}{\omega_0}. \tag{11.8}$$

When the products of the frequency width (11.7) and the time width (11.8) are combined, we obtain

$$\Delta t\,\Delta\omega \geq 2\pi. \tag{11.9}$$

The greater-than sign is used to indicate that this is a minimum, that is, that $y(t)$ and $Y(\omega)$ extend beyond Δt and $\Delta\omega$, respectively. Nonetheless, most of the signal and transform should lie within the bound (11.9).

A relation of the form (11.9) also occurs in quantum mechanics, where it is known as the *Heisenberg uncertainty principle*, with Δt and $\Delta\omega$ being called the uncertainties in t and ω. It is true for transforms in general and states that as a signal is made more localized in time (smaller Δt) the transform becomes less localized (larger $\Delta\omega$). Conversely, the signal $y(t) = \sin\omega_0 t$ is completely localized in frequency and so has an infinite extent in time, $\Delta t \simeq \infty$.

11.2.1 Wave Packet Assessment

Consider the following wave packets:

$$y_1(t) = e^{-t^2/2}, \quad y_2(t) = \sin(8t)e^{-t^2/2}, \quad y_3(t) = (1-t^2)\,e^{-t^2/2}.$$

For each wave packet:

1. Estimate the width Δt. A good measure might be the *full width at half-maxima* (FWHM) of $|y(t)|$.
2. Use your DFT program to evaluate and plot the Fourier transform $Y(\omega)$. Make *both* a linear and a semilog plot (small components are often important, yet not evident in linear plots). Make sure that your transform has a good number of closely spaced values over a range that is large enough to show the periodicity of $Y(\omega)$.
3. What are the units for $Y(\omega)$ and ω? in your DFT?
4. Estimate the width $\Delta\omega$ of the transform. A good measure might be the *full width at half-maxima* of $|Y(\omega)|$.
5. Determine the constant C for the uncertainty principle

$$\Delta t\,\Delta\omega \geq 2\pi C.$$

11.3 SHORT-TIME FOURIER TRANSFORMS (MATH)

The constant amplitude of the functions $\sin\omega t$ and $\cos\omega t$ for all times can limit the usefulness of Fourier transforms. Because these functions and their overtones extend over all times with that constant amplitude, there is considerable overlap among them, and thus the information present in various Fourier components is correlated. This is undesirable for compressed data storage, where you want to store a minimum number of data and want to be able to cut out some of these data with just a minimal effect on the signal reconstruction.[2] In *lossless compression*, which reproduces the original signal exactly, you save space by storing how many times each

[2] Wavelets have also proven to be a highly effective approach to data compression, with the Joint Photographic Experts Group (JPEG) 2000 standard being based on wavelets. In D we give a full example of image compression with wavelets.

data element is repeated and where each element is located. In *lossy compression*, in addition to removing repeated elements, you also eliminate some transform components, consistent with the uncertainty relation (11.9) and with the level of resolution required in the reproduction. This leads to a greater compression.

In §10.4.1 we defined the Fourier transform $Y(\omega)$ of signal $y(t)$ as

$$Y(\omega) = \int_{-\infty}^{+\infty} dt\, \frac{e^{-i\omega t}}{\sqrt{2\pi}}\, y(t) = \langle \omega \,|y\rangle\,. \tag{11.10}$$

As is true for simple vectors, you can think of (11.10) as giving the overlap or scalar product of the basis function $\exp(i\omega t)/\sqrt{2\pi}$ and the signal $y(t)$ [notice that the complex conjugate of the exponential basis function appears in (11.10)]. Another view of (11.10) is as the mapping or projection of the signal into ω space. In this case the overlap projects the amount of the periodic function $\exp(i\omega t)/\sqrt{2\pi}$ in the signal $y(t)$. In other words, the Fourier component $Y(\omega)$ is also the correlation between the signal $y(t)$ and the basis function $\exp(i\omega t)/\sqrt{2\pi}$, which is what results from filtering the signal $y(t)$ through a frequency-ω filter. If there is no $\exp(i\omega t)$ in the signal, then the integral vanishes and there is no output. If $y(t) = \exp(i\omega t)$, the signal is at only one frequency, and the integral is accordingly singular.

The problem signal in Figure 11.1 clearly has different frequencies present at different times and for different lengths of time. In the past this signal might have been analyzed with a precursor of wavelet analysis known as the *short-time Fourier transform*. With that technique, the signal $y(t)$ is "chopped up" into different segments along the time axis, with successive segments centered about successive times $\tau_1, \tau_2, \ldots, \tau_N$. For instance, we show three such segments in Figure 11.1. Once we have the dissected signal, a Fourier analysis is made of each segment. We are then left with a sequence of transforms $\left(Y^{(\mathrm{ST})}_{\tau_1}, Y^{(\mathrm{ST})}_{\tau_2}, \ldots, Y^{(\mathrm{ST})}_{\tau_N}\right)$, one for each short-time interval, where the superscript $^{(\mathrm{ST})}$ indicates short time.

Rather than chopping up a signal, we express short-time Fourier transforming mathematically by imagining translating a *window function* $w(t-\tau)$ by a time τ over the signal in Figure 11.1:

$$Y^{(\mathrm{ST})}(\omega,\tau) = \int_{-\infty}^{+\infty} dt\, \frac{e^{i\omega t}}{\sqrt{2\pi}}\, w(t-\tau)\, y(t). \tag{11.11}$$

Here the values of the translation time τ correspond to different locations of window w over the signal, and the window function is essentially a transparent box of small size on an opaque background. Any signal within the width of the window is transformed, while the signal lying outside the window is not seen. Note that in (11.11) the extra variable τ in the Fourier transform indicating the location of the time around which the window was placed. Clearly, since the short-time transform is a function of two variables, a surface or 3-D plot is needed to view the amplitude as a function of both ω and τ.

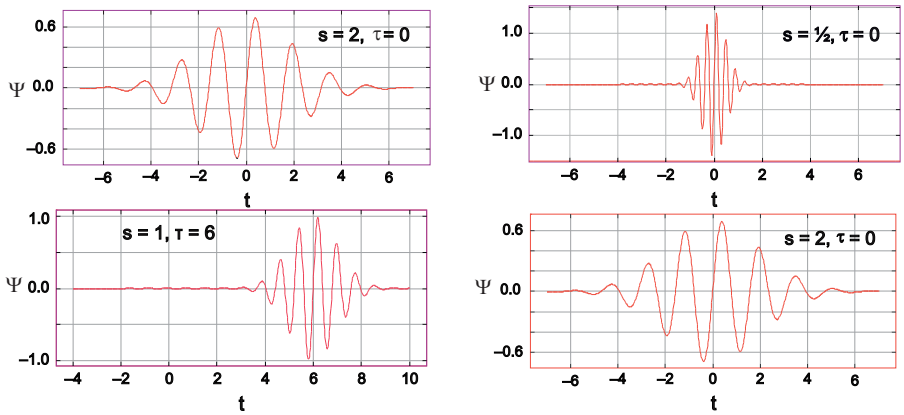
11.4 THE WAVELET TRANSFORM

The wavelet transform of a time signal $y(t)$,

$$Y(s,\tau) = \int_{-\infty}^{+\infty} dt\, \psi^*_{s,\tau}(t) y(t) \quad \text{(wavelet transform)}, \tag{11.12}$$

is similar in concept and notation to a short-time Fourier transform. Rather than using $\exp(i\omega t)$ as the basis functions, we use wave packets or wavelets $\psi_{s,\tau}(t)$ localized in time, such as the those shown in Figure 11.2. Because each wavelet is localized in time, each acts as its own

Figure 11.4 Four wavelet basis functions (daughters) generated by scaling (s) and translating (τ) a oscillating Gaussian mother wavelet. *Clockwise from top:* ($s = 1, \tau = 0$), ($s = 1/2, \tau = 0$), ($s = 1, \tau = 6$), and ($s = 2, \tau = 60$). Note how $s < 1$ is a wavelet with higher frequency, while $s > 1$ has a lower frequency than the $s = 1$ mother. Likewise, the $\tau = 6$ wavelet is just a translated version of the $\tau = 0$ one directly above it.



window function. Because each wavelet is oscillatory, each contains its own small range of frequencies.

Equation (11.12) says that the wavelet transform $Y(s, \tau)$ is a measure of the amount of basis function $\psi_{s,\tau}(t)$ present in the signal $y(t)$. The τ variable indicates the time portion of the signal being decomposed, while the s variable is equivalent to the frequency present during that time:

$$\omega = \frac{2\pi}{s}, \quad s = \frac{2\pi}{\omega} \quad \text{(scale–frequency relation).} \tag{11.13}$$

Because it is key to much that follows, it is a good idea to think about (11.13) for a while. If we are interested in the time *details* of a signal, then this is another way of saying that we are interested in what is happening at small values of the *scale* s. Equation (11.13) indicates that small values of s correspond to high-frequency components of the signal. That being the case, the time details of the signal are in the high-frequency, or low-scale, components.

11.4.1 Generating Wavelet Basis Functions

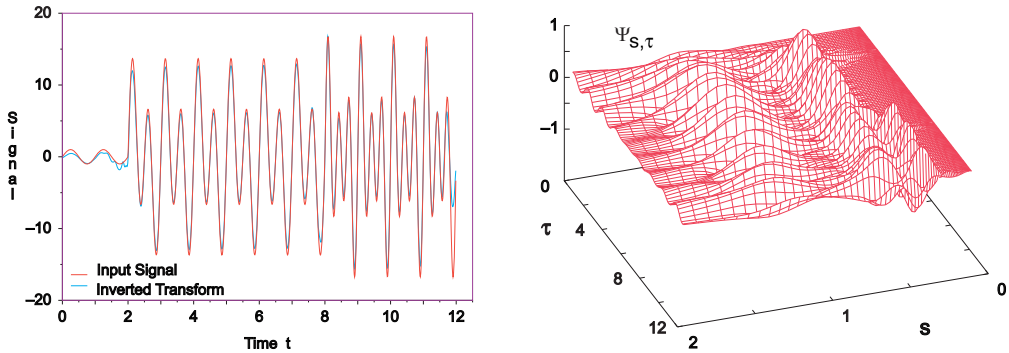
The conceptual discussion of wavelets is over, and it is time to get to work. We first need a technique for generating wavelet bases, and then we need to discretize this technique. As is often the case, the final formulation will turn out to be simple and short, but it will be a while before we get there. Just as the expansion of a function in a complete orthogonal set is not restricted to any particular set, so the theory of wavelets is not restricted to any particular wavelet basis, although there is some art involved in choosing the most appropriate wavelets for a given signal. The standard way to generate a family of wavelet basis functions starts with $\Psi(t)$, a *mother* or *analyzing* function of the real variable t, and then use this to generate daughter wavelets. As a case in point, let us start with the mother wavelet

$$\Psi(t) = \sin(8t)e^{-t^2/2}. \tag{11.14}$$

We then generate the four wavelet basis functions displayed in Figure 11.4 by scaling, translating, and normalizing this mother wavelet:

$$\psi_{s,\tau}(t) \stackrel{\text{def}}{=} \frac{1}{\sqrt{s}}\Psi\left(\frac{t-\tau}{s}\right) = \frac{1}{\sqrt{s}}\sin\left[\frac{8(t-\tau)}{s}\right] e^{-(t-\tau)^2/2s^2}. \tag{11.15}$$

Figure 11.5 *Left:* Comparison of an input and reconstituted signal (11.18) using Morlet wavelets (the curves overlap nearly perfectly). As expected for Fourier transforms, the reconstruction is least accurate near the end-points. *Right:* The continuous wavelet spectrum obtained by analyzing the input signal with Morelet wavelets. Observe how at small values of time τ there is predominantly one frequency present, how a second, higher-frequency (smaller-scale) component enters at intermediate times, and how at larger times a still higher-frequency components enter. (Figure courtesy of Z. Dimcovic.)



We see that larger or smaller values of s, respectively, expand or contract the mother wavelet, while different values of τ shift the center of the wavelet. Because the wavelets are inherently oscillatory, the scaling leads to the same number of oscillations occurring in different time spans, which is equivalent to having basis states with differing frequencies. We see that $s < 1$ produces a higher-frequency wavelet, while $s > 1$ produces a lower-frequency one, both of the same shape. As we shall see, we do not need to store much information to outline the large-time-scale s behavior of a signal (its *smooth envelope*), but we do need more information to specify its short-time-scale s behavior (*details*). And if we want to resolve finer features in the signal, then we will need to have more information on yet finer details. Here the division by $\sqrt{s}$ is made to ensure that there is equal "power" (or energy or intensity) in each region of s, although other normalizations can also be found in the literature. After substituting in the daughters, the wavelet transform (11.12) and its inverse [VdB 99] are

$$Y(s,\tau) = \frac{1}{\sqrt{s}} \int_{-\infty}^{+\infty} dt\, \Psi^* \left(\frac{t-\tau}{s} \right) y(t) \quad \textbf{(Wavelet TF)}, \tag{11.16}$$

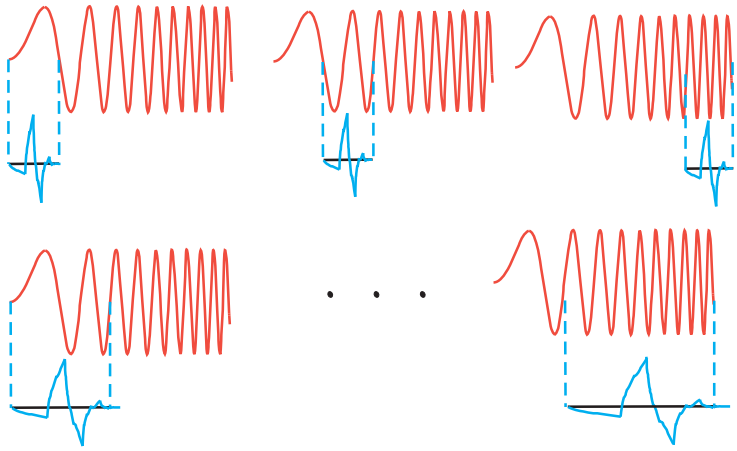
$$y(t) = \frac{1}{C} \int_{-\infty}^{+\infty} d\tau \int_{0}^{+\infty} ds \frac{\psi^*_{s,\tau}(t)}{s^{3/2}} Y(s,\tau) \quad \textbf{(inverse transform)}, \tag{11.17}$$

where the normalization constant C depends on the wavelet used.

The general requirements for a mother wavelet Ψ are [Add 02, VdB 99]

1. $\Psi(t)$ is real.
2. $\Psi(t)$ oscillates around zero such that its average is zero:
$$\int_{-\infty}^{+\infty} \Psi(t)\, dt = 0.$$
3. $\Psi(t)$ is local, that is, a wave packet, and is square-integrable:
$$\Psi(|t| \to \infty) \to 0 \quad \text{(rapidly)}, \qquad \int_{-\infty}^{+\infty} |\Psi(t)|^2\, dt < \infty.$$
4. The transforms of low powers of t vanish, that is, the first p moments:

Figure 11.6 A schematic representation of the steps followed in performing a wavelet transformation over all time displacements and scales. The red signal is first analyzed by evaluating its overlap with a narrow wavelet at the signal's beginning. This produces a coefficient that measures the similarity of the signal to the wavelet. The wavelet is successively shifted over the length of the signal and the overlaps are successively evaluated. After the entire signal is covered, the wavelet is expanded and the entire analysis is repeated. This



$$\int_{-\infty}^{+\infty} t^0\,\Psi(t)\,dt = \int_{-\infty}^{+\infty} t^1\,\Psi(t)\,dt = \cdots = \int_{-\infty}^{+\infty} t^{p-1}\,\Psi(t)\,dt = 0.$$

This makes the transform more sensitive to details than to general shape.

You can think of scale as being like the scale on a map (also discussed in §13.5.2 with reference to fractal analysis) or in terms of *resolution*, as might occur in photographic images. Regardless of the words, we will see in Chapter 12, "Discrete & Continuous Nonlinear Dynamics," that if we have a fractal, then we have a self-similar object that looks the same at all scales or resolutions. Similarly, each wavelet basis function in a set is self-similar to the others, but at a different scale or location.

In summary, wavelet bases are functions of the time variable t, as well as of the two parameters s and τ. The t variable is integrated over to yield a transform that is a function of the time scale s (frequency $2\pi/s$) and window location τ.

As an example of how we use the two degrees of freedom, consider the analysis of a chirp signal $\sin(60t^2)$ (Figure 11.6). We see that a slice at the beginning of the signal is compared to our first basis function. (The comparison is done via the *convolution* of the wavelet with the signal.) This first comparison is with a narrow version of the wavelet, that is, at low scale, and yields a single coefficient. The comparison at this scale continues with the next signal slice and ends when the entire signal has been covered (the top row in Figure 11.6). Then the wavelet is expanded, and comparisons are repeated. Eventually, the data are processed at all scales and at all time intervals. The narrow signals correspond to a high-resolution analysis, while the broad signals correspond to low resolution. As the scales get larger (lower frequencies, lower resolution), fewer details of the time signal remain visible, but the overall shape or gross features of the signal become clearer.

11.4.2 Continuous Wavelet Transform Implementation

We want to develop some intuition as to what wavelet transforms look like before going on to apply them in unknown situations. Accordingly, modify the program you have been using for the discrete Fourier transform so that it now computes the continuous wavelet transform.

1. You will want to see the effect of using different mother wavelets. Accordingly, write a method that calculates the mother wavelet for
   a. a Morlet wavelet (11.2),
   b. a Mexican hat wavelet (11.3),
   c. a Haar wavelet (the square wave in Figure 11.2).
2. Try out your transform for the following input signals and see if the results make sense:
   a. A pure sine wave $y(t) = \sin 2\pi t$,
   b. A sum of sine waves $y(t) = 2.5 \sin 2\pi t + 6 \sin 4\pi t + 10 \sin 6\pi t$,
   c. The nonstationary signal for our problem (11.1)

$$y(t) = \begin{cases} \sin 2\pi t, & \text{for } 0 \le t \le 2, \\ 5 \sin 2\pi t + 10 \sin 4\pi t, & \text{for } 2 \le t \le 8, \\ 2.5 \sin 2\pi t + 6 \sin 4\pi t + 10 \sin 6\pi t, & \text{for } 8 \le t \le 12. \end{cases} \tag{11.18}$$

   d. The half-wave function

$$y(t) = \begin{cases} \sin \omega t, & \text{for } 0 < t < T/2, \\ 0, & \text{for } T/2 < t < T. \end{cases}$$

3. ⊙ Use (11.17) to invert your wavelet transform and compare the reconstructed signal to the input signal (you can normalize the two to each other). On the right in Figure 11.5 we show our comparison.

In Listing 11.1 we give our *continuous wavelet transformation* **`CWT.py`** [Lang]. Because wavelets, with their transforms in two variables, are somewhat hard to grasp at first, we suggest that you write your own code and include a portion that does the inverse transform as a check. In the next section we will describe the *discrete wavelet transformation* that makes optimal discrete choices for the scale and time translation parameters s and τ. Figure 11.5 shows the spectrum produced for the input signal (11.1) in Figure 11.1. As was our goal, we see predominantly one frequency at short times, two frequencies at intermediate times, and three frequencies at longer times.

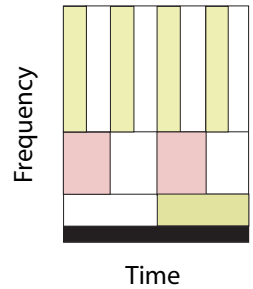
11.5 UNIT II. DISCRETE WAVELET TRANSFORMS, MRA⊙

As was true for DFTs, if a time signal is measured at only N discrete times,

$$y(t_m) \equiv y_m, \quad m = 1, \ldots, N, \tag{11.19}$$

then we can determine only N independent components of the transform Y. The trick is to compute only the N independent components required to reproduce the input signal, consistent with the uncertainty principle. The *discrete wavelet transform* (DWT) evaluates the transforms

Figure 11.7 A graphical representation of the relation between time and frequency resolutions (the uncertainty relation). Each box represents an equal portion of the time–frequency plane but with different proportions of time and frequency.



with discrete values for the scaling parameter s and the time translation parameter τ:

$$\psi_{j,k}(t) = \frac{\Psi\left[(t-k2^j)/2^j\right]}{\sqrt{2^j}} \equiv \frac{\Psi\left(t/2^j - k\right)}{\sqrt{2^j}} \quad \text{(DWT)} \tag{11.20}$$

$$s = 2^j, \quad \tau = \frac{k}{2^j}, \quad k,\ j = 0, 1, \ldots. \tag{11.21}$$

Here j and k are integers whose maximum values are yet to be determined, and we have assumed that the total time interval $T = 1$, so that time is always measured in integer values. This choice of s and τ, based on powers of 2, is called a *dyadic grid* arrangement and is seen to automatically perform the scalings and translations at the different time scales that are at the heart of wavelet analysis.[3] The discrete wavelet transform now becomes

$$Y_{j,k} = \int_{-\infty}^{+\infty} dt\, \psi_{j,k}(t)\, y(t) \simeq \sum_m \psi_{j,k}(t_m) y(t_m) h \quad \text{(DWT)}, \tag{11.22}$$

where the discreteness here refers to the wavelet basis set and *not* the time variable. For an orthonormal wavelet basis, the inverse discrete transform is then

$$y(t) = \sum_{j,\,k=-\infty}^{+\infty} Y_{j,k}\, \psi_{j,k}(t) \quad \text{(inverse DWT)}. \tag{11.23}$$

This inversion will exactly reproduce the input signal at the N input points if we sum over an infinite number of terms [Add 02]. Practical calculations will be less exact.

Listing 11.1 **CWT.py** computes a normalized continuous wavelet transform of the signal data in `input` (here assigned as a sum of sine functions) using Morlet wavelets (courtesy of Z. Dimcovic). The discrete wavelet transform (DWT) in Listing 11.2 is faster and yields a compressed transform but is less transparent.

```
# CWT.py  Continuous Wavelet TF. Based on program by Zlatko Dimcovic

import matplotlib.pylab as p;
from mpl_toolkits.mplot3d import Axes3D ;
from visual import *;

invtrgr = display(x=0, y=0, width=600, height=200, title='Inverse TF')
invtr   = curve(x=list(range(0,240)), display=invtrgr, color=color.green)
iT =  0.0;          fT =  12.0;     W = fT - iT;     N =  240;    h =  W/N
noPtsSig =  N;      noS =  20;      noTau =  90;     iTau =  0.
iS =  0.1;          tau =  iTau;        s =  iS

# Need *very* small s steps for high frequency if s small;
dTau =  W/noTau;    dS =  (W/iS)**(1./noS);
maxY =  0.001;      sig =  zeros((noPtsSig), float)

def signal(noPtsSig, y):                                        # Signal
    t = 0.0;     hs = W/noPtsSig;     t1 = W/6.;     t2 = 4.*W/6.
```

[3]Note that some references scale down with increasing j, in contrast to our scaling up.

```
    for i in range(0, noPtsSig):
        if  t >= iT  and t <=  t1:  y[i] =  sin(2*pi*t)
        elif t >= t1 and t <=  t2: y[i] = 5.*sin(2*pi*t)+10.*sin(4*pi*t);
        elif t >= t2 and t <=  fT:
             y[i] = 2.5*sin(2*pi*t) + 6.*sin(4*pi*t) + 10.*sin(6*pi*t)
        else:
            print("In signal(...) : t out of range.")
            sys.exit(1)
        t += hs
signal(noPtsSig, sig)                                        # Form signal
Yn =  zeros( (noS+1, noTau+1), float)                          # Transform

def morlet(t, s, tau):                                     # Mother wavelet
     T =  (t - tau)/s
     return sin(8*T) * exp( - T*T/2. )

def transform(s, tau, sig):                                     # Finds TF
    integral = 0.
    t = iT;                                                         # Time
    for i in range(0, len(sig) ):
         t += h
         integral += sig[i]*morlet(t, s, tau)*h
    return integral / sqrt(s)

def invTransform(t, Yn):                          # Compute nonormed inverse
    s = iS
    tau = iTau
    recSig_t = 0
    for i in range (0, noS):
        s *= dS                                                  # Scale = s
        tau = iTau
        for j in range (0, noTau):
            tau += dTau
            recSig_t += dTau*dS *(s**(-1.5))*Yn[i,j] * morlet(t, s, tau)
    return recSig_t

print("working, finding transform, count 20")
for i in range( 0, noS):
    s *= dS            # was with *                                # Scaling
    tau = iT
    print(i)
    for j in range(0, noTau):
         tau += dTau                                         # Translation
         Yn[i, j] = transform(s, tau, sig)
print("transform found")
for i in range( 0, noS):
    for j in range( 0, noTau):
        if Yn[i, j] > maxY or Yn[i, j] < - 1 *maxY :
            maxY = abs( Yn[i, j] )                            # Find max Y
tau =  iT
s =  iS
print("normalize")
for i in range( 0, noS):
     s *= dS
     for j in range( 0, noTau):
         tau +=   dTau                                           # Transform
         Yn[i, j] = Yn[i, j]/maxY
     tau = iT
print("finding inverse transform")                       # Find inverse TF
recSigData =  "recSig.dat"
recSig =  zeros(len(sig) )                               # Same resolution
t =  0.0;
print("count to 10")
kco = 0;            j = 0;              Yinv =  Yn
for rs in range(0, len(recSig) ):
    recSig[rs] = invTransform(t, Yinv)             # Find original signal
    invtr.x[rs] = 2*rs - N                                  # Plot inverse
    invtr.y[rs] = 10.*recSig[rs]                               # Transform
    t += h
    if kco %24 == 0:
        j += 1
        print(j)                                  # To see how long it takes
    kco += 1
x = list(range(1, noS + 1))                                     # For x plot
y = list(range(1, noTau + 1))                                   # For y plot
X,Y = p.meshgrid(x, y)                                # Grid for s and tau

def functz(Yn):                                          # Return transform
    z = Yn[X, Y]
    return z

Z = functz(Yn)
fig = p.figure()                                           # Creates figure
```

```
ax = Axes3D(fig)
ax.plot_wireframe(X, Y, Z, color = 'r')
ax.set_xlabel('s: scale')
ax.set_ylabel('Tau')
ax.set_zlabel('Transform')
p.show()

print("nDone")
print("Enter and return a character to finish")
s=raw_input()
```

Notice in (11.20) and (11.22) that we have kept the time variable t in the wavelet basis functions continuous, even though s and τ are made discrete. This is useful in establishing the orthonormality of the basis functions,

$$\int_{-\infty}^{+\infty} dt\, \psi^*_{j,k}(t)\, \psi_{j',k'}(t) = \delta_{jj'}\, \delta_{kk'}, \tag{11.24}$$

where $\delta_{m,n}$ is the Kronecker delta function. Being normalized to 1 means that each wavelet basis has "unit energy"; being orthogonal means that each basis function is independent of the others. And because wavelets are localized in time, the different transform components have low levels of correlation with each other. Altogether, this leads to efficient and flexible data storage.

The use of a discrete wavelet basis makes it clear that we sample the input signal at the discrete values of time determined by the integers j and k. In general, you want time steps that sample the signal at enough times in each interval to obtain the desired level of precision. A rule of thumb is to start with 100 steps to cover each major feature. Ideally, the needed times correspond to the times at which the signal was sampled, although this may require some forethought.

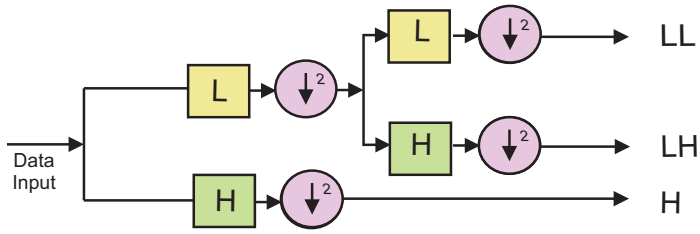
Consider Figure 11.7. We measure a signal at a number of discrete times within the intervals (k or τ values) corresponding to the vertical columns of fixed width along the time axis. For each time interval, we want to sample the signal at a number of scales (frequencies or j values). However, as discussed in §11.2, the basic mathematics of Fourier transforms indicates that the width Δt of a wave packet $\psi(t)$ and the width $\Delta\omega$ of its Fourier transform $Y(\omega)$ are related by an uncertainty principle

$$\Delta\omega\, \Delta t \geq 2\pi.$$

This relation constrains the number of times we can meaningfully sample a signal in order to determine a number of Fourier components. So while we may want a high-resolution reproduction of our signal, we do not want to store more data than are needed to obtain that reproduction. If we sample the signal for times centered about some τ in an interval of width $\Delta\tau$ (Figure 11.7) and then compute the transform at a number of scales s or frequencies $\omega = 2\pi/s$ covering a range of height $\Delta\omega$, then the relation between the height and width is restricted by the uncertainty relation, which means that each of the rectangles in Figure 11.7 has the same area $\Delta\omega\, \Delta t = 2\pi$. The increasing heights of the rectangles at higher frequencies means that a larger range of frequencies should be sampled as the frequency increases. The premise here is that the low-frequency components provide the gross or *smooth* outline of the signal which, being smooth, does not require much detail, while the high-frequency components give the details of the signal over a short time interval and so require many components in order to record these details with high resolution.

Industrial-strength wavelet analyses do not compute explicit integrals but instead apply a technique known as *multiresolution analysis* (MRA). We give an example of this technique in Figure 11.8 and in the code **DWT.py** in Listing 11.2. It is based on a *pyramid algorithm* that

Figure 11.8 A multifrequency dyadic (power-of-2) filter tree used for discrete wavelet transformations. The L boxes represent lowpass filters and the H boxes represent highpass filters. Each filter performs a convolution (transform). The circles containing "↓2" filter out half of the signal that enters them, which is called *subsampling* or *factor-of-2 decimation*. The signal on the left yields a transform with a single low and two high components (less information is needed about the low components for a faithful reproduction).



samples the signal at a finite number of times and then passes it successively through a number of *filters*, with each filter representing a digital version of a wavelet.

Filters were discussed in §10.7, where in (10.59) we defined the action of a linear filter as a convolution of the filter response function with the signal. A comparison of the definition of a filter to the definition of a wavelet transform (11.12) shows that the two are essentially the same. Such being the case, the result of the transform operation is a weighted sum over the input signal values, with each weight the product of the integration weight times the value of the wavelet function at the integration point. Therefore, *rather than tabulate explicit wavelet functions, a set of filter coefficients is all that is needed for discrete wavelet transforms*.

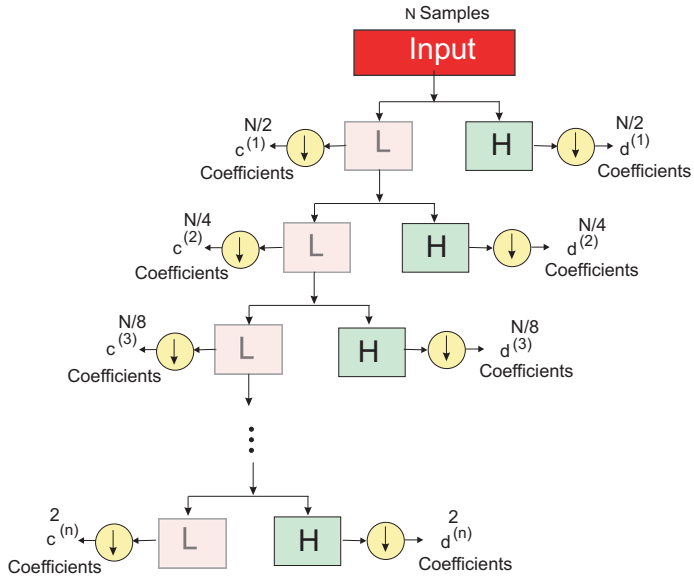
Because each filter in Figure 11.8 changes the relative strengths of the different frequency components, passing the signal through a series of filters is equivalent, in the wavelet sense, to analyzing the signal at different scales. This is the origin of the name "multiresolution analysis." Figure 11.8 shows how the pyramid algorithm passes the signal through a series of highpass filters (H) and then through a series of lowpass filters (L). Each filter changes the scale to that of the level below. Notice too, the circles containing ↓2 in Figure 11.8. This operation filters out half of the signal and so is called *subsampling* or *factor-of-2 decimation*. It is the way we keep the areas of each box in Figure 11.7 constant as we vary the scale and translation times. We consider subsampling further when we discuss the pyramid algorithm.

In summary, the DWT process decomposes the signal into *smooth* information stored in the low-frequency components and *detailed* information stored in the high-frequency components. Because *high-resolution* reproductions of signals require more information about details than about gross shape, the pyramid algorithm is an effective way to compress data while still maintaining high resolution (we implement compression in D). In addition, because components of different resolution are independent of each other, it is possible to lower the number of data stored by systematically eliminating higher-resolution components. The use of wavelet filters builds in progressive scaling, which is particularly appropriate for fractal-like reproductions.

11.5.1 Pyramid Scheme Implementation ⊙

We now wish to implement the pyramid scheme outlined in Figure 11.8. The filters L and H will be represented by matrices, which is an approximate way to perform the integrations or convolutions. Then there is a decimation of the output by one-half, and finally an interleaving of the output for further filtering. This process simultaneously cuts down on the number of

Figure 11.9 An input signal at the top is processed by a tree of high- and low-band filters. The outputs from each filtering are downsampled with every other data kept. The process continues until there are only two data of high-band filtering and two data of low-band filtering. The total number of output data equals the total number of signal data. The process of wavelet analysis/transorm is thus equivalent to a series of filterings.



points in the data set and changes the scale and the resolution. The decimation reduces the number of values of the remaining signal by one half, with the low-frequency part discarded because the details are in the high-frequency parts.

As indicated in Figure 11.9, the pyramid algorithm's DWT successively (1) applies the (soon-to-be-derived) c matrix (11.35) to the whole N-length vector,

$$\begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & -c_0 & c_3 & -c_2 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}, \tag{11.25}$$

(2) applies it to the $(N/2)$-length smooth vector, (3) and then repeats until two smooth components remain. (4) After each filtering, the elements are ordered, with the newest two smooth elements on top, the newest detailed elements below, and the older detailed elements below that. (5) The process continues until there are just two smooth elements left.

To illustrate, here we filter and reorder an initial vector of length $N = 8$:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{pmatrix} \xrightarrow{\text{filter}} \begin{pmatrix} s_1^{(1)} \\ d_1^{(1)} \\ s_2^{(1)} \\ d_2^{(1)} \\ s_3^{(1)} \\ d_3^{(1)} \\ s_4^{(1)} \\ d_4^{(1)} \end{pmatrix} \xrightarrow{\text{order}} \begin{pmatrix} s_1^{(1)} \\ s_2^{(1)} \\ s_3^{(1)} \\ s_4^{(1)} \\ \hline d_1^{(1)} \\ d_2^{(1)} \\ d_3^{(1)} \\ d_4^{(1)} \end{pmatrix} \xrightarrow{\text{filter}} \begin{pmatrix} s_1^{(2)} \\ d_1^{(2)} \\ s_2^{(2)} \\ d_2^{(2)} \\ \hline d_1^{(1)} \\ d_2^{(1)} \\ d_3^{(1)} \\ d_4^{(1)} \end{pmatrix} \xrightarrow{\text{order}} \begin{pmatrix} s_1^{(2)} \\ s_2^{(2)} \\ d_1^{(2)} \\ d_2^{(2)} \\ \hline d_1^{(1)} \\ d_2^{(1)} \\ d_3^{(1)} \\ d_4^{(1)} \end{pmatrix}. \tag{11.26}$$

The discrete inversion of a transform vector back to a signal vector is made using the transpose

(inverse) of the transfer matrix at each stage. For instance,

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & -c_2 & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & -c_0 & c_1 & -c_2 \end{pmatrix} \begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix}. \tag{11.27}$$

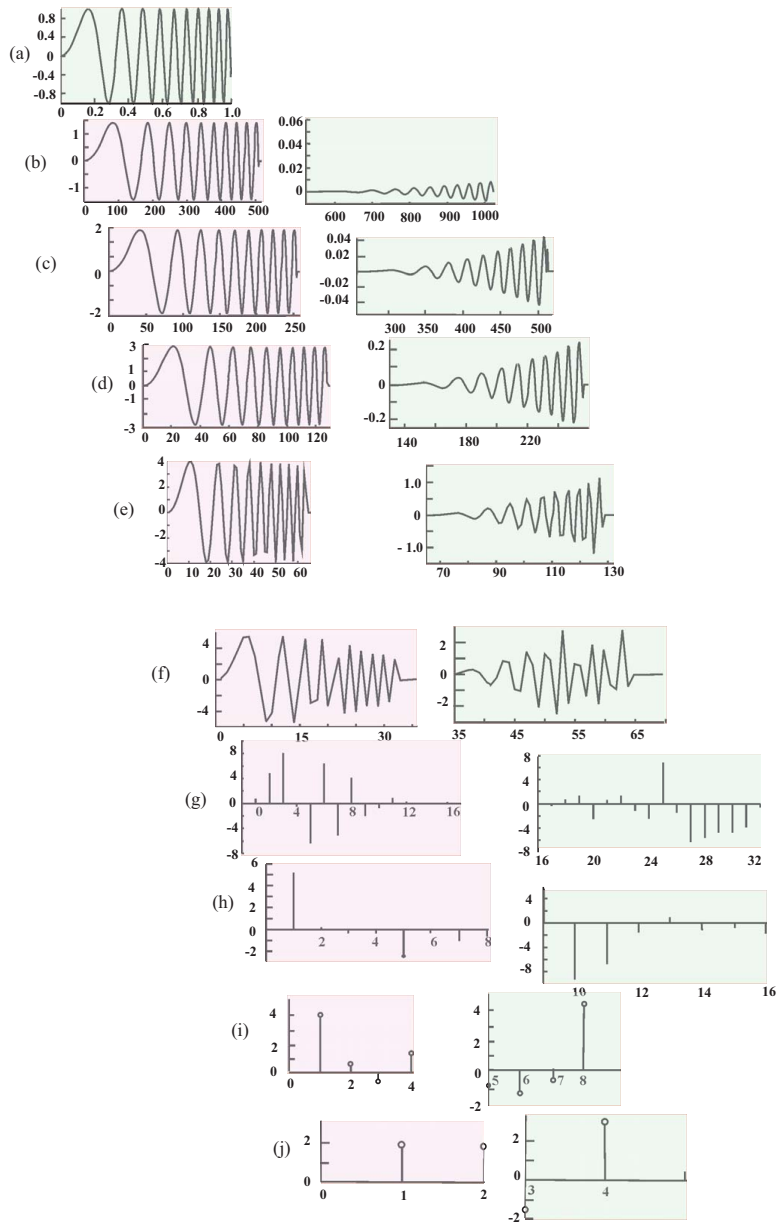
As a more realistic example, imagine that we have sampled the chirp signal $y(t) = \sin(60t^2)$ for 1024 times. The filtering process through which we place this signal is illustrated as a passage from the top to the bottom in Figure 11.9. First the original 1024 samples are passed through a single low band and a single high band (which is mathematically equivalent to performing a series of convolutions). As indicated by the down arrows, the output of the first stage is then downsampled (the number reduced by a factor of 2). This results in 512 points from the high-band filter as well as 512 points from the low-band filter. This produces the first-level output. The output coefficients from the high-band filters are called $\{d_i^{(1)}\}$ to indicate that they show details, and $\{s_i^{(1)}\}$ to indicate that they show smooth features. The superscript indicates that this is the first level of processing. The detail coefficients $\{d^{(1)}\}$ are stored to become part of the final output.

In the next level down, the 512 smooth data $\{s_i^{(1)}\}$ are passed through new low- and high-band filters using a broader wavelet. The 512 outputs from each are downsampled to form a smooth sequence $\{s_i^{(2)}\}$ of size 256 and a detailed sequence $\{d_i^{(2)}\}$ of size 256. Again the detail coefficients $\{d^{(2)}\}$ are stored to become part of the final output. (Note that this is only half the size of the previously stored details.) The process continues until there are only two numbers left for the detail coefficients and two numbers left for the smooth coefficients. Since this last filtering is done with the broadest wavelet, it is of the lowest resolution and therefore requires the least information.

In Figure 11.10 we show the actual effects on the chirp signal of pyramid filtering for various levels in the processing. (The processing is done with four-coefficient *Daub4* wavelets, which we will discuss soon.) At the uppermost level, the Daub4 wavelet is narrow, and so convoluting this wavelet with successive sections of the signal results in smooth components that still contain many large high-frequency parts. The detail components, in contrast, are much smaller in magnitude. In the next stage, the wavelet is dilated to a lower frequency and the analysis is repeated *on just the smooth (low-band) part.* The resulting output is similar, but with coarser features for the smooth coefficients and larger values for the details. Note that in the upper graphs we have connected the points to make the output look continuous, while in the lower graphs, with fewer points, we have plotted the output as histograms to make the points more evident. Eventually the downsampling leads to just two coefficients output from each filter, at which point the filtering ends.

To reconstruct the original signal (called *synthesis* or *transformation*) a reversed process is followed: Begin with the last sequence of four coefficients, upsample them, pass them through low- and high-band filters to obtain new levels of coefficients, and repeat until all the N values of the original signal are recovered. The inverse scheme is the same as the processing scheme (Figure 11.9), only now the direction of all the arrows is reversed.

Figure 11.10 The filtering of the original signal at the top goes through the pyramid algorithm and produces the outputs shown, in successive passes. The sampling is reduced by a factor of 2 in each step. Note that in the upper graphs we have connected the points to emphasize their continuous nature while in the lower graphs we plot the individual output points as histograms.



11.5.2 Daubechies Wavelets via Filtering

We should now be able to understand that digital wavelet analysis has been standardized to the point where classes of wavelet basis functions are specified not by their analytic forms but rather by their *wavelet filter coefficients*. In 1988, the Belgian mathematician Ingrid Daubechies discovered an important class of such filter coefficients [Daub 95]. We will study just the Daub4 class containing the four coefficients c_0, c_1, c_2, and c_3.

Imagine that our input contains the four elements $\{y_1, y_2, y_3, y_4\}$ corresponding to measurements of a signal at four times. We represent a lowpass filter L and a highpass filter H in terms of the four filter coefficients as

$$L = \begin{pmatrix} +c_0 & +c_1 & +c_2 & +c_3 \end{pmatrix} \tag{11.28}$$

$$H = \begin{pmatrix} +c_3 & -c_2 & +c_1 & -c_0 \end{pmatrix}. \tag{11.29}$$

To see how this works, we form an input vector by placing the four signal elements in a column and then multiply the input by L and H:

$$L \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} +c_0 & +c_1 & +c_2 & +c_3 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = c_0 y_0 + c_1 y_1 + c_2 y_2 + c_3 y_3,$$

$$\begin{aligned} H \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} &= \begin{pmatrix} +c_3 & -c_2 & +c_1 & -c_0 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \\ &= c_3 y_0 - c_2 y_1 + c_1 y_2 - c_0 y_3. \end{aligned}$$

We see that if we choose the values of the c_i's carefully, the result of L acting on the signal vector is a single number that may be viewed as a weighted average of the four input signal elements. Since an averaging process tends to smooth out data, the lowpass filter may be thought of as a *smoothing filter* that outputs the general shape of the signal.

In turn, we see that if we choose the c_i values carefully, the result of H acting on the signal vector is a single number that may be viewed as the weighted differences of the input signal. Since a differencing process tends to emphasize the variation in the data, the highpass filter may be thought of as a *detail* filter that produces a large output when the signal varies considerably, and a small output when the signal is smooth.

We have just seen how the individual L and H filters, each represented by a single row, output one number when acting upon an input signal containing four elements in a column. If we want the output of the filtering process Y to contain the same number of elements as the input (four y's in this case), we just stack the L and H filters together:

$$\begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} L \\ H \\ L \\ H \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & -c_0 & c_3 & -c_2 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}. \tag{11.30}$$

Of course the first and third rows of the Y vector will be identical, as will the second and fourth, but we will get to that soon.

Now we go about determining the values of the filter coefficients c_i by placing specific demands upon the output of the filter. We start by recalling that in our discussion of discrete Fourier transforms we observed that a transform is equivalent to a rotation from the time domain to the frequency domain. Yet we know from our study of linear algebra that rotations are described by orthogonal matrices, that is, matrices whose inverses are equal to their transposes. In order for the inverse transform to return us to the input signal, the transfer matrix must be orthogonal. For our wavelet transformation to be orthogonal, we must have the 4×4 filter matrix times its transpose equal to the identity matrix:

$$\begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & -c_0 & c_3 & -c_2 \end{pmatrix} \begin{pmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & -c_2 & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & -c_0 & c_1 & -c_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$\Rightarrow \quad c_0^2 + c_1^2 + c_2^2 + c_3^2 = 1, \quad c_2 c_0 + c_3 c_1 = 0. \tag{11.31}$$

Two equations in four unknowns are not enough for a unique solution, so we now include the further requirement that the detail filter $H = (c_3,\, -c_0,\, c_1,\, -c_2)$ must output a zero if the input is smooth. We define "smooth" to mean that the input is constant or linearly increasing:

$$\begin{pmatrix} y_0 & y_1 & y_2 & y_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} 0 & 1 & 2 & 3 \end{pmatrix}. \tag{11.32}$$

This is equivalent to demanding that the moments up to order p are zero, that is, that we have an "approximation of order p." Explicitly,

$$H\begin{pmatrix} y_0 & y_1 & y_2 & y_3 \end{pmatrix} = H\begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} = H\begin{pmatrix} 0 & 1 & 2 & 3 \end{pmatrix} = 0,$$

$$\Rightarrow \quad c_3 - c_2 + c_1 - c_0 = 0, \quad 0 \times c_3 - 1 \times c_2 + 2 \times c_1 - 3 \times c_0 = 0,$$

$$\Rightarrow \quad c_0 = \frac{1+\sqrt{3}}{4\sqrt{2}} \simeq 0.483, \quad c_1 = \frac{3+\sqrt{3}}{4\sqrt{2}} \simeq 0.836, \tag{11.33}$$

$$c_2 = \frac{3-\sqrt{3}}{4\sqrt{2}} \simeq 0.224, \quad c_3 = \frac{1-\sqrt{3}}{4\sqrt{2}} \simeq -0.129. \tag{11.34}$$

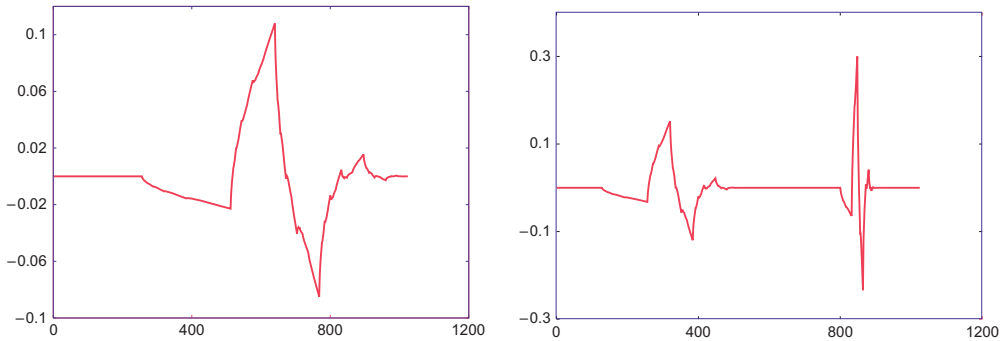
These are the basic Daub4 filter coefficients. They are used to create larger filter matrices by placing the row versions of L and H along the diagonal, with successive pairs displaced two columns to the right. For example, for eight elements,

$$\begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 & 0 & 0 & 0 & 0 \\ c_3 & -c_2 & c_1 & -c_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_0 & c_1 & c_2 & c_3 & 0 & 0 \\ 0 & 0 & c_3 & -c_2 & c_1 & -c_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_0 & c_1 & c_2 & c_3 \\ 0 & 0 & 0 & 0 & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & 0 & 0 & 0 & 0 & c_0 & c_1 \\ c_1 & -c_0 & 0 & 0 & 0 & 0 & c_3 & -c_2 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix}. \tag{11.35}$$

Note that in order not to lose any information, the last pair on the bottom two rows is wrapped over to the left. If you perform the actual multiplications indicated in (11.35), you will note that the output has successive *smooth* and *detailed* information. The output is processed with the pyramid scheme.

The time dependences of two Daub4 wavelets is displayed in Figure 11.11. To obtain these from our filter coefficients, first imagine that an elementary wavelet $y_{1,1}(t) \equiv \psi_{1,1}(t)$ is

Figure 11.11 *Left:* The Daub4 e6 wavelet constructed by inverse transformation of the wavelet coefficients. This wavelet has been found to be particularly effective in wavelet analyses. *Right:* The sum of Daub4 e10 and Daub4 1e58 wavelets of different scale and time displacements.



input into the filter. This should result in a transform $Y_{1,1} = 1$. Inversely, we obtain $y_{1,1}(t)$ by applying the inverse transform to a Y vector with a 1 in the first position and zeros in all the other positions. Likewise, the ith member of the Daubechies class is obtained by applying the inverse transform to a Y vector with a 1 in the ith position and zeros in all the other positions.

On the left in Figure 11.11 is the wavelet for coefficient 6 (thus the e6 notation). On the right in Figure 11.11 is the sum of two wavelets corresponding to the coefficients 10 and 58. We see that the two wavelets have different levels of scale as well as different time positions. So even though the time dependence of the wavelets is not evident when wavelet (filter) coefficients are used, it is there.

11.5.3 DWT Implementation and Exercise

Listing 11.2 gives our program for performing a DWT on the chirp signal $y(t) = \sin(60t^2)$. The method **`pyram`** calls the **`daube4`** method to perform the DWT or inverse DWT, depending upon the value of **`sign`**.

1. Modify the program so that you output to a file the values for the input signal that your code has read in. It is always important to check your input.
2. Try to reproduce the left of Figure 11.10 by using various values for the variable **`nend`** that controls when the filtering ends. A value **`nend=1024`** should produce just the first step in the downsampling (top row in Figure 11.10). Selecting **`nend=512`** should produce the next row, while **`nend=4`** should output just two smooth and detailed coefficients.
3. Reproduce the scale–time diagram shown on the right in Figure 11.10. This diagram shows the output at different scales and serves to interpret the main components of the signal and the time in which they appear. The time line at the bottom of the figure corresponds to a signal of length 1 over which 256 samples were recorded. The low-band (smooth) components are shown on the left, and the high-band components on the right.
   a. The bottommost figure results when **`nend`** $= 256$.
   b. The figure in the second row up results from **`nend`** $= 128$, and we have the output from two filterings. The output contains 256 coefficients but divides time into four intervals and shows the frequency components of the original signal in more detail.
   c. Continue with the subdivisions for **`nend`** $= 64, 32, 16, 8$, and 4.
4. For each of these choices except the topmost, divide the time by 2 and separate the

intervals by vertical lines.

5. The topmost spectrum is your final output. Can you see any relation between it and the chirp signal?
6. Change the sign of **`sign`** and check that the inverse DWT reproduces the original signal.
7. Use the code to visualize the time dependence of the Daubechies mother function at different scales.
   a. Start by performing an inverse transformation on the eight-component signal `[0,0,0,0,1,0,0,0]`. This should yield a function with a width of about 5 units.
   b. Next perform an inverse transformation on a unit vector with $N = 32$ but with all components except the fifth equal to zero. The width should now be about 25 units, a larger scale but still covering the same time interval.
   c. Continue this procedure until you obtain wavelets of 800 units.
   d. Finally, with $N = 1024$, select a portion of the mother wavelet with data in the horizontal interval [590,800]. This should show self-similarity similar to that at the bottom of Figure 11.11.

Listing 11.2 **DWT.py** computes the discrete wavelet transform using the pyramid algorithm for the 2^n signal values stored in `f[ ]` (here assigned as the chirp signal $\sin 60t^2$). The Daub4 digital wavelets are the basis functions, and `sign` = ± 1 for transform/inverse.

```
# DWT.py:  Discrete Wavelet Transform, Daubechies type, global variables

from visual.graph import *

sq3 = sqrt(3);            fsq2 = 4.0*sqrt(2);       N = 1024      # N = 2^n
c0 = (1+sq3)/fsq2;             c1 = (3+sq3)/fsq2       # Daubechies 4 coeff
c2 = (3-sq3)/fsq2;              c3 = (1-sq3)/fsq2
transfgr1 = None          # indicate that displays have not been made yet
transfgr1 = None

def chirp( xi):                                                # chirp signal
    y = sin(60.0*xi**2);
    return y;

def daube4(f, n, sign):            # DWT if sign >= 0, inverse if sign < 0
    global transfgr1, transfgr2
    tr = zeros( (n + 1), float)                          # temporary variable
    if n < 4 : return
    mp = n/2                                              # midpoint of array
    mp1 = mp + 1                                          # midpoint plus one
    if sign >= 0:                                                      # DWT
        j = 1
        i = 1
        maxx  = n/2
        if n > 128:                                       # appropiate scales
            maxy = 3.0
            miny = - 3.0
            Maxy = 0.2
            Miny = - 0.2
            speed = 50                                            # fast rate
        else:
            maxy = 10.0
            miny = - 5.0
            Maxy = 7.5
            Miny = - 7.5
            speed = 8                                       # for lower rate
        if transfgr1:
            transfgr1.display.visible = False
            transfgr2.display.visible = False
            del transfgr1
            del transfgr2
        transfgr1 = gdisplay(x=0, y=0, width=600, height=400,\
                  title='Wavelet TF, down sample + low pass', xmax=maxx,\
                        xmin=0, ymax=maxy, ymin=miny)
        transf  = gvbars(delta=2.*n/N, color=color.cyan, display=transfgr1)
        transfgr2 = gdisplay(x=0, y=400, width=600, height=400,\
                       title='Wavelet TF, down sample + high pass',\
                         xmax=2*maxx, xmin=0, ymax=Maxy, ymin=Miny)
        transf2 = gvbars(delta=2.*n/N, color=color.cyan, display=transfgr2)
        while j <= n - 3:
            rate(speed)
            tr[i] = c0*f[j] + c1*f[j+1] + c2*f[j+2] + c3*f[j+3]# low-pass
            transf.plot(pos = (i, tr[i]) )                  # c coefficients
```

```
            tr[i+mp] = c3*f[j] - c2*f[j+1] + c1*f[j+2] - c0*f[j+3] # high
            transf2.plot(pos = (i + mp, tr[i + mp]) )
            i += 1                                          # d coefficents
            j += 2                                           # downsampling
        tr[i] = c0*f[n-1] + c1*f[n] + c2*f[1] + c3*f[2]          # low-pass
        transf.plot(pos = (i, tr[i]) )                     # c coefficients
        tr[i+mp] = c3*f[n-1] - c2*f[n] + c1*f[1] - c0*f[2]     # high-pass
        transf2.plot(pos = (i+mp, tr[i+mp]) )
    else:                                                   # inverse DWT
        tr[1] = c2*f[mp] + c1*f[n] + c0*f[1] + c3*f[mp1]        # low-pass
        tr[2] = c3*f[mp] - c0*f[n] + c1*f[1] - c2*f[mp1]       # high-pass
        j = 3
        for i in range (1, mp):
            tr[j] = c2*f[i] + c1*f[i+mp] + c0*f[i+1] + c3*f[i+mp1]   # low
            j += 1                                                  # upsample
            tr[j] = c3*f[i] - c0*f[i+mp] + c1*f[i+1] - c2*f[i+mp1]  # high
            j += 1;                                               # upsampling
    for i in range(1, n+1):
        f[i] = tr[i]                                     # copy TF to array

def pyram(f, n, sign):                               # DWT, replaces f by TF
    if (n < 4): return                                       # too few data
    nend = 4                                         # indicates when to stop
    if sign >= 0 :                                            # Transform
        nd = n
        while nd >= nend:                            # Downsample filtering
            daube4(f, nd, sign)
            nd //= 2
    else:                                                    # Inverse TF
        for nd in range (4, n + 1):                          # Upsampling
            daube4(f , nd, sign)
            nd *= 2
f = zeros( (N + 1), float)                                   # data vector
inxi = 1.0/N                                             # for chirp signal
xi = 0.0
for i in range(1, N + 1):
    f[i] = chirp(xi)                                     # Function to TF
    xi   += inxi;
n = N                                                        # must be 2^m
pyram(f, n, 1)                                                       # TF
# pyram(f, n, -1)                                            # Inverse TF
```

Chapter Twelve

Discrete & Continuous Nonlinear Dynamics

Nonlinear dynamics is one of the success stories of computational science. It has been explored by mathematicians, scientists, and engineers, with computers as an essential tool. The computations have led to the discovery of new phenomena such as solitons, chaos and fractals, as you will discover 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 those of other complex systems.

In Unit I we develop the logistic 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 Unit II we explore chaos for a continuous system, the driven realistic pendulum. Our emphasis there is on using phase space as an example of the usefulness of an abstract space to display the simplicity underlying complex behavior. In Unit III we extend the discrete logistic map to nonlinear differential models of coupled predator–prey populations and their corresponding phase space plots.

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Bugs	pdf	12.1	Chaotic Pend I	pdf	12.10
Chaotic Pend II	pdf	12.12	Chaotic Pend III	pdf	12.14

Applets and Animations Applet

Name	*Sections*	*Name*	*Sections*
The Chaotic Pendulum	12.11	Hypersensitive Pendula	12.11
Double Pendulum Movie	12.14	Classical Chaotic Scattering	9.14
HearData: Data2Sound	12.13	Lissajous Figures	12.12
Visualizing with Sound	12.13		

12.1 UNIT I. BUG POPULATION DYNAMICS (DISCRETE)

Problem: The populations of insects and the patterns of weather do not appear to follow any simple laws.[1] 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, discrete law can produce such complicated behavior.

12.2 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

[1] Except maybe in Oregon, where storm clouds come to spend their weekends.

N_i bugs to bug us. We want to define a model of how N_n varies with the discrete generation number n. For guidance, we look to the radioactive decay simulation in Chapter 5, "Monte Carlo Simulations", where the discrete decay law, $\Delta N/\Delta t = -\lambda N$, led to exponential-like decay. Likewise, if we reverse the sign of λ, we should get exponential-like growth, which is a good place to start our modelling. We assume that the bug-breeding rate is proportional to the number of bugs:

$$\frac{\Delta N_i}{\Delta t} = \lambda\, N_i. \tag{12.1}$$

Because we know as an empirical fact that exponential growth usually tapers off, we improve the model by incorporating the observation that bugs do not live on love alone; they must also eat. But bugs, not being 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 (12.1) by introducing a growth rate λ' that decreases as the population N_i approaches N_*:

$$\lambda = \lambda'(N_* - N_i) \quad \Rightarrow \quad \frac{\Delta N_i}{\Delta t} = \lambda'(N_* - N_i)N_i. \tag{12.2}$$

We expect that when N_i is small compared to N_*, the population will grow exponentially, but that as N_i approaches N_*, the growth rate will decrease, eventually becoming negative if N_i exceeds N_*, the carrying capacity.

Equation (12.2) is one form of the *logistic map*. It is usually written as a relation between the number of bugs in future and present generations:

$$N_{i+1} = N_i + \lambda'\,\Delta t(N_* - N_i)N_i, \tag{12.3}$$

$$= N_i\left(1 + \lambda'\,\Delta t N_*\right)\left[1 - \frac{\lambda'\,\Delta t}{1 + \lambda'\,\Delta t N_*}N_i\right]. \tag{12.4}$$

The map looks simple when expressed in terms of natural variables:

$$x_{i+1} = \mu x_i(1 - x_i), \tag{12.5}$$

$$\mu \stackrel{\text{def}}{=} 1 + \lambda'\,\Delta t N_*, \quad x_i \stackrel{\text{def}}{=} \frac{\lambda'\,\Delta t}{\mu}N_i \simeq \frac{N_i}{N_*}, \tag{12.6}$$

where μ is a dimensionless growth parameter and x_i is a dimensionless population variable. Observe that the *growth rate* μ equals 1 when the breeding rate λ' equals 0, and is otherwise expected to be larger than 1. If the number of bugs born per generation $\lambda'\,\Delta t$ is large, then $\mu \simeq \lambda'\,\Delta t N_*$ and $x_i \simeq N_i/N_*$. That is, x_i is essentially a fraction of the carrying capacity N_*. Consequently, we consider x values in the range $0 \le x_i \le 1$, 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 (12.5) on x_i. In general, a map uses a function $f(x)$ to map one number in a sequence to another,

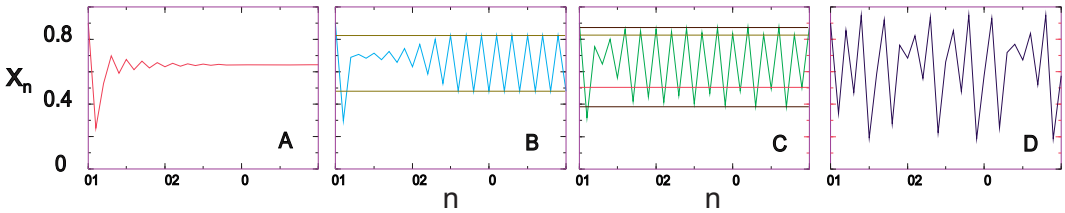
$$x_{i+1} = f(x_i). \tag{12.7}$$

For the logistic map, $f(x) = \mu x(1 - x)$, with the quadratic dependence of f on x making this a nonlinear map, while the dependence on only the one variable x_i makes it a *one-dimensional* map.

12.3 PROPERTIES OF NONLINEAR MAPS (THEORY)

Rather than do some fancy mathematical analysis to determine the properties of the logistic map [Rash 90], 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 Figure 12.1. In 12.1A we see

Figure 12.1 The insect population x_n *versus* the generation number n for four different growth rates: (A) $\mu = 2.8$, a single attractor; (B) $\mu = 3.3$, a double attractor; (C) $\mu = 3.5$, a quadruple attractor; (D) $\mu = 3.8$, a chaotic regime.



equilibration into a single population; in 12.1B we see oscillation between two population levels; in 12.1C we see oscillation among four levels; and in 12.1D 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 usually has little effect on the population dynamics (similar to what we found when generating pseudorandom numbers). In contrast, the dynamics are unusually sensitive to the value of the growth parameter μ. For those values of μ at which the dynamics are complex, there may be extreme sensitivity to the initial condition x_0 as well as to the exact value of μ.

12.3.1 Fixed Points

An important property of the map (12.5) 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 $i+1$; that is,

$$x_{i+1} = x_i = x_*. \tag{12.8}$$

Using the logistic map (12.5) to relate x_{i+1} to x_i yields the algebraic equation

$$\mu x_*(1 - x_*) = x_* \quad \Rightarrow \quad x_* = 0 \quad \text{or} \quad x_* = \frac{\mu - 1}{\mu}. \tag{12.9}$$

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 (Figure 12.1A). 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 (§12.8) tells us that the stability of a population is determined by the magnitude of the derivative of the mapping function $f(x_i)$ at the fixed point [Rash 90]:

$$\left| \frac{df}{dx} \right|_{x_*} < 1 \quad \text{(stable)}. \tag{12.10}$$

For the one cycle of the logistic map (12.5), we have

$$\left. \frac{df}{dx} \right|_{x_*} = \mu - 2\mu x_* = \begin{cases} \mu, & \text{stable at } x_* = 0 \text{ if } \mu < 1, \\ 2 - \mu, & \text{stable at } x_* = \frac{\mu - 1}{\mu} \text{ if } \mu < 3. \end{cases} \tag{12.11}$$

12.3.2 Period Doubling, Attractors

Equation (12.11) tells us that while the equation for fixed points (12.9) may be satisfied for all values of μ, 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* (Figure 12.1B). 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 demanding that generation $i+2$ have the same population as generation i:

$$x_i = x_{i+2} = \mu x_{i+1}(1 - x_{i+1}) \quad \Rightarrow \quad x_* = \frac{1 + \mu \pm \sqrt{\mu^2 - 2\mu - 3}}{2\mu}. \tag{12.12}$$

We see that as long as $\mu > 3$, the square root produces a real number and thus that physical solutions exist (complex or negative x_* values are nonphysical). We leave it to your computer explorations to discover how the system continues to double periods as μ continues to increase. In all cases the pattern is the same: One of the populations bifurcates into two.

12.4 MAPPING IMPLEMENTATION

Program 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 [Feig 79] of the different behavior patterns shown in Figure 12.1. 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:

1. **Transients:** Irregular behaviors before reaching a steady state that differ for different seeds.
2. **Asymptotes:** In some cases the steady state is reached after only 20 generations, while for larger μ 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 (12.9).
5. **Multiple cycles:** Examine the map orbits for a growth parameter μ increasing continuously through 3. Observe how the system continues to double periods as μ increases. To illustrate, in Figure 12.1C 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 appears stable for a finite number of generations and then jumps 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, mean 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 values of μ and x_0. Systems may start out with nearly identical values for μ and x_0 but end up with quite different ones. 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.[2]

   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 $\epsilon \simeq 2 \times 10^{-14}$.

[2] You may recall from Chapter 5, "Monte Carlo Simulations," that a random sequence of events does not even have step-by-step correlations.

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.

12.5 BIFURCATION DIAGRAM (ASSESSMENT)

Computing and watching the population change with generation number gives a good idea of the basic dynamics, at least until it gets too complicated to discern patterns. In particular, as the number of bifurcations keeps increasing and the system becomes chaotic, it is hard for us 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 μ is an elegant way to summarize the results of extensive computer simulations.

A *bifurcation diagram* for the logistic map is given in Figure 12.2, while one for a Gaussian map is given in Figure 12.3. For each value of μ, hundreds of iterations are made to make sure that all transients essentially die out, and then the values (μ, x_*) are written to a file for hundreds of iterations after that. If the system falls into an n cycle for this μ 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 is repeated to ensure that no fixed points are missed. When finished, your program will have stepped through all the values of growth parameter μ, and for each value of μ it will have stepped through all the values of the initial population x_0. Our sample program **Bugs.py** is shown in Listing 12.1.

Listing 12.1 **Bugs.py** produces the bifurcation diagram of the logistic map. A finished program requires finer grids, a scan over initial values, and removal of duplicates.

```
# Bugs.py creates bifurcation diagram for Logistic Map

from visual.graph import *

m_min = 1.0;     m_max = 4.0;      step = 0.025

graph1 = gdisplay(width = 600, height = 400, title = 'Logistic map', xtitle = 'm',
                  ytitle = 'x', xmax = 4.0, xmin = 1.0, ymax = 1.0, ymin = 0.0)
pts = gdots(shape = 'square', size = 1, color = color.green)
lasty = int(1000 * 0.5)                  # to eliminate later some points
count = 0                              # to plot later every two iterations
for m in arange(m_min, m_max, step):
    y = 0.5
    for i in range(1,401,1):                          # to avoid transients
        y = m*y*(1-y)
    for i in range(201, 402, 1):                      # to avoid transients
        y = m*y*(1 - y)
        inty = int(1000 * y)
        if inty != lasty and count%2==0:  pts.plot(pos=(m,y))# no repeats
        lasty = inty
        count   += 1
```

12.5.1 Bifurcation Diagram Implementation

The last part of this problem is to reproduce Figure 12.2 at various levels of detail. (You can listen to a sonification of this diagram use one of the applet there to create your own sonification (see tabulated links at beginning of chapter). 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

Figure 12.2 A bifurcation plot of attractor population x^* *versus* growth rate μ for the logistic map. The inset shows some details of a three-cycle window. (The colors indicate the regimes over which we distributed the work on different CPUs.)

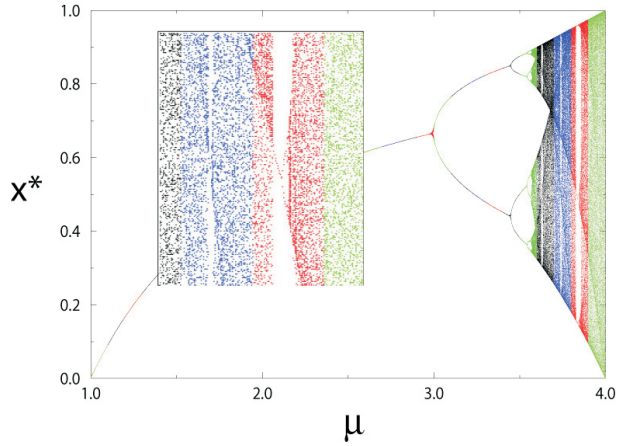
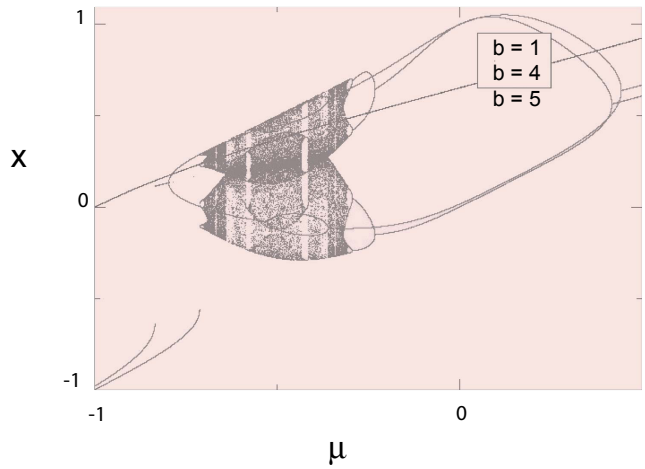


Figure 12.3 A bifurcation plot for the Gaussian map of Table 12.1. (Courtesy of W. Hager.)



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 plot a point at each pixel, you will 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 a finer resolution is a waste of time.

12.5.2 Visualization Algorithm: Binning

1. Break up the range $1 \leq \mu \leq 4$ into 1000 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 the next several hundred values of (μ, x_*) to a file.
4. Print 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 the output with Java's standard print commands (although **`printf`** and **`DecimalFormat`** do 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 $0 \le x_n \le 1$, this means that $0 \le 100 * x_n \le 1000$, 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, μ) from your file (they just take up space and plot on top of each other). You can do that in Unix/Linux with the *sort -u* command.
6. Plot your file of x_* *versus* μ. Use small symbols for the points and do not connect them.
7. Enlarge (zoom in on) 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

$$\mu_k \simeq 3,\ 3.449,\ 3.544,\ 3.5644,\ 3.5688,\ 3.569692,\ 3.56989,\ \ldots. \tag{12.13}$$

   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 μ values to see the structures.
10. A close examination of Figure 12.2 shows regions where, with a slight increase in μ, 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.

12.5.3 Feigenbaum Constants (Exploration)

Feigenbaum discovered that the sequence of μ_k values (12.13) at which bifurcations occur follows a regular pattern [Feig 79]. Specifically, it converges geometrically when expressed in terms of the distance between bifurcations δ:

$$\mu_k \;\to\; \mu_\infty - \frac{c}{\delta^k}, \quad \delta = \lim_{k\to\infty} \frac{\mu_k - \mu_{k-1}}{\mu_{k+1} - \mu_k}. \tag{12.14}$$

Use your sequence of μ_k values to determine the constants in (12.14) and compare them to those found by Feigenbaum:

$$\mu_\infty \simeq 3.56995, \quad c \simeq 2.637, \quad \delta \simeq 4.6692. \tag{12.15}$$

Amazingly, the value of the *Feigenbaum constant* δ is universal for all second-order maps.

12.6 LOGISTIC MAP RANDOM NUMBERS (EXPLORATION) ⊙

There are claims that the logistic map in the chaotic region ($\mu \ge 4$),

$$x_{i+1} \simeq 4x_i(1 - x_i), \tag{12.16}$$

can be used to generate random numbers [P&R 95]. Although successive x_i's are correlated, if the population for approximately every sixth generation is examined, the correlation is effectively gone and random numbers result. To make the sequence more uniform, a trigonometric

Table 12.1 Several Nonlinear Maps to Explore

Name	$f(x)$	Name	$f(x)$
Logistic	$\mu x(1-x)$	Tent	$\mu(1-2\,\lvert x-1/2\rvert)$
Ecology	$xe^{\mu(1-x)}$	Quartic	$\mu[1-(2x-1)^4]$
Gaussian	$e^{bx^2}+\mu$		

transformation is used:

$$y_i = \frac{1}{\pi}\cos^{-1}(1-2x_i). \tag{12.17}$$

Use the random-number tests discussed in Chapter 5, "Monte Carlo Simulation," to test this claim.

12.7 OTHER MAPS (EXPLORATION)

Bifurcations and chaos are characteristic properties of nonlinear systems. Yet systems can be nonlinear in a number of ways. Table 12.1 lists four maps that generate x_i sequences containing bifurcations. The tent map derives its nonlinear dependence from the absolute value operator, while the logistic map is a subclass of the ecology map. Explore the properties of these other maps and note the similarities and differences.

12.8 SIGNALS OF CHAOS: LYAPUNOV COEFFICIENTS ⊙

The Lyapunov coefficient λ_i provides an analytic measure of whether a system is chaotic [Wolf 85, Ram 00, Will 97]. Physically, the coefficient is a measure of the growth rate of the solution near an attractor. For 1-D maps there is only one such coefficient, whereas in general there is a coefficient for each direction in space. The essential assumption is that neighboring paths x_n near an attractor have an n (or time) dependence $L \propto \exp(\lambda t)$. Consequently, orbits that have $\lambda > 0$ diverge and are chaotic; orbits that have $\lambda = 0$ remain marginally stable, while orbits with $\lambda < 0$ are periodic and stable. Mathematically, the Lyapunov coefficient or exponent is defined as

$$\lambda = \lim_{t\to\infty}\frac{1}{t}\log\frac{L(t)}{L(t_0)}, \tag{12.18}$$

where $L(t)$ is the distance between neighboring phase space trajectories at time t.

We calculate the Lyapunov exponent for a general 1-D map,

$$x_{n+1} = f(x_n), \tag{12.19}$$

and then apply the result to the logistic map. To determine stability, we examine perturbations about a reference trajectory x_0 by adding a small perturbation and iterating once [Mann 90, Ram 00]:

$$\hat{x}_0 = x_0 + \delta x_0, \quad \hat{x}_1 = x_1 + \delta x_1. \tag{12.20}$$

We substitute this into (12.19) and expand f in a Taylor series around x_0:

$$x_1 + \delta x_1 = f(x_0+\delta x_0) \simeq f(x_0) + \left.\frac{\delta f}{\delta x}\right|_{x_0}\delta x_0 = x_1 + \left.\frac{\delta f}{\delta x}\right|_{x_0}\delta x_0,$$

$$\Rightarrow \quad \delta x_1 \simeq \left(\frac{\delta f}{\delta x}\right)_{x_0}\delta x_0. \tag{12.21}$$

(This is the proof of our earlier statement about the stability of maps.) To deduce the general result we examine one more iteration:

$$\delta x_2 \simeq \left(\frac{\delta f}{\delta x}\right)_{x_1} \quad \delta x_1 = \left(\frac{\delta f}{\delta x}\right)_{x_0} \left(\frac{\delta f}{\delta x}\right)_{x_1} \delta x_0, \tag{12.22}$$

$$\Rightarrow \quad \delta x_n = \prod_{i=0}^{n-1} \left(\frac{\delta f}{\delta x}\right)_{x_i} \delta x_0. \tag{12.23}$$

This last relation tells us how trajectories differ on the average after n steps:

$$|\delta x_n| = L^n |\delta x_0|, \quad L^n = \prod_{i=0}^{n-1} \left| \left(\frac{\delta f}{\delta x}\right)_{x_i} \right|. \tag{12.24}$$

We now solve for the Lyapunov number L and take its logarithm to obtain the Lyapunov exponent:

$$\lambda = \ln(L) = \lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} \ln \left| \left(\frac{\delta f}{\delta x}\right)_{x_i} \right|. \tag{12.25}$$

For the logistic map we obtain

$$\lambda = \frac{1}{n} \sum_{i=0}^{n-1} \ln |\mu - 2\mu x_i|, \tag{12.26}$$

where the sum is over iterations.

The code **`LyapLog.py`** in Listing 12.2 computes the Lyapunov exponents for the bifurcation plot of the logistic map. In Fig. 12.4 left we show its output, and note that the sign changes in λ where the system becomes chaotic, and abrupt changes in slope at bifurcations. (A similar curve is obtained for the fractal dimension of the logistic map as, indeed, the two are proportional.) In Figure 12.4 left we show its output and note the sign changes in λ where the system becomes chaotic, and abrupt changes in slope at bifurcations. (A similar curve is obtained for the fractal dimension of the logistic map, as indeed the two are proportional.)

12.8.1 Shannon Entropy ⊙

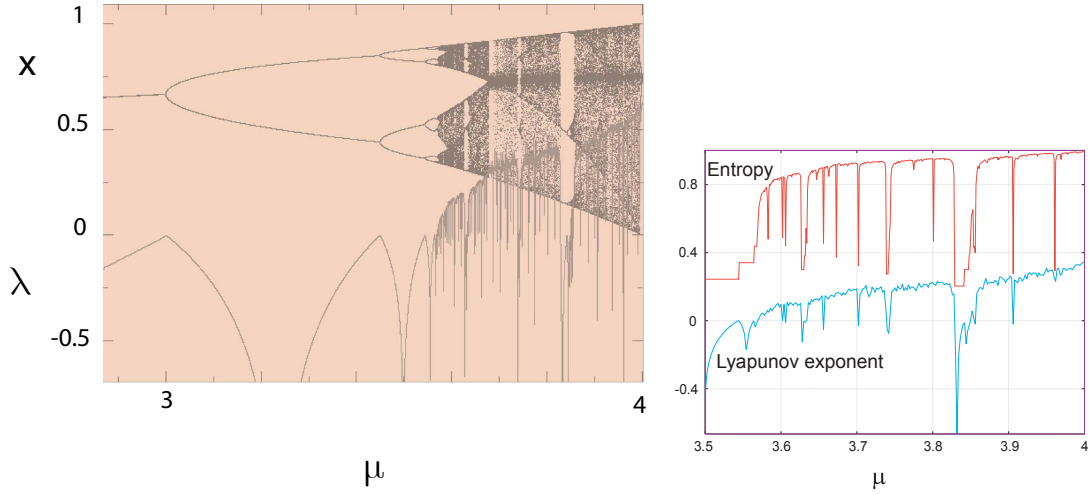
Shannon entropy, like the Lyapunov exponent, is another analytic measure of chaos. It is a measure of uncertainty that has proven useful in communication theory [Shannon 48, Ott 02, G,T&C 06] and led to the concept of information entropy. Imagine that an experiment has N possible outcomes. If the probability of each is $p_1, p_2, \ldots, p_N$, with normalization such that $\sum_{i=1}^{N} p_i = 1$, the Shannon entropy is defined as

$$S_{\text{Shannon}} = -\sum_{i=1}^{N} p_i \, ln \, p_i. \tag{12.27}$$

Listing 12.2 **LyapLog.py** computes Lyapunov exponents for the bifurcation plot of the logistic map as a function of growth rate. Note the fineness of the μ grid.

```
# LyapLog.py:                    Lyapunov coef for logistic map

from visual.graph import *

m_min = 2.1;          m_max = 4.0;          step = 0.25
graph1 = gdisplay( title = 'Lyapunov coef (blue) for logistic map (red)',
```

Figure 12.4 *Left:* Lyapunov exponent (top) and bifurcation values (bottom) for the logistic map as functions of the growth rate μ. Notice how the Lyapunov exponent, whose value is a measure of chaos, changes abruptly at the bifurcations. *Right:* Shannon entropy reduced by a factor of 5 (top) and the Lyapunov coefficient for the logistic map (bottom). Notice the close relation between the thermodynamic measure of disorder (entropy) and the nonlinear dynamics measure of chaos (Lyapunov).




```
                    xtitle = 'm', ytitle = 'x , Lyap',
                    xmax=4.0,xmin=0,ymax = 1.0, ymin = - 0.6)
funct1 = gdots(color = color.red)
funct2 = gcurve(color = color.yellow)

for m in arange(m_min, m_max, step):                                    # m loop
    y = 0.5
    suma = 0.0
    for i in range(1, 401, 1):   y = m*y*(1 - y)                  # Skip transients
    for i in range(402, 601, 1):
        y = m*y*(1 - y)
        funct1.plot(pos = (m, y) )
        suma = suma  +  log(abs(m*(1. - 2.*y) ))                   # Lyapunov
    funct2.plot(pos = (m, suma/401) )                                   # Normalize
```

If $p_i \equiv 0$, there is no uncertainty and $S_{\text{Shannon}} = 0$, as you might expect. If all N outcomes have equal probability, $p_i \equiv 1/N$, we obtain $S_{\text{Shannon}} = \ln N$, which diverges slowly as $N \to \infty$.

The code **`Entropy.py`** in Listing 12.3 computes the Shannon entropy for the the logistic map as a function of the growth parameter μ. The results (Figure 12.4 left) are seen to be quite similar to the Lyapunov exponent, again with discontinuities occurring at the bifurcations.

12.9 UNIT I QUIZ

1. Consider the logistic map.
   a. Make sketches of what a graph of population x_i *versus* generation number i would look like for extinction and for a period-two cycle.
   b. Describe in words and possibly a diagram the relation between the preceding two sketches and the bifurcation plot of x_i *versus* i.
2. Consider the *tent map*. Rather than compute this map, study it with just a piece of paper.
   a. Make sketches of what a graph of population x_i *versus* generation number i would look like for extinction, for a period-one cycle, and for a period-two cycle.
   b. Show that there is a single fixed point for $\mu > 1/2$ and a period-two cycle for $\mu > 1$.

Listing 12.3 **Entropy.py** computes the Shannon entropy for the logistic map as a function of growth parameter μ.

```
# Entropy.py Shannon Entropy with Logistic map using Tkinter

try:
    from tkinter import *
except:
    from Tkinter import *
import math
from numpy import zeros, arange

global Xwidth, Yheight
root = Tk( );   root.title('Entropy versus mu ')
mumin = 3.5;  mumax = 4.0;  dmu = 0.005; nbin = 1000;  nmax = 100000
prob = zeros( (1000), float)
minx = mumin;  maxx = mumax;  miny=0; maxy=2.5; Xwidth=500; Yheight=500

c = Canvas(root, width = Xwidth, height = Yheight)    # initialize canvas
c.pack()                                                          # pack canvas

Button(root, text = 'Quit', command = root.quit).pack()   # to begin quit

def world2sc(xl, yt, xr, yb): # x - left, y - top, x - right, y - bottom
    ''' maxx: window width          maxy: window height
    rm:  right margin           bm: bottom margin
    lm:  left margin            tm: right margin
    bx, mx, by, my: global constants for linear transformations
    xcanvas = mx*xworld + mx     ycanvas = my*yworld + my
    from world (double) to window (int) coordinates
    '''

    maxx = Xwidth   # canvas width          -------------------------
    maxy = Yheight  # canvas height        |      |     |tm         |
    lm = 0.10*maxx  # left margin          |   ---|----|------- ---|
    rm = 0.90*maxx  # right margin         |lm |  |            |   |
    bm = 0.85*maxy  # bottom margin        |---|  |            |   |
    tm = 0.10*maxy  # top margin           |-- |--|------------|   |
    mx = (lm - rm)/(xl - xr) #             |   |  bm         rm|   |
    bx = (xl*rm - xr*lm)/(xl - xr) #       |   |--|------------|   |
    my = (tm - bm)/(yt - yb)       #       |                       |
    by = (yb*tm - yt*bm)/(yb - yt) #       |-----------------------|
    linearTr = [mx, bx, my, by]                              # (maxx, maxy)
    return linearTr          # returns a list with 4 elements

# Plot y, x, axes; world coord converted to canvas coordinates
def xyaxis(mx, bx, my, by):            # to be called after call workd2sc
    x1 = (int)(mx*minx + bx)             # minima and maxima converted to
    x2 = (int)(mx*maxx + bx)                          # canvas coordinades
    y1  = (int)(my*maxy + by)
    y2 = (int)(my*miny + by)
    yc  = (int)(my*0.0 + by)
    c.create_line(x1, yc, x2, yc, fill = "red")            # plots x axis
    c.create_line(x1, y1, x1, y2, fill = 'red')          # plots y - axis

    for i in range (7):                                  # to plot x tics
       x = minx + (i - 1)*0.1                          # world coordinates
       x1 = (int)(mx*x + bx)                     # convert to canvas coord
       x2 = (int)(mx*minx + bx)
       y = miny + i*0.5                                # real coordinates
       y2 = (int)(my*y + by)                     # convert to canvas coords
       c.create_line(x1, yc - 4, x1, yc + 4, fill = 'red')      # tics  x
       c.create_line(x2 - 4, y2, x2 + 4, y2, fill = 'red')      # tics  y
       c.create_text(x1 + 10, yc + 10, text = '%5.2f'% (x), \
                     fill = 'red', anchor = E)                   # x axis
       c.create_text(x2 + 30, y2, text = '%5.2f'% (y),\
                     fill = 'red', anchor = E)                   # y axis
    c.create_text(70, 30, text = 'Entropy', fill = 'red', anchor = E) # y
    c.create_text(420, yc - 10, text = 'mu', fill = 'red', anchor = E)# x

mx, bx, my, by = world2sc(minx, maxy, maxx, miny)         # return a list
xyaxis(mx, bx, my, by)                                  # give values to axis
mu0 = mumin*mx + bx                                       # for the beginning
entr0 = my*0.0 + by                                  # first coord. mu0, entr0
for mu in arange(mumin, mumax, dmu):                                # mu loop
    print(mu)
    for j in range(1, nbin):
        prob[j] = 0
    y  = 0.5
    for n in range(1, nmax + 1):
        y = mu*y*(1.0 - y) # Logistic map, Skip transients
        if (n > 30000):
            ibin = int(y*nbin)  +  1
            prob[ibin]  +=  1
    entropy = 0.
```

```
    for ibin in range(1, nbin):
        if (prob[ibin]>0):
            entropy = entropy - (prob[ibin]/nmax)*math.log10(prob[ibin]/nmax)
    entrpc = my*entropy + by                    # entropy to canvas coords
    muc = mx*mu + bx                                # mu to canvas coords
    c.create_line(mu0, entr0, muc, entrpc, width = 1, fill = 'blue')
    mu0 = muc                                  # begin values for next line
    entr0 = entrpc
root.mainloop()                                     # makes effective events
```

12.10 UNIT II. PENDULUMS BECOME CHAOTIC

In Unit I on bugs we discovered that a simple nonlinear difference equation yields solutions that may be simple, complicated, or chaotic. Unit III will extend that model to the differential form, which also exhibits complex behaviors. Now we search for similar nonlinear, complex behaviors in the differential equation describing a realistic pendulum. 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 on the use of phase space plots for revealing the beauty and simplicity underlying complicated behaviors. We also provide two chaotic pendulum applets (Figure 12.5)for assistance in understanding the new concepts. Our study is based on the description in [Rash 90], on the analytic discussion of the parametric oscillator in [L&L,M 76], and on a similar study of the vibrating pivot pendulum in [G,T&C 06].

Consider the pendulum on the left in Figure 12.5. We see a pendulum of length l *driven* by an external sinusoidal torque $\mathbf{f}$ through air with a coefficient of drag α. Because there is no restriction that the angular displacement θ be small, we call this a *realistic* pendulum. Your **problem** is to describe the motion of this pendulum, first when the driving torque is 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.

12.11 CHAOTIC PENDULUM ODE

What we call a *chaotic pendulum* is just a pendulum with friction and a driving torque (Figure 12.5 left) but with no small-deflection-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 [Rash 90]:

$$I\,\frac{d^2\theta}{dt^2} = -mgl\,\sin\theta - \beta\,\frac{d\theta}{dt} + \tau_0\cos\omega t, \tag{12.28}$$

$$\Rightarrow \quad \frac{d^2\theta}{dt^2} = -\omega_0^2\,\sin\theta - \alpha\,\frac{d\theta}{dt} + f\cos\omega t, \tag{12.29}$$

$$\text{where} \quad \omega_0 = \frac{mgl}{I}, \quad \alpha = \frac{\beta}{I}, \quad f = \frac{\tau_0}{I}. \tag{12.30}$$

Equation (12.29) is a second-order time-dependent nonlinear differential equation. Its nonlinearity arises from the $\sin\theta$, as opposed to the θ, dependence of the gravitational torque. The parameter ω_0 is the natural frequency of the system arising from the restoring torque, α is a measure of the strength of friction, and f is a measure of the strength of the driving torque. In our standard ODE form, $d\mathbf{y}/dt = \mathbf{y}$ (Chapter 9, "Differential Equation Applications"), we

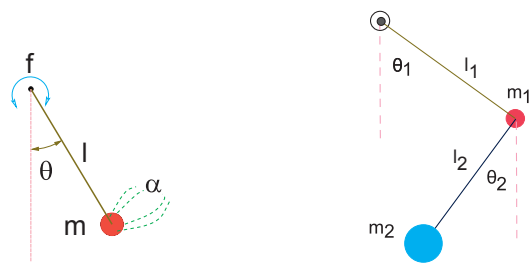
have two simultaneous first-order equations:

$$\frac{dy^{(0)}}{dt} = y^{(1)}, \tag{12.31}$$

$$\frac{dy^{(1)}}{dt} = -\omega_0^2 \sin y^{(0)} - \alpha y^{(1)} + f \cos \omega t,$$

$$\text{where} \quad y^{(0)} = \theta(t), \quad y^{(1)} = \frac{d\theta(t)}{dt}. \tag{12.32}$$

Figure 12.5 *Left:* A pendulum of length l driven through resistive air (dotted arcs) by an external sinusoidal torque (semicircle). The strength of the external torque is given by f and that of air resistance by α. *Right:* A double pendulum with neither air resistance nor a driving force. In both cases there is a gravitational torque.



12.11.1 Free Pendulum Oscillations

If we ignore friction and external torques, (12.29) takes the simple form

$$\frac{d^2\theta}{dt^2} = -\omega_0^2 \sin\theta \tag{12.33}$$

If the displacements are small, we can approximate $\sin\theta$ by θ and obtain the linear equation of simple harmonic motion with frequency ω_0:

$$\frac{d^2\theta}{dt^2} \simeq -\omega_0^2\theta \quad \Rightarrow \quad \theta(t) = \theta_0 \sin(\omega_0 t + \phi). \tag{12.34}$$

In Chapter 9, "Differential Equation Applications," we studied how nonlinearities produce anharmonic oscillations, and indeed (12.33) is another good candidate for such studies. As before, we expect solutions of (12.33) for the free realistic pendulum to be periodic, but with a frequency $\omega \simeq \omega_0$ 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 pendulums swing slower (have longer periods) as their angular displacements are made larger.

12.11.2 Solution as Elliptic Integrals

The analytic solution to the realistic pendulum is a textbook problem [L&L,M 76, M&T 03, Schk 94], except that it is hardly a solution and 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 θ_m. Because the initial energy is all potential, we know

that the total energy of the system equals its initial potential energy (Figure 12.5),

$$E = \mathrm{PE}(0) = mgl - mgl\cos\theta_m = 2mgl\sin^2\left(\frac{\theta_m}{2}\right). \tag{12.35}$$

Yet since $E = \mathrm{KE} + \mathrm{PE}$ is a constant, we can write for any value of θ

$$2mgl\sin^2\frac{\theta_m}{2} = \frac{1}{2}I\left(\frac{d\theta}{dt}\right)^2 + 2mgl\sin^2\frac{\theta}{2},$$

$$\Rightarrow \quad \frac{d\theta}{dt} = 2\omega_0\left[\sin^2\frac{\theta_m}{2} - \sin^2\frac{\theta}{2}\right]^{1/2} \quad \Rightarrow \quad \frac{dt}{d\theta} = \frac{T_0/\pi}{\left[\sin^2(\theta_m/2) - \sin^2(\theta/2)\right]^{1/2}},$$

$$\Rightarrow \quad \frac{T}{4} = \frac{T_0}{4\pi}\int_0^{\theta_m}\frac{d\theta}{\left[\sin^2(\theta_m/2) - \sin^2(\theta/2)\right]^{1/2}} = \frac{T_0}{4\pi\sin\theta_m}F\left(\frac{\theta_m}{2}\frac{\theta}{2}\right), \tag{12.36}$$

$$\Rightarrow \quad T \simeq T_0\left[1 + \frac{1}{4}\sin^2\frac{\theta_m}{2} + \frac{9}{64}\sin^4\frac{\theta_m}{2} + \cdots\right], \tag{12.37}$$

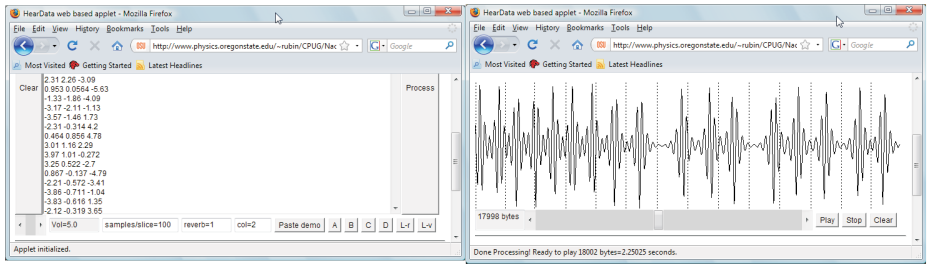
where we have assumed that it takes $T/4$ for the pendulum to travel from $\theta = 0$ to θ_m. The integral in (12.36) is an *elliptic integral of the first kind*. If you think of an elliptic integral as a generalization of a trigonometric function, then this is a closed-form solution; otherwise, it's an integral needing computation. The series expansion of the period (12.37) is obtained by expanding the denominator and integrating it term by term. It tells us, for example, that an amplitude of $80°$ leads to a 10% slowdown of the pendulum relative to the small θ result. In contrast, we will determine the period empirically without the need for any expansions.

12.11.3 Implementation and Test: Free Pendulum

As a preliminary to the solution of the full equation (12.29), modify your **`rk4`** program to solve (12.33) for the free oscillations of a realistic pendulum.

1. Start your pendulum at $\theta = 0$ with $\dot\theta(0) \neq 0$. Gradually increase $\dot\theta(0)$ to increase the importance of nonlinear effects.
2. Test your program for the linear case ($\sin\theta \to \theta$) and verify that
   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 three successive passes of the amplitude through $\theta = 0$. (You need *three* 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 (12.37).
5. Verify that as the initial KE approaches $2mgl$, the motion remains oscillatory but not harmonic (Figure 12.8).
6. At $E = 2\,\mathrm{mgl}$ (the *separatrix*), the motion changes from oscillatory to rotational ("over the top" or "running"). See how close you can get to the separatrix and to its infinite period.
7. Applet ⊙ Use the applet **`HearData`** (Figure 12.6) to convert your different oscillations to sound and hear the difference between harmonic motion (boring) and anharmonic motion

Figure 12.6 The data screen (*left*) and the output screen (*right*) of the applet `HearData` that converts data into sounds. Columns of $[t_i, x(t_i)]$ data are pasted into the data window, processed into the graph in the output window, and then converted to sound data that are played by Java.



containing overtones (interesting).

12.12 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 $x(t)$ (Figure 12.7). As we see from the phase space figures, the solutions form geometric objects that are easy to recognize. (We provide two applets **`Pend1`** and **`Pend2`** to help the reader make the connections between phase space shapes and the corresponding physical motion.)

The position and velocity of a free harmonic oscillator are given by the trigonometric functions

$$x(t) = A\sin(\omega t), \quad v(t) = \frac{dx}{dt} = \omega A\cos(\omega t). \tag{12.38}$$

When substituted into the total energy, we obtain two important results:

$$E = \text{KE} + \text{PE} = \left(\frac{1}{2}m\right)v^2 + \left(\frac{1}{2}\omega^2 m^2\right)x^2 \tag{12.39}$$

$$= \frac{m\omega^2 A^2}{2}\cos^2(\omega t) + \frac{m\omega^2 A^2}{2}\sin^2(\omega t) = \frac{1}{2}m\omega^2 A^2. \tag{12.40}$$

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 same energy start at different places on the same ellipse and transverse the same orbits.

In Figures 12.7–12.10 we show various phase space structures. *Study these figures and their captions* and note the following:

- The orbits of anharmonic oscillations will still be ellipselike but with angular corners that become more distinct with increasing nonlinearity.
- 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).

Figure 12.7 Three potentials and their characteristic behaviors in phase space. The different orbits below the potentials correspond to different energies, as indicated by the limits of maximum displacements within the potentials (dashed lines). Notice the absence of trajectories in regions forbidden by energy conservation. *Left:* A repulsive potential leads to open orbits in phase space characteristic of nonperiodic motion. The phase space trajectories cross at the hyperbolic point in the middle, an unstable equilibrium point. *Middle:* The harmonic oscillator leads to symmetric ellipses in phase space; the closed orbits indicate periodic behavior, and the symmetric trajectories indicate a symmetric potential. *Right:* A nonharmonic oscillator. Notice that the ellipselike trajectories below are neither ellipses nor symmetric with respect to the $v(t)$ axis.

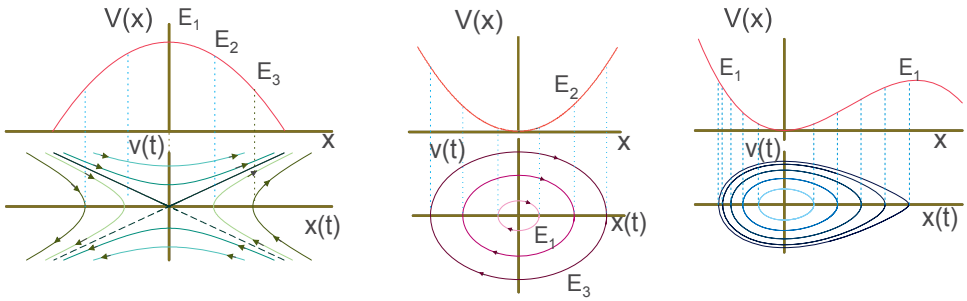
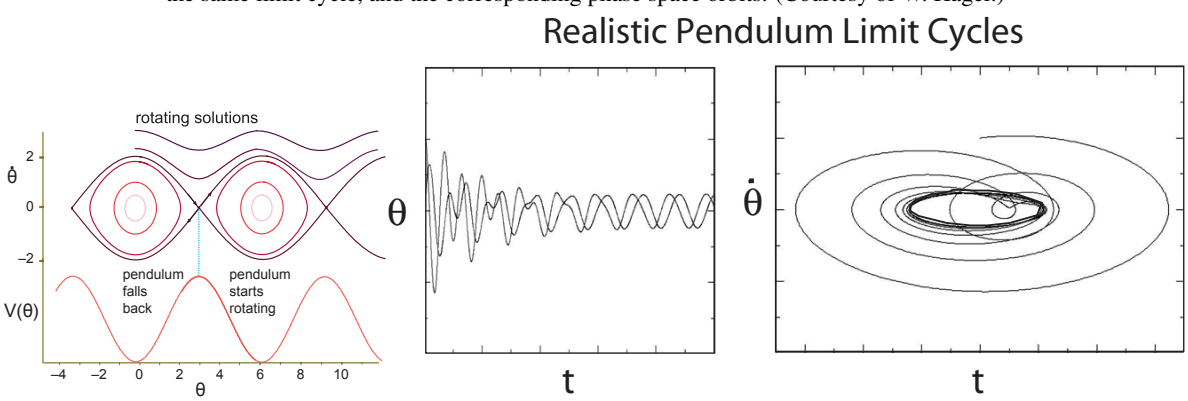
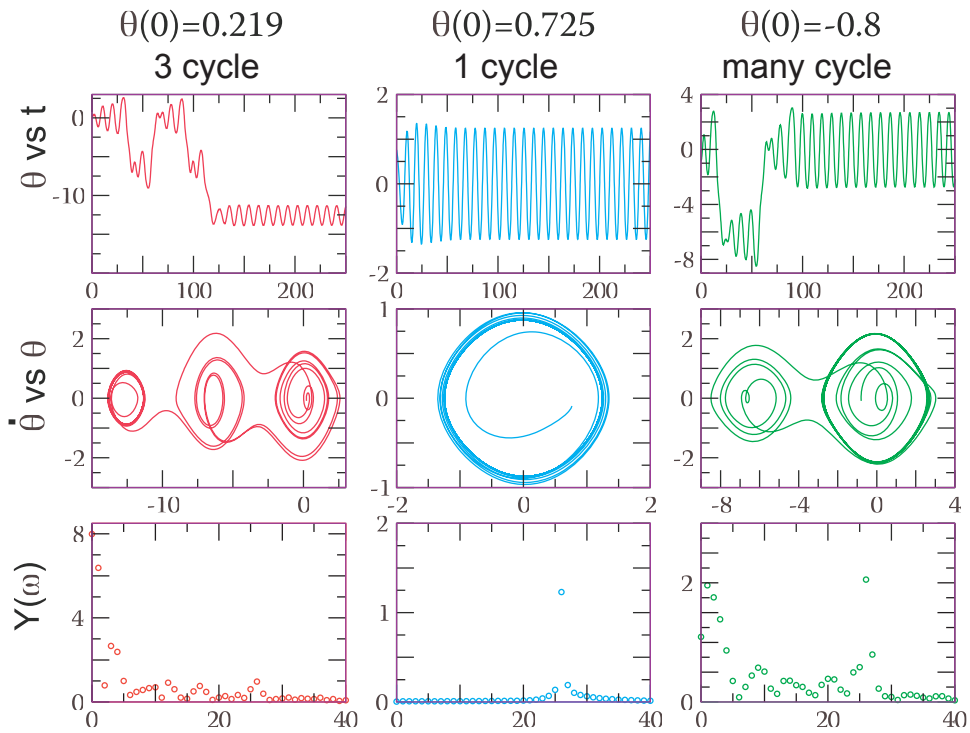


Figure 12.8 *Left:* Phase space trajectories for a plane pendulum (*i.e.* confined to a 2-D plane) including "over the top" or rotating solutions. At the bottom of the figure is shown the corresponding θ dependence of the potential. *Right:* Position *versus* time for two initial conditions of a chaotic pendulum that end up with the same limit cycle, and the corresponding phase space orbits. (Courtesy of W. Hager.)



- Regions of space where the potential is repulsive lead to open trajectories in phase space (Figure12.7).
- As seen in Figure 12.8 left, 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*.
- For certain parameters, a closed *limit cycle* occurs in which the energy pumped in by the external torque exactly balances that lost by friction (Figure 12.8 right).
- Because solutions for different initial conditions are unique, different orbits do not cross. Nonetheless, open orbits join at points of unstable equilibrium (*hyperbolic points* in Figure12.7 left) where an indeterminacy exists.

Figure 12.9 Position *versus* time, phase space plot, and Fourier spectrum for a chaotic pendulum with $\omega_0 = 1$, $\alpha = 0.2$, $f = 0.52$, and $\omega = 0.666$ and three different initial conditions. The leftmost column displays three dominant cycles, the center column only one, while the rightmost column has multiple cycles. (Examples of chaotic behavior can be seen in Figure 12.10.)

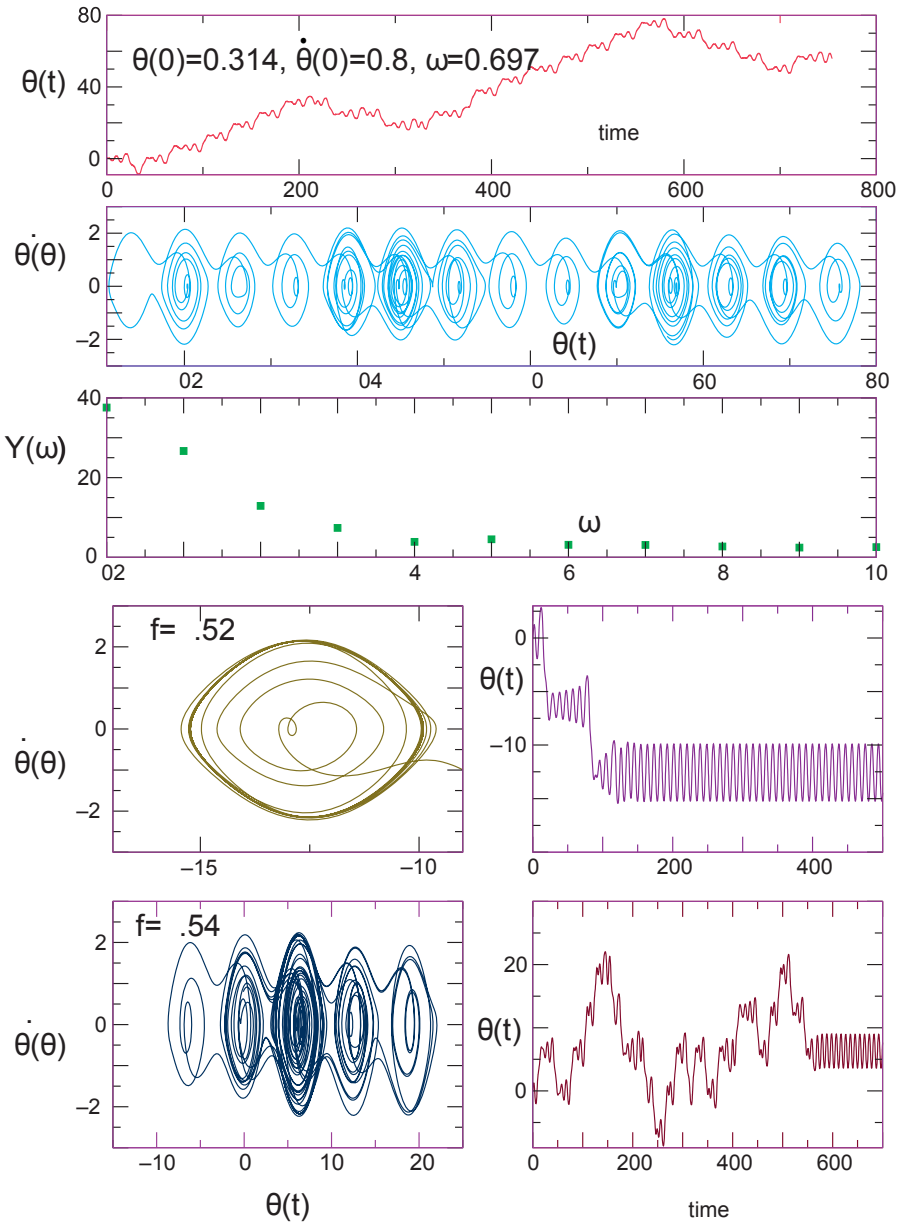


12.12.1 Chaos in Phase Space

It is easy to solve the nonlinear ODE (12.31) on the computer using our usual techniques. However, it is not so easy to understand the solutions because they are so rich in complexity. The solutions are easier to understand in phase space, 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. Geometric structures in phase space is one of them, and determination of the Lyupanov coefficient (discussed in §12.8) is another. Both signal the simplicity lying within the complexity.

What may be surprising is that even though the ellipselike figures in phase space were observed originally for free systems with no friction and no driving torque, similar structures continue to exist for driven systems with friction. The actual trajectories may not remain on a single structure for all times, but they are *attracted* to them and return to them often. 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 (Figure 12.9 middle). If viewed for long times and for many initial conditions, chaotic trajectories (flows) through phase space, while resembling the familiar geometric figures, they may contain dark or diffuse *bands* in places rather than single lines. The continuity of trajectories within bands means that a continuum of solutions is possible and that the system flows continuously among the different trajectories forming the band. The transitions among solutions cause the coordinate space solutions to appear chaotic and are what makes them hypersensitive to the initial conditions (the slightest change in which causes the system to flow to nearby trajectories).

Figure 12.10 Some examples of complicated behaviors of a realistic pendulum. Moving down we see $\theta(t)$ behaviors, a broadband Fourier spectrum, a phase space diagram containing regular patterns with dark bands, and a broad Fourier spectrum. These features are characteristic of chaos. In the bottom two rows we see how the behavior changes abruptly after a slight change in the magnitude of the force and that for $f = 0.54$ there occur the characteristic broad bands of chaos.



Pick out the following phase space structures in your simulations.

Limit cycles: When a 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 (Figure 12.8 right):

$$\langle f \cos \omega t \rangle = \left\langle \alpha \frac{d\theta}{dt} \right\rangle = \left\langle \alpha \frac{d\theta(0)}{dt} \cos \omega t \right\rangle \quad \Rightarrow \quad f = \alpha \frac{d\theta(0)}{dt}. \tag{12.41}$$

This leads to *limit cycles* that appear as closed ellipselike figures, 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 the initial conditions. These are orbits, such as fixed points and limit cycles, into which the system settles or returns to often. 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 with the motion at an earlier time. They are distinguished from predicable attractors by being fractal (Chapter 13, "Fractals & Statistical Growth") chaotic, and highly sensitive to the initial conditions [J&S 98]. 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 (12.41), the driving torque can overpower the natural oscillations, resulting in a steady-state motion at the frequency of the driver. This is called *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 the driving frequency and the natural frequency:

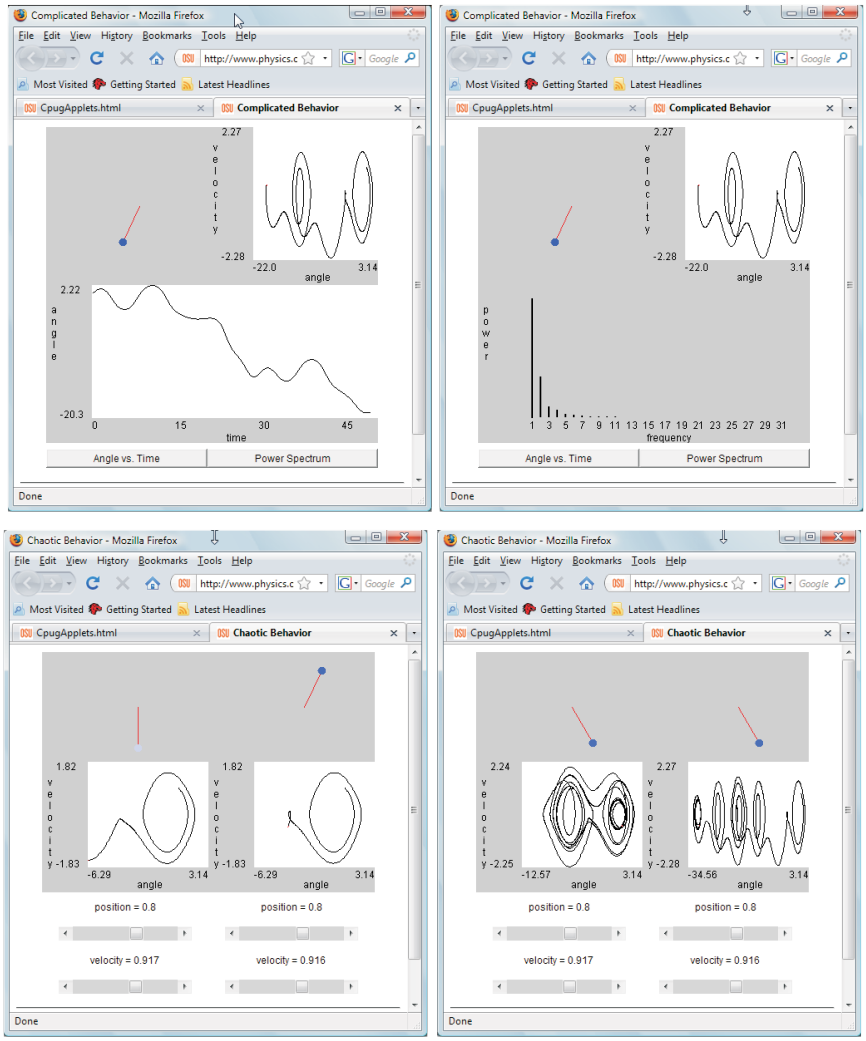
$$\frac{\omega}{\omega_0} = \frac{n}{m}, \quad n, m = \text{integers}. \tag{12.42}$$

Butterfly effects: One of the classic quips about the hypersensitivity of chaotic systems to the 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 counterintuitive because we know that systems with essentially identical initial conditions should behave the same, eventually the systems diverge. The applet **`Pend2`** (Figure12.11 bottom) lets you compare two simulations of nearly identical initial conditions. As seen on the right in Figure 12.11, the initial conditions for both pendulums differ by only 1 part in 917, and so the initial paths in phase space are the same. Nonetheless, at just the time shown here, the pendulums balance in the vertical position, and then one falls before the other, leading to differing oscillations and differing phase space plots from this time onward.

12.12.2 Assessment in Phase Space

The challenge in understanding simulations of the chaotic pendulum (12.31) is that the 4-D parameter space $(\omega_0, \alpha, f, \omega)$ is so immense that only sections of it can be studied systematically. We expect that sweeping through driving frequency ω should show resonances and beating; sweeping through the frictional force α should show underdamping, critical damping,

Figure 12.11 *Top row:* Output from the applet `Pend1` that produces an animation of a chaotic pendulum, along with the corresponding position *versus* time and phase space plots. *Right:* the resulting Fourier spectrum produced by `Pend1`. *Bottom row:* output from the applet `Pend2` producing an animation of two chaotic pendula along with the corresponding phase space plots, and the final output with limit cycles (dark bands).



and overdamping; and sweeping through the driving torque f should show mode locking (for the right values of ω). All these behaviors can be found in the solution of your differential equation, yet they are mixed together in complex ways.

Applet Applet In this assessment you should try to reproduce the behaviors shown in the phase space diagrams in Figure 12.9 and in the applets in Figure 12.11. *Beware*: Because the system is chaotic, you should expect that your results will be sensitive to the exact values of the initial conditions and to the details of your integration routine. We suggest that you experiment; start with the parameter values we used to produce our plots and then observe the effects of making *very small* changes in parameters until you obtain different modes of behavior. Consequently, an inability to reproduce our results for the parameter values does not necessarily imply that something is "wrong."

1. Take your solution to the realistic pendulum and include friction, making α an input parameter. Run it for a variety of initial conditions, including over-the-top ones. Since no

energy is fed to the system, you should see spirals in phase space. Note, if you plot large points at uniform time steps without connecting them, then the spacing between points gives an indication of the speed of the pendulum.

2. Next, verify that with no friction, but with a very small driving torque, you obtain a perturbed ellipse in phase space.
3. Set the driving torque's frequency to be close to the natural frequency ω_0 of the pendulum and search for beats (Figure 12.6 right). 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, scan the frequency ω of the driving torque and search for nonlinear resonance (it looks like beating).
5. **Explore chaos**: Start off with the initial conditions we used in Figure 12.9,

$$(x_0,\, v_0) = (-0.0885,\, 0.8), \quad (-0.0883,\, 0.8), \quad (-0.0888,\, 0.8).$$

   To save time and storage, you may want to use a larger time step for plotting than the one used to solve the differential equations.
6. Indicate which parts of the phase space plots correspond to transients. (The applets may help you with this, especially if you watch the phase space features being built up in time.)
7. Ensure that you have found:
   a. a period-3 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 the phase space appear as bands.
8. Look for the "butterfly effect" (Figure 12.11 bottom). Start two pendulums off with identical positions but with velocities that differ by 1 part in 1000. Notice that the initial motions are essentially identical but that at some later time the motions diverge.

12.13 EXPLORATION: BIFURCATIONS OF CHAOTIC PENDULUMS

We have seen that a chaotic system contains a number of dominant frequencies and that the system tends to "jump" from one to another. This means that the dominant frequencies occur sequentially, in contrast to linear systems where they occur simultaneously. We now want to explore this jumping as a computer experiment. If we sample the instantaneous angular velocity $\dot{\theta} = d\theta/dt$ of our chaotic simulation at various instances in time, we should obtain a series of values for the frequency, with the major Fourier components occurring more often than others.[3] These are the frequencies to which the system is *attracted*. That being the case, if we make a scatterplot of the sampled $\dot{\theta}$s for many times at one particular value of the driving force and then change the magnitude of the driving force slightly and sample the frequencies again, the resulting plot may show distinctive patterns of frequencies. That a bifurcation diagram similar to the one for bug populations results is one of the mysteries of life.

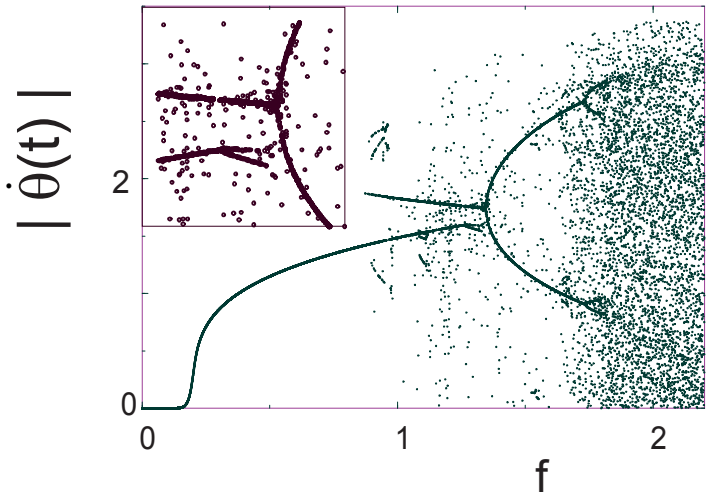
In the scatterplot in Figure 12.12, we sample $\dot{\theta}$ for the motion of a chaotic pendulum with a vibrating pivot point (in contrast to our usual vibrating external torque). This pendulum is similar to our chaotic one (12.29), but with the driving force depending on $\sin\theta$:

$$\frac{d^2\theta}{dt^2} = -\alpha\,\frac{d\theta}{dt} - \left(\omega_0^2 + f\cos\omega t\right)\sin\theta. \tag{12.43}$$

Essentially, the acceleration of the pivot is equivalent to a sinusoidal variation of g or ω_0^2.

[3] We refer to this angular velocity as $\dot{\theta}$ since we have already used ω for the frequency of the driver and ω_0 for the natural frequency.

Figure 12.12 A bifurcation diagram for the damped pendulum with a vibrating pivot (see also the similar diagram for a double pendulum, Figure 12.14). 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 f. Note that the heavy line results from the overlapping of points, not from connecting the points (see enlargement in the inset).



Analytic [L&L,M 76, § 25–30] and numeric [DeJ 92, G,T&C 06] studies of this system exist. To obtain the bifurcation diagram in Figure 12.12:

1. Use the initial conditions $\theta(0) = 1$ and $\dot{\theta}(0) = 1$.
2. Set $\alpha = 0.1$, $\omega_0 = 1$, and $\omega = 2$, and vary $0 \leq f \leq 2.25$.
3. For each value of f, wait 150 periods of the driver before sampling to permit transients to die off. Sample $\dot{\theta}$ for 150 times at the instant the driving force passes through zero.
4. Plot the 150 values of $|\dot{\theta}|$ *versus* f.

12.14 ALTERNATE PROBLEM: THE DOUBLE PENDULUM

For those of you who have already studied a chaotic pendulum, an alternative is to study a double pendulum without any small-angle approximation (Figure 12.5 right and Fig. 12.13, and animation **`DoublePend.mp4`**). A double pendulum has a second pendulum connected to the first, and because each pendulum acts as a driving force for the other, we need not include an external driving torque to produce a chaotic system (there are enough degrees of freedom without it).

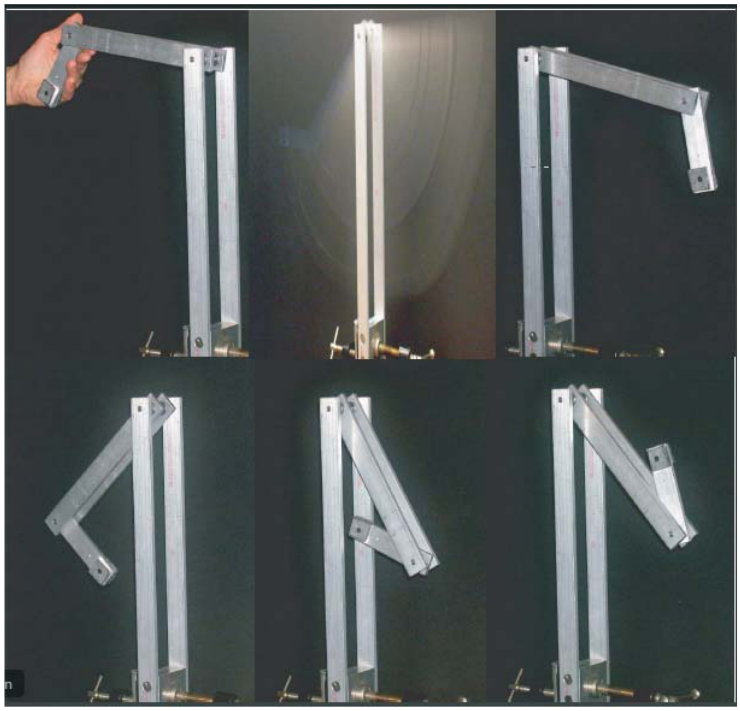
The equations of motions for the double pendulum are derived most directly from the Lagrangian formulation of mechanics. The Lagrangian is fairly simple but has the θ_1 and θ_2 motions innately coupled:

$$L = \mathrm{KE} - \mathrm{PE} = \frac{1}{2}(m_1 + m_2)l_1^2\dot{\theta}_1{}^2 + \frac{1}{2}m_2 l_2^2\dot{\theta}_2{}^2 \tag{12.44}$$

$$+ m_2 l_1 l_2 \dot{\theta}_1\dot{\theta}_2 \cos(\theta_1 - \theta_2) + (m_1 + m_2)g l_1 \cos\theta_1 + m_2 g l_2 \cos\theta_2.$$

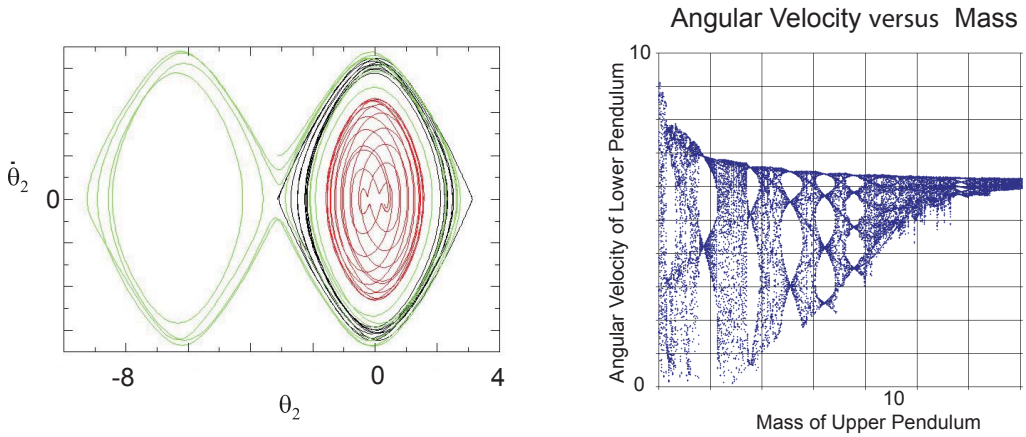
Usually textbooks approximate these equations for small oscillations, which diminish the effects of the coupling. “Slow” and “fast” modes then result for in-phase and antiphase oscillations, respectively, that look much like regular harmonic motions. What’s more interesting is the motion that results without any small-angle restrictions, particularly when the pendulums

Figure 12.13 Photographs of a double pendulum built by a student in the OSU Physics Department. The upper pendulum consists of two separated shafts so that the lower one can rotate completely around. Both pendula can go over their tops. The first two frames show the pendulum released from rest and then moving quickly. The photography with faster shutter speeds stops the motion in various stages. (Photograph, R. Landau.)



Loading, permission pending, DoublePend.mpg

Figure 12.14 *Left:* Phase space trajectories for a double pendulum with $m_1 = 10m_2$ and with two dominant attractors. *Right:* A bifurcation diagram for the double pendulum displaying the instantaneous velocity of the lower pendulum as a function of the mass of the upper pendulum. (Both plots are courtesy of J. Danielson.)



have enough initial energy to go over the top (Figure 12.13). On the left in Figure 12.14 we see several phase space plots for the lower pendulum with $m_1 = 10m_2$. When given enough initial kinetic energy to go over the top, the trajectories are seen to flow between two major attractors as energy is transferred back and forth to the upper pendulum.

On the right in Figure 12.14 is a bifurcation diagram for the double pendulum. This was created by sampling and plotting the instantaneous angular velocity $\dot{\theta}_2$ of the lower pendulum at 70 times as the pendulum passed through its equilibrium position. The mass of the upper pendulum (a convenient parameter) was then changed, and the process repeated. The resulting structure is fractal and indicates bifurcations in the number of dominant frequencies in the motion. A plot of the Fourier or wavelet spectrum as a function of mass is expected to show similar characteristic frequencies.

12.15 ASSESSMENT: FOURIER/WAVELET ANALYSIS OF CHAOS

We have seen that a realistic pendulum experiences a restoring torque, $\tau_g \propto \sin\theta \simeq \theta - \theta^3/3! + \theta^5/5! + \cdots$, that contains nonlinear terms that 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 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 structure has five. You should also notice that when the pendulum goes over the top, its spectrum contains a steady-state (DC) component.

1. Dust off your program for analyzing 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. Wait for the transients to die out before conducting your analysis.

3. Compare your results with the output of the **`Pend1`** applet (Figure 12.11 top).
4. Try to deduce a relation among the Fourier components, the natural frequency ω_0, and the driving frequency ω.
5. 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 by plotting the power spectrum in both linear and semi-logarithmic plots. (The power spectrum often varies over several orders of magnitude.)

Wavelet Exploration: We saw in Chapter 11, "Wavelet Analysis & Data Compression", that a wavelet expansion is more appropriate than a Fourier expansion for a signal containing components that occur for finite periods of time. Because chaotic oscillations are just such signals, repeat the Fourier analysis of this section using wavelets instead of sines and cosines. Can you discern the temporal sequence of the various components?

12.16 EXPLORATION: ALTERNATE PHASE SPACE PLOTS

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 [Abar 93]. Here τ is a *lag time* and 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,

$$v(t) = \frac{dx(t)}{dt} \simeq \frac{x(t+\tau) - x(t)}{\tau}. \tag{12.45}$$

We see that plotting $x(t+\tau)$ *versus* $x(t)$ is equivalent to plotting $v(t)$ *versus* $x(t)$.

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 τ. Compare your results to the conventional phase space plots you obtained previously. ▌

12.17 FURTHER EXPLORATIONS

1. The nonlinear behavior in once-common objects such as vacuum tubes and metronomes is described by the **van der Pool equation**,
   indexVan der Pool equation

$$\frac{d^2x}{dt^2} + \mu(x^2 - x_0^2)\,\frac{dx}{dt} + \omega_0^2 x = 0. \tag{12.46}$$

   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 (12.46) as describing an oscillator with x-dependent damping (the μ term). If $x > x_0$, friction slows the system down; if $x < x_0$, friction speeds the system up. Orbits internal to the limit cycle spiral out until they reach the limit cycle; orbits external to it spiral in.

2. **Duffing oscillator:** Another damped, driven nonlinear oscillator is

$$\frac{d^2\theta}{dt^2} - \frac{1}{2}\theta(1-\theta^2) = -\alpha\frac{d\theta}{dt} + f\cos\omega t. \tag{12.47}$$

While similar to the chaotic pendulum, it is easier to find multiple attractors for this oscillator [M&L 85].

3. **Lorenz attractor:** In 1962 Lorenz [Tab 89] was looking for a simple model for weather prediction and simplified the heat transport equations to

$$\frac{dx}{dt} = 10(y-x), \quad \frac{dy}{dt} = -xz + 28x - y, \quad \frac{dz}{dt} = xy - \frac{8}{3}z. \tag{12.48}$$

The solution of these simultaneous first-order 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 3-D computer fly:** Make $x+y$, $x+z$, and $y+z$ plots of the equations

$$x = \sin ay - z\cos bx, \quad y = z\sin cx - \cos\, dy, \quad z = e\sin x. \tag{12.49}$$

Here the parameter e controls the degree of apparent randomness.

5. **Hénon–Heiles potential:** The potential and Hamiltonian

$$V(x,y) = \frac{1}{2}x^2 + \frac{1}{2}y^2 + x^2y - \frac{1}{3}y^3, \quad H = \frac{1}{2}p_x^2 + \frac{1}{2}p_y^2 + V(x,y), \tag{12.50}$$

are used to describe three interacting astronomical objects. The potential binds the objects near the origin but releases them if they move far out. The equations of motion follow from the Hamiltonian equations:

$$\frac{dp_x}{dt} = -x - 2xy, \quad \frac{dp_y}{dt} = -y - x^2 + y^2, \quad \frac{dx}{dt} = p_x, \quad \frac{dy}{dt} = p_y.$$

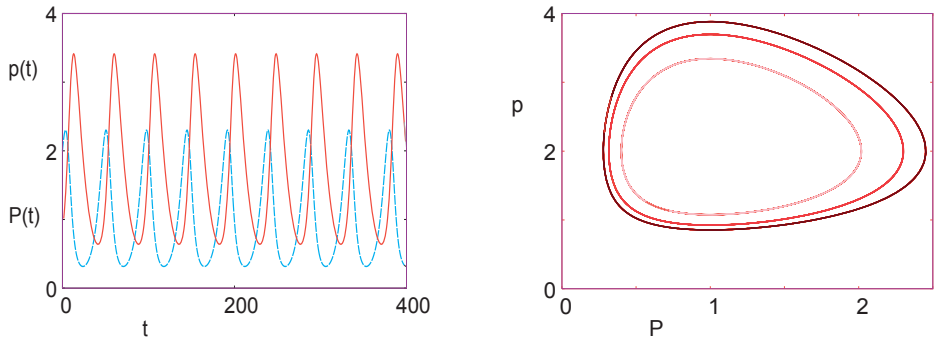
a. Numerically solve for the position $[x(t), y(t)]$ for a particle in the Hénon–Heiles potential.
b. Plot $[x(t), y(t)]$ for a number of initial conditions. Check that the initial condition $E < \frac{1}{6}$ leads to a bounded orbit.
c. Produce a Poincaré section in the (y, p_y) plane by plotting (y, p_y) each time an orbit passes through $x = 0$.

12.18 UNIT III. COUPLED PREDATOR–PREY MODELS ⊙

In Unit I we saw complicated behavior arising from a model of bug population dynamics in which we imposed a maximum population. We described that system with a discrete logistic map. In Unit II we saw complex behaviors arising from differential equations and learned how to use phase space plots to understand them. In this unit we study the differential equation model describing predator–prey population dynamics proposed by the American physical chemist Lotka [Lot 25] and the Italian mathematician Volterra [Volt 26]. Differential equations are easy to solve numerically and should be a good approximation if populations are large. However, there are equivalent discrete map versions of the model as well. Though simple, versions of these equations are used to model biological systems and neural networks.

Problem: Is it possible to use a small number of predators to control a population of pests so that the number of pests remains approximately constant? Include in your considerations the interaction between the populations as well as the competition for food and predation time.

Figure 12.15 *Left:* The time dependences of the populations of prey *p(t)* (solid curve) and of predator *P(t)* (dashed curve) from the Lotka–Volterra model. *Right:* A phase space plot of Prey population *p* as a function of predator population *P*. The different orbits correspond to different initial conditions.



12.19 LOTKA–VOLTERRA MODEL

We extend the logistic map to the Lotka–Volterra Model (LVM) to describe two populations coexisting in the same geographical region. Let

$$p(t) = \text{prey density}, \quad P(t) = \text{predator density}. \tag{12.51}$$

In the absence of interactions between the species, we assume that the prey population p breeds at a per-capita rate of a, which would lead to exponential growth:

$$\frac{dp}{dt} = ap, \quad \Rightarrow \quad p(t) = p(0)e^{at}. \tag{12.52}$$

Yet exponential growth does not occur because the predators P eat more prey as their numbers increase. The interaction rate between predator and prey requires both to be present, with the simplest assumption being that it is proportional to their joint probability:

$$\text{Interaction rate} = bpP.$$

This leads to a prey growth rate including both predation and breeding:

$$\frac{dp}{dt} = a\,p - b\,p\,P, \quad \text{(LVM-I for prey)}. \tag{12.53}$$

If left to themselves, predators P will also breed and increase their population. Yet predators need animals to eat, and if there are no other populations to prey upon, they will eat each other (or their young) at a per-capita mortality rate m:

$$\left.\frac{dP}{dt}\right|_{\text{competition}} = -mP, \quad \Rightarrow \quad P(t) = P(0)e^{-mt}. \tag{12.54}$$

However, once there are prey to interact with (read "eat") at the rate bpP, the predator population will grow at the rate

$$\frac{dP}{dt} = \epsilon\,b\,p\,P - m\,P \quad \text{(LVM-I for predators)}, \tag{12.55}$$

where ϵ is a constant that measures the efficiency with which predators convert prey interactions into food.

Equations (12.53) and (12.55) are two simultaneous ODEs and are our first model. We solve them with the **`rk4`** algorithm of Chapter 9, "Differential Equation Applications", after

placing them in the standard dynamic form,

$$
\begin{aligned}
& d\mathbf{y}/dt = \mathbf{f}(\mathbf{y}, t), \\
y_0 &= p, & f_0 &= a\, y_0 - b\, y_0\, y_1, \\
y_1 &= P, & f_1 &= \epsilon\, b\, y_0\, y_1 - m\, y_1.
\end{aligned}
\tag{12.56}
$$

A sample code to solve these equations is **`PredatorPrey.py`**.

Listing 12.4 **PredatorPrey** computes the population dynamics for a group of predators and prey.

```
# PredatorPrey.py: Variations on the Lotka - Volterra model

from visual.graph import *

Tmin = 0.0
Tmax = 500.0                                                    # endpoints
y = zeros( (2), float)
Ntimes = 1000
y[0] = 2.0
y[1] = 1.3                                                      # initialize
h = (Tmax - Tmin)/Ntimes
t = Tmin

def f( t, y, F):   # FUNCTION of your choice here
        F[0] = 0.2*y[0]*(1 - (y[0]/(20.0) )) - 0.1*y[0]*y[1] # RHS 1st eq
        F[1]  = - 0.1*y[1] + 0.1*y[0]*y[1];                  # RHS 2nd eq

def rk4(t, y, h, Neqs):  # rk4 method, *DO NOT MODIFY*, instead, modify f
    F = zeros((Neqs), float)
    ydumb = zeros((Neqs), float)
    k1    = zeros((Neqs), float)
    k2    = zeros((Neqs), float)
    k3    = zeros((Neqs), float)
    k4    = zeros((Neqs), float)
    f(t, y, F)
    for i in range(0, Neqs):
        k1[i] = h*F[i]
        ydumb[i] = y[i] + k1[i]/2.
    f(t + h/2., ydumb, F)
    for i in range(0, Neqs):
        k2[i] = h*F[i]
        ydumb[i] = y[i] + k2[i]/2.
    f(t + h/2., ydumb, F)
    for i in range(0, Neqs):
        k3[i] =  h*F[i]
        ydumb[i] = y[i] + k3[i]
    f(t + h, ydumb, F)
    for i in range(0, Neqs):
        k4[i] = h*F[i]
        y[i] = y[i] + (k1[i] + 2.*(k2[i] + k3[i]) + k4[i])/6.

# == == == == == == == = == == == == = rk4 ends == == == == == == ==

graph1 = gdisplay(x= 0,y= 0, width = 500, height = 400,
                  title = 'Prey p(green) and predator P(yellow) vs time',
           xtitle = 't', ytitle = 'P, p',xmin=0,xmax=500,ymin=0,ymax=3.5)
funct1 = gcurve(color = color.yellow)
funct2 = gcurve(color = color.green)
graph2 = gdisplay(x= 0,y= 400, width = 500, height = 400,
                  title = 'Predator P vs prey p',
               xtitle = 'P', ytitle = 'p',xmin=0,xmax=2.5,ymin=0,ymax=3.5)
funct3 = gcurve(color = color.red)

for t in arange(Tmin, Tmax + 1, h):
    funct1.plot(pos = (t, y[0]) )
    funct2.plot(pos = (t, y[1]) )
    funct3.plot(pos = (y[0], y[1]) )
    rate(60)
    rk4(t, y, h, 2)
```

12.19.1 Lotka–Volterra Assessment

Results from the code **`PredatorPrey.py`** are shown in Figure 12.15. On the left we see that the two populations oscillate out of phase with each other in time; when there are many prey, the

predator population eats them and grows; yet then the predators face a decreased food supply and so their population decreases; that in turn permits the prey population to grow, and so forth. On the right in Figure 12.15 we plot a phase space plot (phase space plots are discussed in Unit II) of $P(t)$ *versus* $p(t)$. A closed orbit here indicates a limit cycle that repeats indefinitely. Although increasing the initial number of predators does decrease the maximum number of pests, it is not a satisfactory solution to our **problem**, as the large variation in the number of pests cannot be called control.

1. Explain the correlation between extremums in prey and predator populations?
2. Since predators eat prey, one might expect the existence of a large number of predators to lead to the eating of a large number of prey. Explain why the maxima in predator population and the minima in prey population do not occur at the same time.
3. Why do the extreme values of the population just repeat with no change in their values?
4. Explain the meaning of the spirals in the predator-prey phase space diagram.
5. What are the phase space orbits closed?
6. What different initial conditions would lead to different phase-space orbits?
7. Discuss the symmetry and lack of symmetry in the phase-space orbits.

12.19.2 LVM with Prey Limit

The initial assumption in the LVM that prey grow without limit in the absence of predators is clearly unrealistic. As with the logistic map, we include a limit on prey numbers that accounts for depletion of the food supply as the prey population grows. Accordingly, we modify the constant growth rate $a \to a(1 - p/K)$ so that growth vanishes when the population reaches a limit K, the *carrying capacity*,

$$\frac{dp}{dt} = a\,p\left(1 - \frac{p}{K}\right) - b\,p\,P, \quad \frac{dP}{dt} = \epsilon\,b\,p\,P - m\,P \quad \text{(LVM-II)}. \tag{12.57}$$

The behavior of this model with prey limitations is shown in Figure 12.16. We see that both populations exhibit damped oscillations as they approach their equilibrium values. In addition, and as hoped for, the equilibrium populations are independent of the initial conditions. Note how the phase space plot spirals inward to a single close limit cycle, on which it remains, with little variation in prey number. This is control, and we may use it to predict the expected pest population.

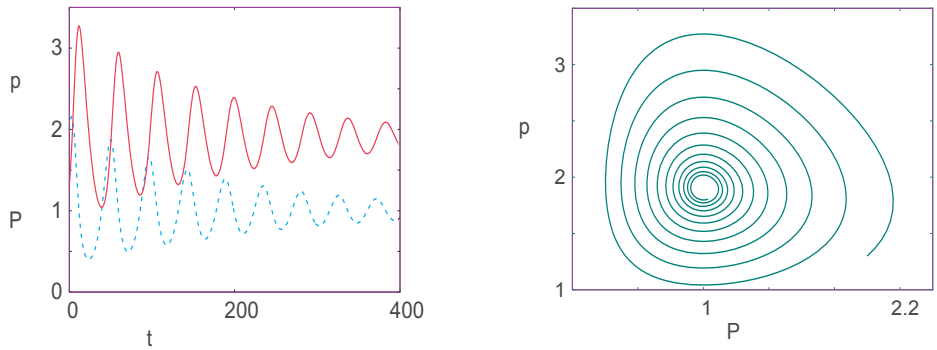
12.19.3 LVM with Predation Efficiency

An additional unrealistic assumption in the original LVM is that the predators immediately eat all the prey with which they interact. As anyone who has watched a cat hunt a mouse knows, predators spend time finding prey and also chasing, killing, eating, and digesting it (all together called *handling*). This extra time decreases the rate bpP at which prey are eliminated. We define the *functional response* p_a as the probability of one predator finding one prey. If a single predator spends time t_{search} searching for prey, then

$$p_a = b\,t_{\text{search}}\,p \;\Rightarrow\; t_{\text{search}} = \frac{p_a}{bp}. \tag{12.58}$$

If we call t_h the time a predator spends handling a single prey, then the effective time a predator spends handling a prey is $p_a\,t_h$. Such being the case, the total time T that a predator spends

Figure 12.16 *Left:* The Lotka–Volterra model of prey population *p(t)* (solid curve) and predator population *P(t)* (dashed curve) versus time when a prey population limit is included. *Right:* Prey population *p* as a function of predator population *P*.



finding and handling a single prey is

$$T = t_{\text{search}} + t_{\text{handling}} = \frac{p_a}{bp} + p_a t_h \quad \Rightarrow \quad \frac{p_a}{T} = \frac{bp}{1 + bpt_h},$$

where p_a/T is the effective *rate* of eating prey. We see that as the number of prey $p \to \infty$, the efficiency in eating them $\to 1$. We include this efficiency in (12.57) by modifying the rate b at which a predator eliminates prey to $b/(1 + bpt_h)$:

$$\frac{dp}{dt} = ap\left(1 - \frac{p}{K}\right) - \frac{bpP}{1 + bpt_h}, \quad \text{(LVM-III)}. \tag{12.59}$$

To be more realistic about the predator growth, we also place a limit on the predator carrying capacity but make it proportional to the number of prey:

$$\frac{dP}{dt} = mP\left(1 - \frac{P}{kp}\right), \quad \text{(LVM-III)}. \tag{12.60}$$

Solutions for the extended model (12.59) and (12.60) are shown in Figure 12.17. Observe the existence of three dynamic regimes as a function of b:

- small b: no oscillations, no overdamping,
- medium b: damped oscillations that converge to a stable equilibrium,
- large b: limit cycle.

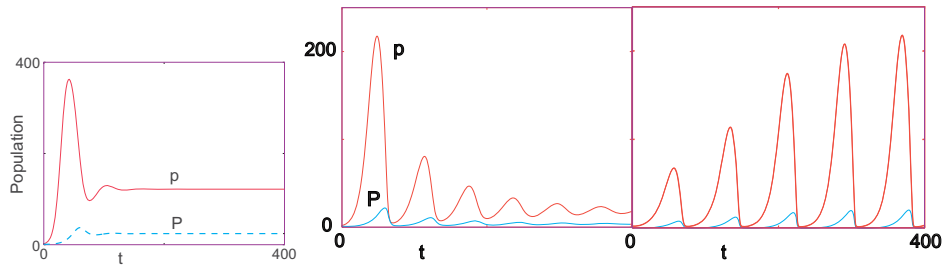
The transition from equilibrium to a limit cycle is called a *phase transition*.

We finally have a satisfactory solution to our **problem**. Although the prey population is not eliminated, it can be kept from getting too large and from fluctuating widely. Nonetheless, changes in the parameters can lead to large fluctuations or to nearly vanishing predators.

12.19.4 LVM Implementation and Assessment

1. Write a program to solve (12.59) and (12.60) using the `rk4` algorithm and the following parameter values.
2. For each of the three models, construct
   a. a time series for prey and predator populations,
   b. phase space plots of predator *versus* prey populations.

Figure 12.17 Lotka–Volterra model with predation efficiency and prey limitations. From left to right: overdamping, $b = 0.01$; damped oscillations, $b = 0.1$, and limit cycle, $b = 0.3$.



Model	a	b	ϵ	m	K	k
LVM-I	0.2	0.1	1	0.1	0	—
LVM-II	0.2	0.1	1	0.1	20	—
LVM-III	0.2	0.1	—	0.1	500	0.2

3. **LVM-I**: Compute the equilibrium values for the prey and predator populations. Do you think that a model in which the cycle amplitude depends on the initial conditions can be realistic? Explain.
4. **LVM-II**: Calculate numerical values for the equilibrium values of the prey and predator populations. Make a series of runs for different values of prey carrying capacity K. Can you deduce how the equilibrium populations vary with prey carrying capacity?
5. Make a series of runs for different initial conditions for predator and prey populations. Do the cycle amplitudes depend on the initial conditions?
6. **LVM-III**: Make a series of runs for different values of b and reproduce the three regimes present in Figure 12.17.
7. Calculate the critical value for b corresponding to a phase transition between the stable equilibrium and the limit cycle.

12.19.5 Two Predators, One Prey (Exploration)

1. Another version of the LVM includes the possibility that two populations of predators P_1 and P_2 may "share" the same prey population p. Investigate the behavior of a system in which the prey population grows logistically in the absence of predators:

$$\frac{dp}{dt} = ap\left(1 - \frac{p}{K}\right) - (b_1 P_1 + b_2 P_2)\, p, \tag{12.61}$$

$$\frac{dP}{dt} = \epsilon_1 b_1 p P_1 - m_1 P_1, \qquad \frac{dP_2}{dt} = \epsilon_2 b_2 p P_2 - m_2 P_2. \tag{12.62}$$

   a. Use the following values for the model parameters and initial conditions: $a = 0.2$, $K = 1.7$, $b_1 = 0.1$, $b_2 = 0.2$, $m_1 = m_2 = 0.1$, $\epsilon_1 = 1.0$, $\epsilon_2 = 2.0$, $p(0) = P_2(0) = 1.7$, and $P_1(0) = 1.0$.
   b. Determine the time dependences for each population.
   c. Vary the characteristics of the second predator and calculate the equilibrium population for the three components.
   d. What is your answer to the question, "Can two predators that share the same prey coexist?"
2. The nonlinear nature of the Lotka–Volterra model can lead to chaos and fractal behavior. Search for chaos by varying the growth rates.

Chapter Thirteen

Fractals & Statistical Growth

It is common to notice regular and eye-pleasing natural objects, such as plants and sea shells, that do not have well-defined geometric shapes. When analyzed mathematically, some of these objects have a dimension that is a fractional number, rather than an integer, and so are called "fractals". In this chapter we implement simple, statistical models that generate 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. Detailed applications of fractals can be found in the literature [Mand 82, Arm 91, E&P 88, Sand 94, PhT 88].

VIDEO LECTURES, APPLETS AND ANIMATIONS FOR THIS CHAPTER

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Fractals I	pdf	13.1-.2	Fractals II	pdf	13.3-

Applets & Animations (see too Python Codes) Applet

Name	*Sections*	*Name*	*Sections*	*Name*	*Sections*
Sierpinski	13.2	Fern	13.3		
Tree	13.3	Film	13.4	Column	13.6
DLA	13.7	Ballistic Movie	13.4		

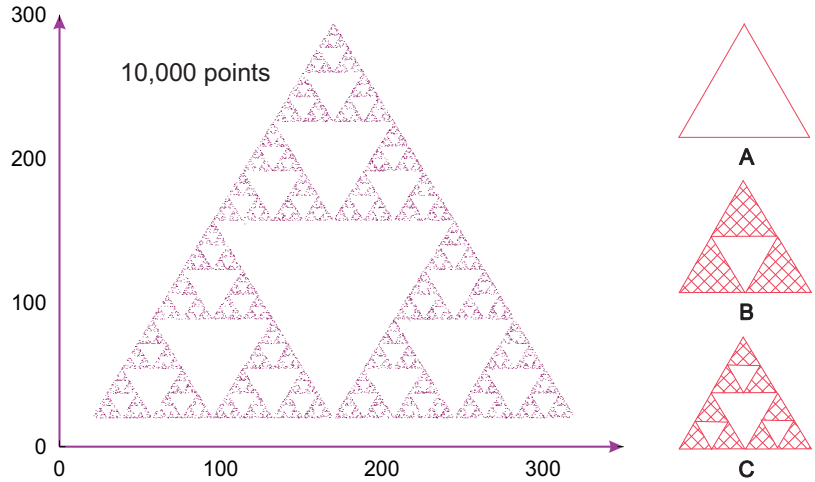
13.1 FRACTIONAL DIMENSION (MATH)

Benoit Mandelbrot, who first studied fractional-dimension figures with supercomputers at IBM Research, gave them the name *fractals* [Mand 82]. Some geometric objects, such as 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 $\langle r^n \rangle$ is a measure of the size, and there is an infinite number of moments. Likewise, when we deal with complicated objects, there are different definitions of dimension and each may give a somewhat different value. In addition, the fractal dimension is often defined by using a measuring box whose size approaches zero, which is not practical for realistic applications.

Our first definition of the fractional dimension d_f (or *Hausdorff–Besicovitch dimension*) is based on our knowledge that a line has dimension 1, a triangle has dimension 2, and a cube has dimension 3. It seems perfectly reasonable to ask if there is some mathematical formula that agrees with our experience with 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,

Figure 13.1 *Left:* A statistical fractal Sierpiński gasket containing 10,000 points. Note the self-similarity at different scales. *Right:* A geometric Sierpiński gasket constructed by successively connecting the midpoints of the sides of each equilateral triangle. The first three steps in the process are labeled as A, B, C.



as do equilateral triangles and squares, and that have uniform density. We postulate that the dimension of an object is determined by the dependence of its total mass upon its length:

$$M(L) \propto L^{d_f}, \tag{13.1}$$

where the power d_f is the *fractal dimension*. As you may verify, this rule works with the 1-D, 2-D, and 3-D regular figures in our experience, so it is a reasonable definition. When we apply (13.1) 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 as mass/volume for a solid object. That being the case, for a planar object we hypothesize that

$$\rho = \frac{M(L)}{\text{area}} \propto \frac{L^{d_f}}{L^2} \propto L^{d_f - 2}. \tag{13.2}$$

13.2 THE SIERPIŃSKI GASKET (PROBLEM 1)

Applet To generate our first fractal (Figure 13.1), we play a game of chance in which we place dots at points picked randomly within a triangle [Burde 74]. Here are the rules (which you should try out in the margins now).

1. Draw an equilateral triangle with vertices and coordinates:

$$\text{vertex } 1\colon (a_1, b_1); \quad \text{vertex } 2\colon (a_2, b_2); \quad \text{vertex } 3\colon (a_3, b_3).$$

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.

Mathematically, the coordinates of successive points are given by the formulas

$$(x_{k+1}, y_{k+1}) = \frac{(x_k, y_k) + (a_n, b_n)}{2}, \quad n = \text{integer}\,(1 + 3r_i), \tag{13.3}$$

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 those on the left in Figure 13.1.

13.2.1 Sierpiński 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.)

13.2.2 Assessing Fractal Dimension

The topology in Figure 13.1 was first analyzed by the Polish mathematician Sierpiński. Observe that there is the same structure in a small region as there is in the entire figure. In other words, if the figure had infinite resolution, any part of the figure could be scaled up in size and would be similar to the whole. This property is called *self-similarity*.

We construct a regular form of the Sierpiński gasket by removing an inverted equilateral triangle from the center of all filled equilateral triangles to create the next figure (Figure 13.1 right). 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 (13.2) to determine its fractal dimension. Assume that each triangle has mass m and assign unit density to the single triangle:

$$\rho(L = r) \propto \frac{M}{r^2} = \frac{m}{r^2} \stackrel{\text{def}}{=} \rho_0 \quad \text{(Figure 13.1A)}$$

Next, for the equilateral triangle with side $L = 2$, the density

$$\rho(L = 2r) \propto \frac{(M = 3m)}{(2r)^2} = \frac{3}{4} m r^2 = \frac{3}{4}\rho_0 \quad \text{(Figure 13.1B)}$$

We see that the extra white space in Figure 13.1B leads to a density that is $\frac{3}{4}$ that of the previous stage. For the structure in Figure 13.1C, we obtain

$$\rho(L = 4r) \propto \frac{(M = 9m)}{(4r)^2} = \frac{9}{16}\frac{m}{r^2} = \left(\frac{3}{4}\right)^2 \rho_0. \quad \text{(Figure 13.1C)}$$

We see that as we continue the construction process, the density of each new structure is $\frac{3}{4}$ that of the previous one. Interesting. Yet in (13.2) we derived that

$$\rho \propto CL^{d_f - 2}. \tag{13.4}$$

Equation (13.4) implies that a plot of the logarithm of the density ρ *versus* the logarithm of the length L for successive structures yields a straight line of slope

$$d_f - 2 = \frac{\Delta \log \rho}{\Delta \log L}. \tag{13.5}$$

As applied to our problem,

$$d_f = 2 + \frac{\Delta \log \rho(L)}{\Delta \log L} = 2 + \frac{\log 1 - \log \frac{3}{4}}{log 1 - \log 2} \simeq 1.58496. \tag{13.6}$$

As is evident in Figure 13.1, as the gasket grows larger (and consequently more massive), it contains more open space. So even though its mass approaches infinity as $L \to \infty$, its density approaches zero! And since a 2-D figure like a solid triangle has a constant density as its length increases, a 2-D figure has 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 2-D object but more than a 1-D object does; it is a fractal with dimension ≤ 1.6.

13.3 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 graceful as a fern (Figure 13.2 left) 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 shapes of ferns.

13.3.1 Self-Affine Connection (Theory)

In (13.3), 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

$$(x', y') = s(x, y) = (sx, sy) \quad \text{(scaling)}. \tag{13.7}$$

If the scale factor $s > 0$, an amplification occurs, whereas if $s < 0$, a reduction occurs. In our definition (13.3) of the Sierpiński gasket, we also added in a constant a_n. This is a *translation operation*, which has the general form

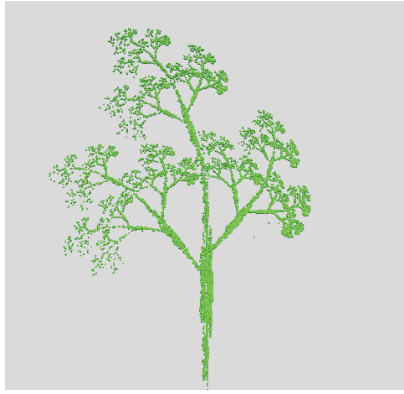
$$(x', y') = (x, y) + (a_x, a_y) \quad \text{(translation)}. \tag{13.8}$$

Another operation, not used in the Sierpiński gasket, is a *rotation* by angle θ:

$$x' = x\cos\theta - y\sin\theta, \quad y' = x\,\sin\theta + y\cos\theta \quad \text{(rotation)}. \tag{13.9}$$

The entire set of transformations, scalings, rotations, and translations defines 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 by a single θ (in that case, we can have contractions and reflections). What is important is that the object created with these rules turns out to be self-similar; each step leads to new parts of the object that bear the same relation to the ancestor parts as the ancestors did to theirs. This is what makes the object look similar at all scales.

Figure 13.2 *Left:* A fractal fern generated by 30,000 iterations of the algorithm (13.10). Enlarging this fern shows that each frond with a similar structure. *Right:* A fractal tree created with the simple algorithm (13.13).



13.3.2 Barnsley's Fern Implementation

Applet

We obtain a Barnsley's fern [Barns 93] by extending the dots game to one in which new points are selected using an affine connection with some elements of chance mixed in:

$$(x,y)_{n+1} = \begin{cases} (0.5, 0.27y_n), & \text{with 2\% probability,} \\ (-0.139x_n + 0.263y_n + 0.57 \\ \quad 0.246x_n + 0.224y_n - 0.036), & \text{with 15\% probability,} \\ (0.17x_n - 0.215y_n + 0.408 \\ \quad 0.222x_n + 0.176y_n + 0.0893), & \text{with 13\% probability,} \\ (0.781x_n + 0.034y_n + 0.1075 \\ \quad -0.032x_n + 0.739y_n + 0.27), & \text{with 70\% probability.} \end{cases} \tag{13.10}$$

To select a transformation with probability $\mathcal{P}$, we select a uniform random number $0 \le r \le 1$ and perform the transformation if r is in a range proportional to $\mathcal{P}$:

$$\mathcal{P} = \begin{cases} 2\%, & r < 0.02, \\ 15\%, & 0.02 \le r \le 0.17, \\ 13\%, & 0.17 < r \le 0.3, \\ 70\%, & 0.3 < r < 1. \end{cases} \tag{13.11}$$

The rules (13.10) and (13.11) can be combined into one:

$$(x,y)_{n+1} = \begin{cases} (0.5, 0.27y_n), & r < 0.02, \\ (-0.139x_n + 0.263y_n + 0.57 \\ \quad 0.246x_n + 0.224y_n - 0.036), & 0.02 \le r \le 0.17, \\ (0.17x_n - 0.215y_n + 0.408 \\ \quad 0.222x_n + 0.176y_n + 0.0893), & 0.17 < r \le 0.3, \\ (0.781x_n + 0.034y_n + 0.1075, \\ \quad -0.032x_n + 0.739y_n + 0.27), & 0.3 < r < 1. \end{cases} \tag{13.12}$$

Although (13.10) makes the basic idea clearer, (13.12) is easier to program.

The starting point in Barnsley's fern (Figure 13.2) 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 the stems and the fronds are. Nevertheless, the stem can be viewed as a compressed copy of a frond, and the fractal obtained with (13.10) is still *self-affine*, yet with a dimension that varies in different parts of the figure.

Listing 13.1 **Fern3D.py** simulates the growth of ferns in 3-D.

```
# Fern3D.py:  Fern in 3D, see Barnsley, "Fractals Everywhere"

from visual import *
import random
imax = 20000                                                    # points to draw
x = 0.5                                                        # initial x coord
y = 0.0                                                        # initial y coord
z = -0.2
xn = 0.0
yn = 0.0
graph1 = display(width=500, height=500, forward=(-3,0,-1),\
       title='3D Fractal Fern (rotate via right mouse button)', range=10)
graph1.show_rendertime = True
# Using points: cycle=27 ms, render=6 ms
# Using spheres: cycle=750 ms, render=30 ms
pts = points(color=color.green, size=0.01)
for i in range(1,imax):
    r = random.random();                                         # random number
    if ( r <= 0.1):                                            # 10% probability
        xn = 0.0
        yn = 0.18*y
        zn = 0.0
    elif ( r > 0.1 and r <= 0.7):                              # 60% probability
        xn =  0.85 * x
        yn =  0.85 * y + 0.1 * z + 1.6
        zn = -0.1  * y + 0.85 * z
            # print xn,yn,zn
    elif ( r > 0.7 and r <= 0.85):                            # 15 % probability
        xn =  0.2 * x - 0.2 * y
        yn =  0.2 * x + 0.2 * y + 0.8
        zn=   0.3 * z
    else:
         xn = -0.2 * x +0.2 * y                                # 15% probability
         yn =  0.2 * x +0.2 * y + 0.8
         zn =  0.3 * z
    x = xn
    y = yn
    z = zn
    xc = 4.0*x                                             # linear TF for plot
    yc = 2.0*y-7
    zc = z
    pts.append(pos=(xc,yc,zc))
```

13.3.3 Self-Affinity in Trees Implementation

Loading (permission may be pending?) movie FractalGrowth.mpg

Now that you know how to grow ferns, look around and notice the regularity in trees (such as in Figure 13.2 right). 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:

$$(x_{n+1}, y_{n+1}) = \begin{cases} (0.05x_n, 0.6y_n), & 10\% \text{ probability}, \\ (0.05x_n, -0.5y_n + 1.0), & 10\% \text{ probability}, \\ (0.46x_n - 0.15y_n, 0.39x_n + 0.38y_n + 0.6), & 20\% \text{ probability}, \\ (0.47x_n - 0.15y_n, 0.17x_n + 0.42y_n + 1.1), & 20\% \text{ probability}, \\ (0.43x_n + 0.28y_n, -0.25x_n + 0.45y_n + 1.0), & 20\% \text{ probability}, \\ (0.42x_n + 0.26y_n, -0.35x_n + 0.31y_n + 0.7), & 20\% \text{ probability}. \end{cases} \tag{13.13}$$

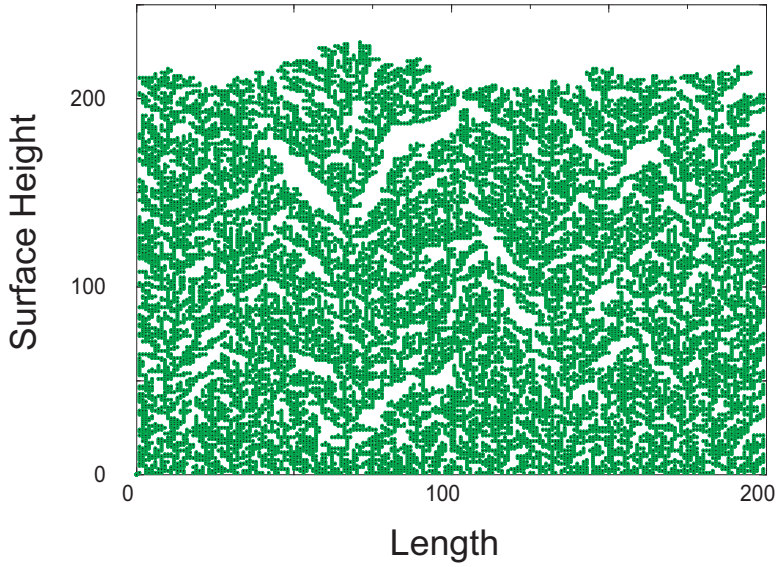
13.4 BALLISTIC DEPOSITION (PROBLEM 3)

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.

13.4.1 Random Deposition Algorithm

The idea of simulating random depositions was first reported in [Vold 59], which includes tables of random numbers used to simulate the sedimentation of moist spheres in hydrocarbons. We shall examine a method of simulation [Fam 85] that results in the deposition shown in Figure 13.3. 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 from the left side of the line. The simulation consists of generating uniform random sites between 0 and L and having a particle stick

Figure 13.3 A simulation of the ballistic deposition of 20,000 particles onto a substrate of length 200. The vertical height increases in proportion to the length of deposition time, with the top being the final surface.



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 it 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:

$$h_r = \begin{cases} h_r + 1, & \text{if } h_r \geq h_{r-1}, \quad h_r > h_{r+1}, \\ \max[h_{r-1}, h_{r+1}], & \text{if } h_r < h_{r-1}, \quad h_r < h_{r+1}. \end{cases} \tag{13.14}$$

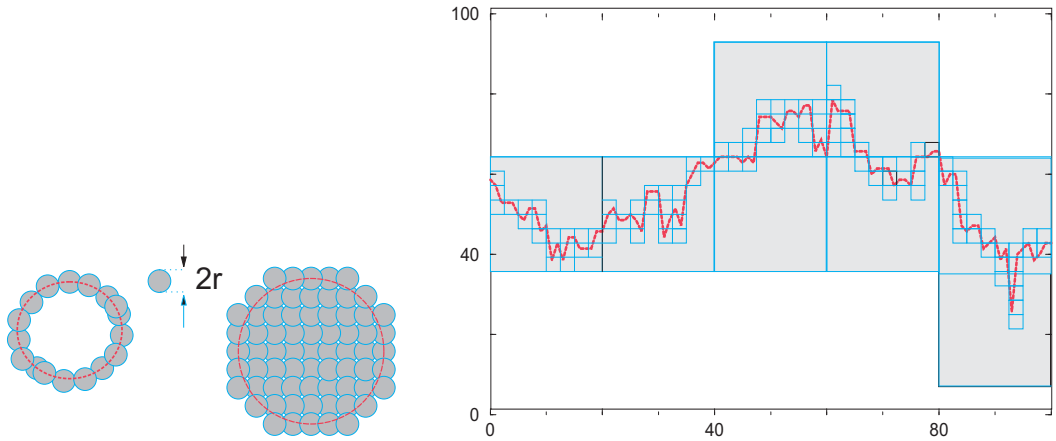
Our simulation is **`Fractals/Film.py`** with the essential loop:

```
spot = int(random)
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 (Figure 13.3), which 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.)

Exercise: Extend the simulation of random deposition to two dimensions, so rather than making a line of particles we now deposit an entire surface.

Figure 13.4 Examples of the use of "box" counting to determine fractal dimension. On the left the "boxes" are circles and the perimeter is being covered. In the middle an entire figure is being covered, and on the right a "coastline" is being covered by boxes of two different sizes (scales). The fractal dimension can be deduced by recording the number of box of different scale needed to cover the figures.



13.5 LENGTH OF BRITISH COASTLINE (PROBLEM 4)

In 1967 Mandelbrot [Mand 67] 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 1 with finite length. However, coastlines are geographic not geometric curves, with each portion of the coast often statistically self-similar to the entire coast yet on a reduced scale. In the latter cases the perimeter may be modeled as a fractal, in which case the length is either infinite or meaningless. Mandelbrot deduced the dimension of the west coast of Britain to be $d_f = 1.25$. In your **problem** we ask you to determine the dimension of the perimeter of one of our fractal simulations.

13.5.1 Coastlines as Fractals (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 it meaning for an object that may be infinitely self-similar. Let us assume that a map maker has a ruler of length r. If she walks along the coastline and counts the number of times N that she must place the ruler down in order to *cover* the coastline, she will obtain a value for the length L of the coast as Nr. Imagine now that the map maker keeps repeating her walk with smaller and smaller rulers. If the coast was a geometric figure or a *rectifiable curve*, at some point the length L would become essentially independent of r and would approach a constant. Nonetheless, as discovered empirically by Richardson [Rich 61] for natural coastlines, such as those of South Africa and Britain, the perimeter appears to be a function of r:

$$L(r) = Mr^{1-d_f}, \tag{13.15}$$

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 $r \to 0$. Yet for a fractal with $d_f > 1$, the perimeter $L \to \infty$ as $r \to 0$. This means that as a consequence of self-similarity, fractals may be of finite size but have infinite perimeters. Physically, at some point there may be no more details to discern as $r \to 0$ (say, at the quantum or Compton size limit), and so the limit may not be meaningful.

13.5.2 Box Counting Algorithm

Consider a line of length L broken up into segments of length r (Figure 13.4 left). The number of segments or "boxes" needed to cover the line is related to the size r of the box by

$$N(r) = \frac{L}{r} = \frac{C}{r}, \tag{13.16}$$

where C is a constant. A proposed definition of fractional dimension is the power of r in this expression as $r \to 0$. 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 would take to *cover* or fill a circle of area A (Figure 13.4 middle), we will find that

$$N(r) = \lim_{r\to 0} \frac{A}{\pi r^2} \quad \Rightarrow \quad d_f = 2, \tag{13.17}$$

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 $r \to 0$ to cover some object, then the fractal dimension d_f can be deduced as

$$N(r) = C\left(\frac{1}{r}\right)^{d_f} = C'\, s^{d_f} \quad (\text{as } r \to 0), \tag{13.18}$$

$$\log N(r) = \log C - d_f \log(r) \quad (\text{as } r \to 0), \tag{13.19}$$

$$\Rightarrow \quad d_f = -\lim_{r\to 0} \frac{\Delta N(r)}{\Delta r}. \tag{13.20}$$

Here $s \propto 1/r$ is called the *scale* in geography, so $r \to 0$ corresponds to an infinite scale. To illustrate, you may be familiar with the low scale on a map being 10,000 m to a centimeter, while the high scale is 100 m to a centimeter. If we want the map to show small details (sizes), we need a map of high 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 (13.15). (If you cannot wait to see box counting in action, in the auxiliary files you will find an applet **`Jfracdim`** that goes through all the steps of box counting before your eyes and even plots the results.)

13.5.3 Coastline Implementation and Exercise

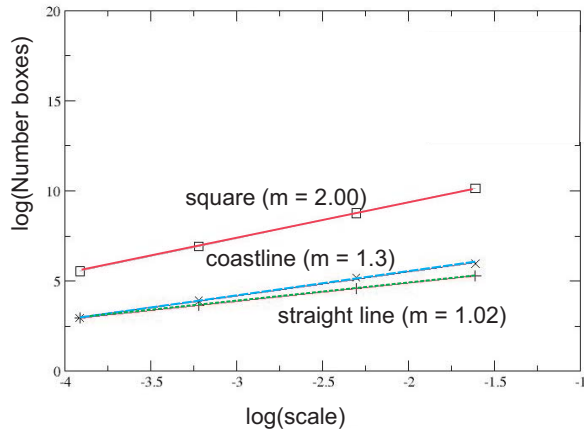
Applet

Rather than ruin our eyes using a geographic map, we use a mathematical one. Specifically, with a little imagination you will see that the top portion of Figure 13.3 looks like a natural coastline. Determine d_f by covering this figure, or one you have generated, with a semitransparent piece of graph paper[1], and counting the number of boxes containing any part of the coastline (Figures 13.4 and 13.5).

1. Print your coastline graph with the same physical scale (*aspect ratio*) for the vertical and horizontal axes. This is required because the graph paper you will use for box counting has square boxes and so you want your graph to also have the same vertical

[1] 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!

Figure 13.5 Fractal dimensions of a line, box, and coastline determined by box counting. The slope at vanishingly small scale determines the dimension.



and horizontal scales. Place a piece of graph paper over your printout and look though the graph paper at your coastline. If you do not have a piece of graph paper available, or if you are unable to obtain a printout with the same aspect ratio for the horizontal and vertical axes, add a series of closely spaced horizontal and vertical lines to your coastline printout and use these lines as your graph paper. (Box counting should still be accurate if both your coastline and your graph paper are have the same aspect ratios.)

2. The vertical height in our printout was 17 cm, and the largest division on our graph paper was 1 cm. This sets the scale of the graph as 1:17, or $s = 17$ for the largest divisions (lowest scale). Measure the vertical height of your fractal, compare it to the size of the biggest boxes on your "piece" of graph paper, and thus determine your lowest scale.
3. With our largest boxes of 1 cm$\times$1 cm, we found that the coastline passed through $N = 24$ large boxes, that is, that 24 large boxes covered the coastline at $s = 17$. Determine how many of the largest boxes (lowest scale) are needed to cover your coastline.
4. With our next smaller boxes of $0.5\,\text{cm} \times 0.5\,\text{cm}$, we found that 51 boxes covered the coastline at a scale of $s = 34$. Determine how many of the midsize boxes (midrange scale) are needed to cover your coastline.
5. With our smallest boxes of 1 mm $\times$ 1 mm, we found that 406 boxes covered the coastline at a scale of $s = 170$. Determine how many of the smallest boxes (highest scale) are needed to cover your coastline.
6. Equation (13.20) tells us that as the box sizes get progressively smaller, we have

$$\log N \simeq \log A + d_f \log s,$$

$$\Rightarrow \quad d_f \simeq \frac{\Delta \log N}{\Delta \log s} = \frac{\log N_2 - \log N_1}{\log s_2 - \log s_1} = \frac{\log(N_2/N_1)}{\log(s_2/s_1)}.$$

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. In our example we found a slope of $d_f = 1.23$. Determine the slope and thus the fractal dimension for your coastline. Although only two points are needed to determine the slope, use your lowest scale point as an important check. (Because the fractal dimension is defined as a limit for infinitesimal box sizes, the highest scale points are more significant.)

7. As given by (13.15), the perimeter of the coastline

$$L \propto s^{1.23-1} = s^{0.23}. \tag{13.21}$$

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 self-similar at all levels, then the scale s will keep getting larger and larger with no limits (or at least until we get down to some quantum limits). This means

$$L \propto \lim_{s\to\infty} s^{0.23} = \infty. \tag{13.22}$$

Does your fractal imply an infinite coastline? Does it make sense that a small island like Britain, which you can walk around, has an infinite perimeter?

13.6 CORRELATED GROWTH, FORESTS, FILMS (PROBLEM 5)

It is an empirical fact that in nature there is increased likelihood that a plant will grow if there is another one nearby (Figure 13.6 left). This *correlation* is also valid for the "growing" of surface films, as in the previous algorithm. Your **problem** is to include correlations in the surface simulation.

13.6.1 Correlated Ballistic Deposition Algorithm

Applet

A variation of the ballistic deposition algorithm, known as the *correlated ballistic deposition algorithm*, simulates mineral deposition onto substrates on which dendrites form [Tait 90]. 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 (Figure 13.6 right):

$$\mathcal{P} = c\, d^{\eta}. \tag{13.23}$$

Here η is a parameter and c is a constant that sets the probability scale.[2] For our implementation we choose $\eta = -2$, which means that there is an inverse square attraction between the particles (decreased probability as they get farther apart).

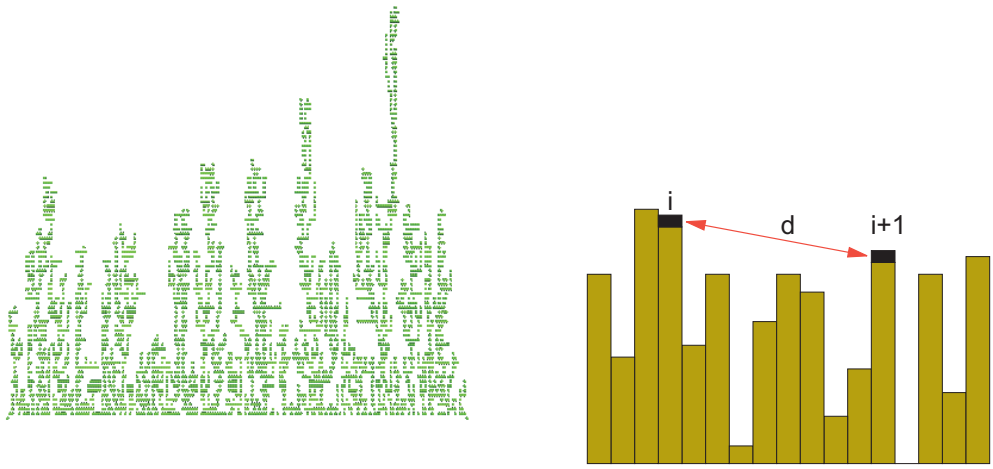
As in our study of uncorrelated deposition, a uniform random number in the interval $[0, L]$ determines the column in which the particle will be deposited. We use the same rules about the heights as before, but now a second random number is used in conjunction with (13.23) to decide if the particle will stick. For instance, if the computed probability is 0.6 and if $r < 0.6$, the particle will be accepted (sticks); if $r > 0.6$, the particle will be rejected.

13.7 GLOBULAR CLUSTER (PROBLEM 6)

Consider a bunch of grapes on an overhead vine. Your **problem** is to determine how its tantalizing shape arises. In a flash of divine insight, you realize that these shapes, as well as others such as those of dendrites, colloids, and thin-film structure, appear to arise from an aggregation process that is limited by diffusion.

[2] 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.

Figure 13.6 *Left:* A view that might be seen in the undergrowth of a forest or after a correlated ballistic deposition. *Right:* The probability of particle $i+1$ sticking in one column depends upon the distance d from the previously deposited particle i.



13.7.1 Diffusion-Limited Aggregation Algorithm

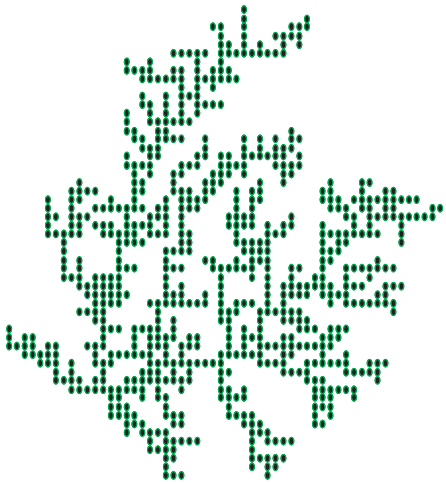
Applet

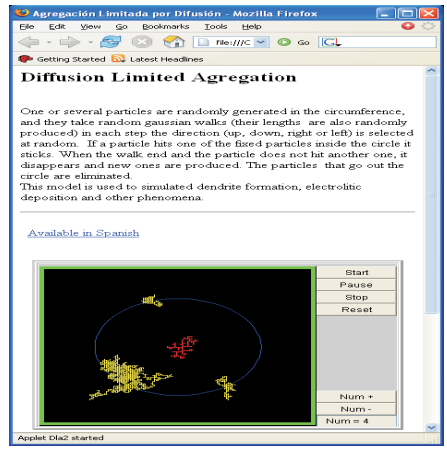
A model of diffusion-limited aggregation (DLA) has successfully explained the relation between a cluster's perimeter and mass [W&S 83]. We start with a 2-D lattice containing a seed particle in the middle, draw a circle around the particle, and place another particle on the circumference of the circle at some random angle. We then release the second particle and have it execute a random walk, much like the one we studied in Chapter 5, "Monte Carlo Simulations," but restricted to vertical or horizontal jumps between lattice sites. This is a type of *Brownian motion* that simulates diffusion. To make the model more realistic, we let the length of each step vary according to a random Gaussian distribution. If at some point during its random walk, the particle encounters 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 (Figure 13.7 and applet **dla**).

Listing 13.2 **Column.py** simulates correlated ballistic deposition of minerals onto substrates on which dendrites form.

```
# Column.py

from visual import *;  import random
maxi = 100000;   npoints = 200                    # Number iterations, spaces
i = 0;              dist = 0;            r = 0;            x = 0;    y = 0
oldx = 0;           oldy = 0;            pp = 0.0;           prob = 0.0
hit = zeros( (200), int)
graph1 = display(width = 500, height = 500,
                 title = 'Correlated Ballistic Deposition', range=250.)
pts = points(color=color.green, size=2)
for i in range(0, npoints):  hit[i] = 0                      # Clear array
oldx = 100;         oldy = 0

for i in range(1, maxi + 1):
    r = int(npoints*random.random() )
    x = r - oldx
    y = hit[r] - oldy
    dist = x*x  +  y*y
    if (dist ==  0): prob = 1.0          # Sticking prob depends on last x
    else: prob = 9.0/dist
    pp = random.random()
    if (pp < prob):
```

Figure 13.7 *Left:* A globular cluster of particles of the type that might occur in a colloid. *Right:* The applet **Dla2en.html** in the auxiliary files lets you watch these clusters grow. Here the cluster is at the center of the circle, and random walkers are started at random points around the circle.






```
        if(r>0 and r<(npoints - 1) ):
            if( (hit[r] >=  hit[r - 1]) and (hit[r] >=  hit[r + 1]) ):
                hit[r] = hit[r] + 1
            else:
                if (hit[r - 1] > hit[r + 1]):
                    hit[r] = hit[r - 1]
                else: hit[r] = hit[r + 1]
        oldx = r
        oldy = hit[r]
        olxc = oldx*2 - 200    # linear TF 0<oldx<200  -> - 200<olxc<200
        olyc = oldy*4 - 200    # linear TF 0<oldy<100  -> - 200<olxy<200
        pts.append(pos=(olxc,olyc))
```

1. Write a subroutine that generates random numbers with a Gaussian distribution.[3]
2. Define a 2-D 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 spacings 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 2π.
6. Compute the x and y positions of the new particle on the circle.
7. Determine whether the particle moves horizontally or vertically by generating a uniform random number $0 < r_{xy} < 1$ and applying the rule

$$\text{if} \begin{cases} r_{xy} < 0.5, & \text{motion is vertical,} \\ r_{xy} > 0.5, & \text{motion is horizontal.} \end{cases} \tag{13.24}$$

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 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:
    a. If occupied, the particle stays at its present position and the walk is over.
    b. If unoccupied, the particle jumps one lattice spacing.

[3] We indicated how to do this in § 6.8.1.

Figure 13.8 Number 8 by the American painter Jackson Pollock. (Used with permission, Neuberger Museum, State University of New York.) Some researchers claim that Pollock's paintings exhibit a characteristic fractal structure, while some other researchers question this [Poll 06]. See if you can determine the fractal dimensions within this painting.

c. 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.

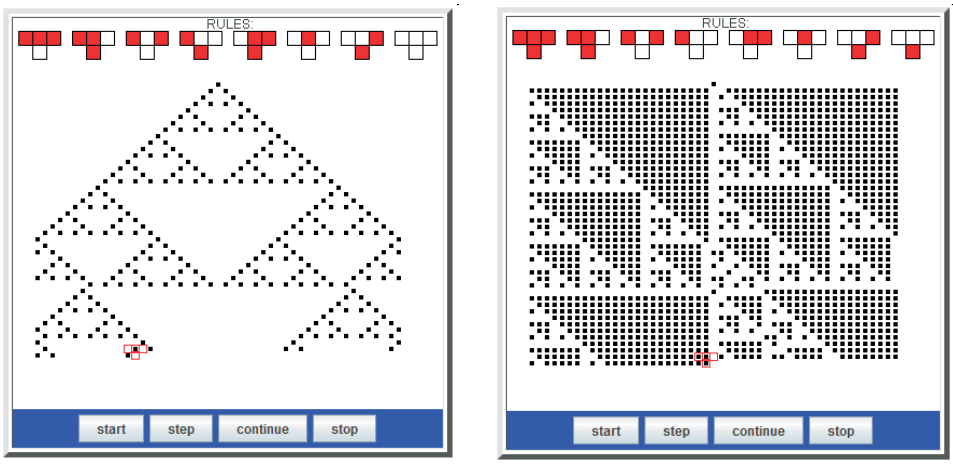
Because many particles are lost, you may need to generate hundreds of thousands of particles to form a cluster of several hundred particles.

13.7.2 Fractal Analysis of DLA or Pollock

A cluster generated with the DLA technique is shown in Figure 13.7. We wish to analyze it to see if the structure is a fractal and, if so, to determine its dimension. (As an alternative, you may analyze the fractal nature of the Pollock painting in Figure 13.8, a technique used to determine the authenticity of this sort of art.) As a control, *simultaneously* analyze a geometric figure, such as a square or circle, whose dimension is known. The analysis is a variation of the one used to determine the length of the coastline of Britain.

1. If you have not already done so, use the box counting method to determine the fractal dimension of a simple square.
2. 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.)
3. Count the number of particles within the square.
4. Compute the density ρ by dividing the number of particles by the number of sites available in the box (49 in our example).
5. Repeat the procedure using larger and larger squares.
6. Stop when the cluster is covered.
7. The (box counting) fractal dimension d_f is estimated from a log-log plot of the density

Figure 13.9 The rules for two versions of the Game of Life. The rules, given graphically on the top row, create the gaskets below. (Output obtained from the applet `JCellAut` in the auxiliary files.)



ρ versus L. If the cluster is a fractal, then (13.2) 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.)

13.8 FRACTALS IN BIFURCATION PLOT (PROBLEM 7)

Recollect the project involving the logistics map where we plotted the values of the stable population numbers *versus* the growth parameter μ. 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.

13.9 FRACTALS FROM CELLULAR AUTOMATA

We have already indicated in places how statistical models may lead to fractals. There is a class of statistical models known as cellular automata *that produce complex behaviors from very simple systems. Here we study some.*

Cellular automata were developed by von Neumann and Ulam in the early 1940s (von Neumann was also working on the theory behind modern computers then). Though very simple, cellular automata have found applications in many branches of science [Peit 94, Sipp 96]. Their classic definition is [Barns 93]:

> A cellular automaton is a discrete dynamical system in which space, time, and the states of the system are discrete. Each point in a regular spatial lattice, called a cell, can have any one of a finite number of states, and the states of the cells in the lattice are updated according to a local rule. That is, the state of a cell at a given time depends only on its won state one time step previously, and the states of its nearby neighbors at the previous time step. All cells on the lattice are updated synchronously, and so the state of the entice lattice advances in discrete time steps.

Applet A cellular automaton in two dimensions consists of a number of square cells that grow upon each other. A famous one, invented by Conway in the 1970s, is Conway's Game of Life. In this, cells with value 1 are alive, while cells with value 0 are dead. Cells grow according to the following rules:

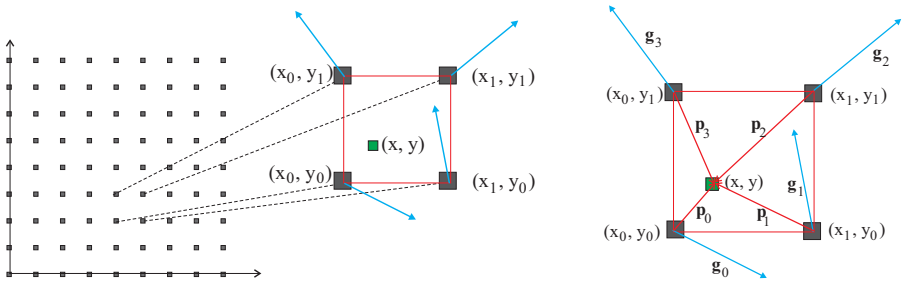
1. If a cell is alive and if two or three of its eight neighbors are alive, then the cell remains alive.
2. If a cell is alive and if more than three of its eight neighbors are alive, then the cell dies because of overcrowding.
3. If a cell is alive and only one of its eight neighbors is alive, then the cell dies of loneliness.
4. If a cell is dead and more than three of its neighbors are alive, then the cell revives.

A variation on the Game of Life is to include a "rule one out of eight" that a cell will be alive if exactly one of its neighbors is alive, otherwise the cell will remain unchanged.

Listing 13.3 **Gameoflife.py** is an extension of Conway's Game of Life in which cells always revive if one out of eight neighbors is alive.

```
# Gameoflife.py: Cellular automata in 2 dimensions

'''* Rules: a cell can be either dead (0) or alive (1)
   * If a cell is alive:
   * on next step will remain alive if
   * 2 or 3 of its closer 8 neighbors are alive.
   * If > 3 of 8 neighbors are alive, cell dies of overcrowdedness
   * If less than  2 neighbors are alive the cell dies of loneliness
   * A dead cell will be alive if  3 of its 8 neighbors are alive'''

from visual.graph import * ; import random
scene = display(width= 500,height= 500, title= 'Game of Life')# 1st array
cell  = zeros((50,50));          cellu = zeros((50,50))
curve(pos= [(-49,-49),(-49,49),(49,49),(49,-49),(-49,-49)],color=color.white)
boxes = points(shape='square', size=8, color=color.cyan)

def drawcells(ce):
    boxes.pos = []                                  # erase previous cells
    for j in range(0,50):
        for i in range(0,50):
            if ce[i,j] == 1:
                xx = 2*i-50
                yy = 2*j-50
                boxes.append(pos=(xx,yy))

def initial():
   for j in range (20,28):                          # initial state of cells
        for  i in range(20, 28):
            r= int(random.random()*2)
            cell[j,i] = r                          # dead or alive at random
   return cell

def gameoflife(cell):                     # rules and  analysis of neighbors
    for i in range(1,49):                 # observe 8 neighbors of cell[i,j]
        for j in range(1,49):
            sum1 = cell[i-1,j-1]+cell[i,j-1]+cell[i+1,j-1] # sum 8 neighb
            sum2 = cell[i-1,j]+cell[i+1,j]+cell[i-1,j+1]+cell[i,j+1]+cell[i+1,j+1]
            alive = sum1+sum2
            if cell[i,j] == 1:
                if  alive == 2 or alive == 3:              # remains alive
                    cellu[i,j] = 1                                 # lives
                if  alive > 3 or alive < 2:     # overcrowded or solitude
                    cellu[i,j] = 0                                  # dies
            if  cell[i,j] == 0:
                if  alive == 3:
                    cellu[i,j] = 1                              # revives
                else:
                    cellu[i,j] = 0                          # remains dead
    alive = 0
    return cellu
temp = initial()
drawcells(temp)
while True:
    rate(6)
    cell = temp
    temp = gameoflife(cell)
```

Figure 13.10 The coordinates used in adding Perlin noise. *Left:* The rectangular grid used to locate a square in space and a corresponding point within the square. As shown with the arrows, unit vectors $\mathbf{g}_i$ with random orientation are assigned at each grid point. *Right:* A point within each square is located by drawing the four $\mathbf{p}_i$. The $\mathbf{g}_i$ vectors are the same as on the left.




```
drawcells(cell)
```

In 1983 Wolfram developed the statistical mechanics of cellular automata and indicated how one can be used to generate a Sierpiński gasket [Wolf 83]. Since we have already seen that a Sierpiński gasket exhibits fractal geometry (§13.2), this represents a microscopic model of how fractals may occur in nature. This model uses eight rules, given graphically at the top of Figure 13.9, to generate new cells from old. We see all possible configurations for three cells in the top row, and the begetted next generation in the row below. At the bottom of Figure 13.9 is a Sierpiński gasket of the type created by the applet **`JCellAut`** in the auxiliary files (under Applets). This plays the game and lets you watch and control the growth of the gasket.

13.10 PERLIN NOISE ADDS REALISM ⊙

We have already seen in this chapter how statistical fractals are able to generate objects with a striking resemblance to those in nature. This appearance of realism may be further enhanced by including a type of coherent randomness known as Perlin noise. The resulting textures so resemble those of clouds, smoke, and fire that one cannot help but wonder if a similar mechanism might also be occurring in nature. The technique we are about to discuss was developed by Ken Perlin of New York University, who won an Academy Award (an Oscar) in 1997 for it and has continued to improve it [Perlin]. This type of coherent noise has found use in important physics simulations of stochastic media [Tick 04], as well as in video games and motion pictures (Tron).

The inclusion of Perlin noise in a simulation adds both randomness and a type of coherence among points in space that tends to make dense regions denser and sparse regions sparser. This is similar to our correlated ballistic deposition simulations (§13.6.1) and related to chaos in its long-range randomness and short-range correlations. We start with some known function of x and y and add noise to it. For this purpose Perlin used the mapping or *ease* function (Figure 13.11 right)

$$f(p) = 3p^2 - 2p^3. \tag{13.25}$$

As a consequence of its S shape, this mapping makes regions close to 0 even closer to 0, while making regions close to 1 even closer (in other words, it increases the tendency to clump, which shows up as higher contrast). We then break space up into a uniform rectangular grid of points (Figure 13.10 left) and consider a point (x, y) within a square with vertices (x_0, y_0), (x_1, y_0), (x_0, y_1), and (x_1, y_1). We next assign unit gradients vectors $\mathbf{g}_0$–$\mathbf{g}_3$ with random orientation at each grid point. A point within each square is located by drawing the four $\mathbf{p}_i$

Figure 13.11 The mapping used in adding Perlin noise. *Left:* The numbers s, t, u, and v are represented by perpendiculars to the four vertices, with lengths proportional to their values. *Right:* The function $3p^2 - 2p^3$ is used as a map of the noise at a point like (x,y) to others close by.

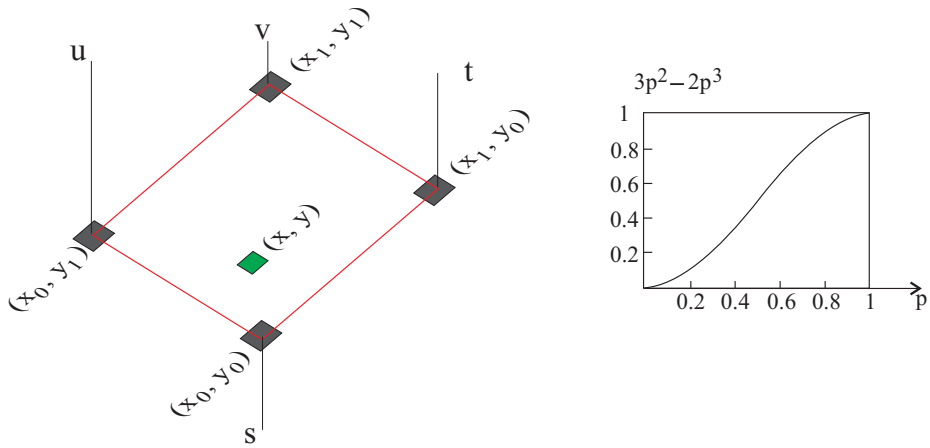
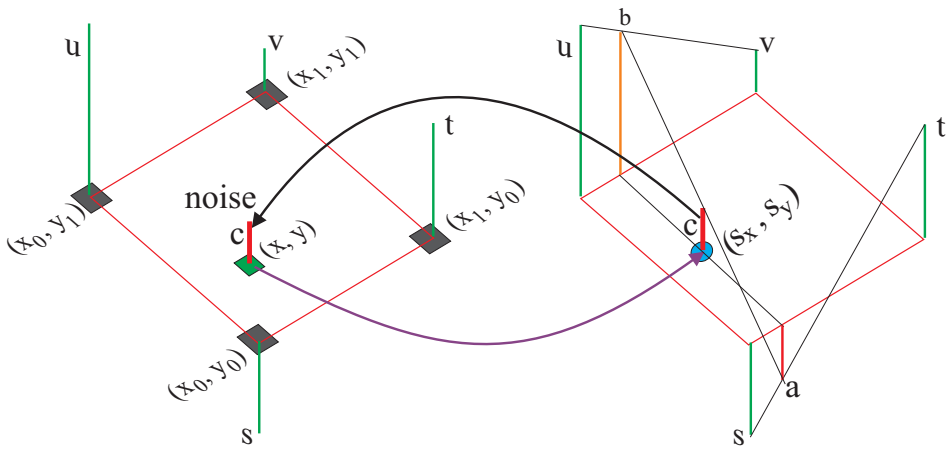


Figure 13.12 Perlin noise mapping. *Left*: The point (x, y) is mapped to point (s_x, x_y). *Right*: Using (13.29). Then three linear interpolations are performed to find c, the noise at (x, y).



vectors (Figure13.10 right):

$$\mathbf{p}_0 = (x - x_0)\mathbf{i} + (y - y_0)\mathbf{j}, \quad \mathbf{p}_1 = (x - x_1)\mathbf{i} + (y - y_0)\mathbf{j}, \tag{13.26}$$

$$\mathbf{p}_2 = (x - x_1)\mathbf{i} + (y - y_1)\mathbf{j}, \quad \mathbf{p}_3 = (x - x_0)\mathbf{i} + (y - y_1)\mathbf{j}. \tag{13.27}$$

Next the scalar products of the $\mathbf{p}'$s and the $\mathbf{g}'$s are formed:

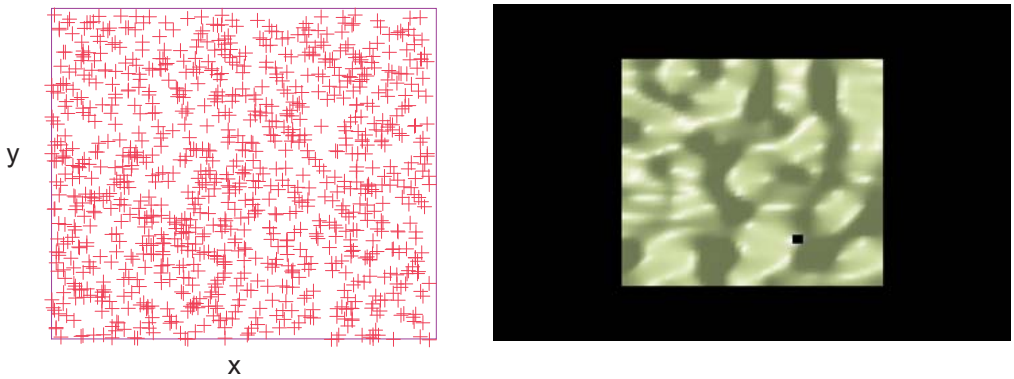
$$s = \mathbf{p}_0 \cdot \mathbf{g}_0, \quad t = \mathbf{p}_1 \cdot \mathbf{g}_1, \quad v = \mathbf{p}_2 \cdot \mathbf{g}_2, \quad u = \mathbf{p}_3 \cdot \mathbf{g}_3. \tag{13.28}$$

As shown on the left in Figure 13.11, the numbers s, t, u, and v are assigned to the four vertices of the square and represented there by lines perpendicular to the square with lengths proportional to the values of s, t, u, and v (which can be positive or negative).

The actual mapping proceeds via a number of steps (Figure 13.12):

1. Transform the point (x, y) to (s_x, s_y),
$$s_x = 3x^2 - 2x^3, \quad s_y = 3y^2 - 2y^3. \tag{13.29}$$
2. Assign the lengths s, t, u, and v to the vertices in the mapped square.

Figure 13.13 After the addition of Perlin noise, the random scatterplot on the left becomes the clusters on the right.



3. Obtain the height a (Figure 13.12) via linear interpolation between s and t.
4. Obtain the height b via linear interpolation between u and v.
5. Obtain s_y as a linear interpolation between a and b.
6. The vector c so obtained is now the two components of the noise at (x, y).

Perlin's original C code to accomplish this mapping (along with other goodies) is found in [Perlin]. It takes as input the plot of random points (r_{2i}, r_{2i+1}) on the left in Figure 13.13 (which is the same as Figure 5.1) and by adding coherent noise produces the image on the right in Figure 13.13. The changes we made from the original program are (1) including an `int` before the variables `p[], start, i,` and `j`, and (2) adding `# include <time.h>` and the line `srand(time(NULL));` at the beginning of method `init( )` in order to obtain different random numbers each time the program runs. The main method of the C program we used is below. The program outputs a data file that we visualized with OpenDX to produce the image `montania.tiff` on the right in Figure 13.13.

13.10.1 Including Ray Tracing

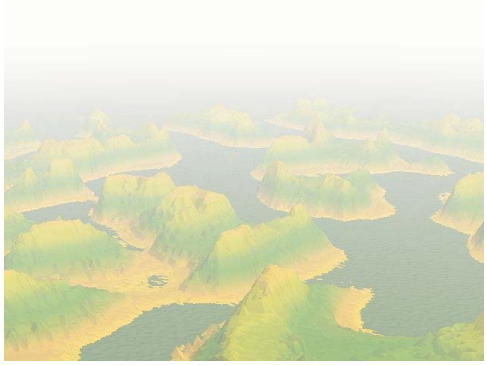
Ray tracing is a technique that renders an image of a scene by simulating the way rays of light actually travel [Pov-Ray]. To avoid tracing rays that do not contribute to the final image, ray-tracing programs start at the viewer, trace rays backward onto the scene, and then back again onto the light sources. You can vary the location of the viewer and light sources and the properties of the objects being viewed, as well as atmospheric conditions such as fog, haze, and fire.

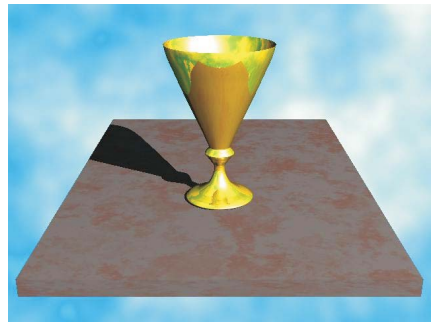
As an example of what can be done, on the left in Figure 13.14 we show the output from the ray-tracing program Pov-Ray [Pov-Ray], using as input the coherent random noise on the right in Figure 13.13. The program options we used are given in Listing 13.4 and are seen to include commands to color the islands, to include waves, and to give textures to the sky and the sea. Pov-Ray also allows the possibility of using Perlin noise to give textures to the objects to be created. For example, the stone cup on the right in Figure 13.14 has a marblelike texture produced by Perlin noise.

Listing 13.4 **Islands.pov** in the Codes/Animations/Fractals/ directory gives the `Pov-Ray` ray-tracing commands needed to convert the coherent noise random plot of Figure 13.13 into the mountain-like image in Figure 13.14 left.

```
// Islands.pov  Pov–Ray program to create Islands, by Manuel J Paez
plane {
```

Figure 13.14 *Left:* The output from the Pov-Ray ray-tracing program that took as input the 2-D coherent random noise plot in Figure 13.13 and added height and fog. *Right:* An image of a surface of revolution produced by Pov-Ray in which the marblelike texture is created by Perlin noise.





```
  <0, 1, 0>, 0                                                    // Sky
  pigment { color rgb <0, 0, 1> }
  scale 1
  rotate <0, 0, 0>
  translate y*0.2
}
global_settings {
  adc_bailout 0.00392157
  assumed_gamma 1.5
  noise_generator 2
}
#declare Island_texture = texture {
  pigment {
    gradient <0, 1, 0>                              // Vertical direction
    color_map {                                      // Color the islands
      [ 0.15 color rgb <1, 0.968627, 0> ]
      [ 0.2  color rgb <0.886275, 0.733333, 0.180392>   ]
      [ 0.3  color rgb <0.372549, 0.643137, 0.0823529>  ]
      [ 0.4  color rgb <0.101961, 0.588235, 0.184314>   ]
      [ 0.5  color rgb <0.223529, 0.666667, 0.301961>   ]
      [ 0.6  color rgb <0.611765, 0.886275, 0.0196078>  ]
      [ 0.69 color rgb <0.678431, 0.921569, 0.0117647>  ]
      [ 0.74 color rgb <0.886275, 0.886275, 0.317647>   ]
      [ 0.86 color rgb <0.823529, 0.796078, 0.0196078>  ]
      [ 0.93 color rgb <0.905882, 0.545098, 0.00392157> ]
      }
   }
   finish {
     ambient rgbft <0.2, 0.2, 0.2, 0.2, 0.2>
     diffuse 0.8
   }
}
camera {                                  // Camera characteristics and location
  perspective
  location <-15, 6, -20>                                      // Located here
  sky <0, 1, 0>
  direction <0, 0, 1>
  right <1.3333, 0, 0>
  up <0, 1, 0>
  look_at <-0.5, 0, 4>                                  //looking at that point
  angle 36
}
light_source {<-10, 20, -25>, rgb <1, 0.733333, 0.00392157>}        // Light

#declare Islands = height_field {              // Takes  gif and finds heights
  gif "d:\pov\montania.gif"                        // Windows directory naming
  scale <50, 2, 50>
  translate <-25, 0, -25>
}
object {                                                            // Islands
  Islands
  texture {
    Island_texture
    scale 2
  }
}
box {                                        // Upper face of the box is the sea
  <-50, 0, -50>, <50, 0.3, 50>                // Location of 2 opposite vertices
  translate <-25, 0, -25>
```

```
  texture {                                                    // Simulate waves
    normal {
      spotted
      0.4
      scale <0.1, 1, 0.1>
    }
    pigment { color rgb <0.164706, 0.556863, 0.901961> }
  }
}
fog {                                                  // A constant fog is defined
   fog_type 1
   distance 30
    rgb <0.984314, 1, 0.964706>
}
```

13.11 QUIZ

1. Recall how box counting is used to determine the fractal dimension of an object. Imagine that the result of some experiment or simulation is an interesting geometric figure.
   a. What might be the physical/theoretical importance of determining that this object is a fractal?
   b. What might be the importance of determining its fractal dimension?
   c. Why is it important to use *more* than two sizes of boxes?
   d. Below is a figure composed of boxes of side 1.

1	1	1
1	1	1
1	1	1

   Use box counting to determine the fractal dimension of this figure.

Chapter Fourteen

High-Performance and Parallel Computing Hardware, and Programming for It

This chapter discusses a number of topics associated with high-performance computing (HPC) as performed on a supercomputer. Although this may sound like something only specialists should be reading, assuming history as a guide, present HPC hardware and software will be desktop machines in less than a decade. In Unit I we discuss the theory of a high-performance computer's memory and central processor design, then Unit II examines parallel computing, and finally Unit III leads the reader through a tutorial that demonstrates some techniques for writing programs that are optimized for HPC hardware. By working through the tutorial part of the chapter, the reader will experiment with their computer's memory and experience some of the concerns, techniques, rewards, and shortcomings of HPC. The chapter concludes with some practical tips appropriate for the forthcoming peta- and exascale computing.

HPC is a broad subject, and our presentation is brief and given from a practitioner's point of view. The text [Quinn 04] surveys parallel computing and MPI from a computer science point of view. References on parallel computing include [Quinn 04, Pan 96, VdeV 94, Fox 94]. More recent developments, such as programming for multicore computers, cell computers, and field-programmable gate accelerators, are discussed in journals and magazines [CiSE] and somewhat at the end of the chapter.

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
High Performance Computing	pdf	14.1–.4	HPC Hardware	pdf	14.4–.6, 14.13
HPC Exercises	pdf	14.14–.15	HPC Exercises II	pdf	14.14–.15
Parallel Computing	pdf	14.10			

14.1 UNIT I. HIGH-PERFORMANCE COMPUTERS (CS)

By definition, supercomputers are the fastest and most powerful computers available, and at this instant, the term "supercomputers" almost always refers to parallel machines. They are the superstars of the high-performance class of computers. 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 a good balance among the following major elements:

- Multistaged (pipelined) functional units.
- Multiple central processing units (CPUs) (parallel machines).

Figure 14.1 The logical arrangement of the CPU and memory showing a Fortran array $A(N)$ and matrix $M(N, N)$ loaded into memory.

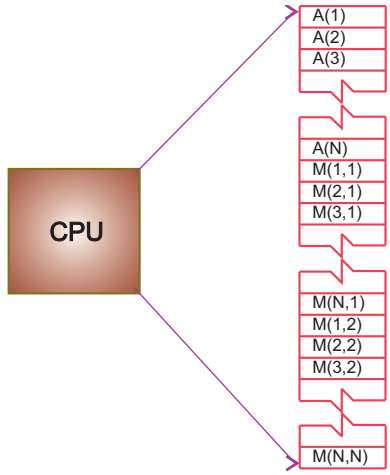
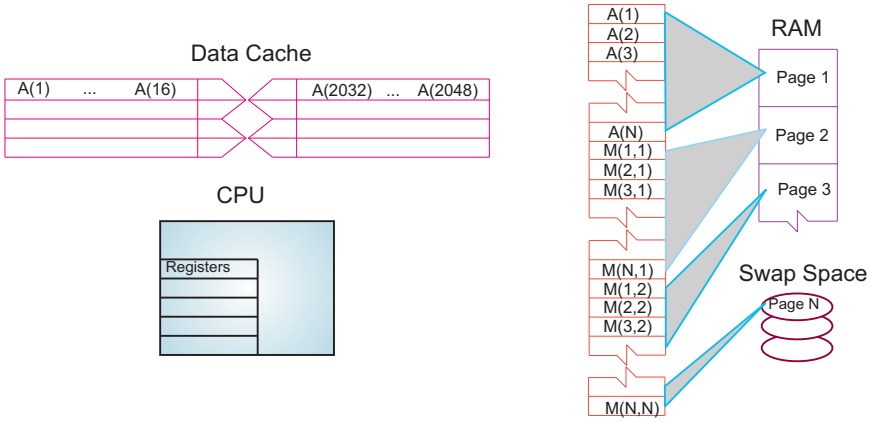


Figure 14.2 The elements of a computer's memory architecture in the process of handling matrix storage.



- Multiple cores.
- Fast central registers.
- Very large, fast memories.
- Very fast communication among functional units.
- Vector, video, or array processors.
- Software that integrates the above effectively.

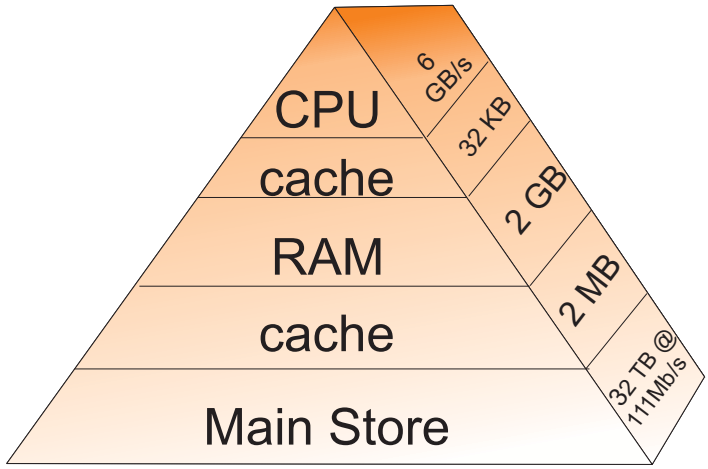
As a simple example, it makes little sense to have a CPU of incredibly high speed coupled to a memory system and software that cannot keep up with it (the present state of affairs).

14.2 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 Figure 14.1 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 is in *column-major* order:

$$M(1,1)\, M(2,1)\, M(3,1)\, M(1,2)\, M(2,2)\, M(3,2)\, M(1,3)\, M(2,3)\, M(3,3),$$

Figure 14.3 Typical memory hierarchy for a single-processor, high-performance computer (B = bytes, K, M, G, T = kilo, mega, giga, tera).



while in Python, Java and C it is 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 place. Some may be in RAM, some on the disk, some in cache, and some in the CPU. To give some of these words more meaning, in Figures 14.2 and 14.3 we show simple models of the 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* 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.

Cache (high-speed buffer): A small, very fast bit of memory that holds instructions, addresses, and data in their passage between the very fast CPU registers and the slower RAM. This is seen in the next level down the pyramid in Figure 14.3. The main memory is also called *dynamic RAM* (DRAM), while the cache is called *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.

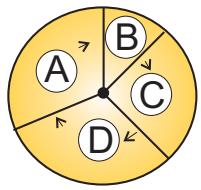
Cache and data lines: The data transferred to and from the cache or CPU are grouped into cache lines or data lines. The time it takes to bring data from memory into the cache is called *latency*.

RAM: Random-access memory or central memory is in the middle memory in the hierarchy in Figure 14.3. RAM can be accessed directly, that is, in random order, and it can be accessed quickly, that is, without mechanical devices. It is where your program resides while it is being processed.

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 a book; they are numbered and kept track of with a *table of contents*. Typical page sizes are from 4 to 16 KB.

Hard disk: Finally, at the bottom of the memory pyramid is permanent storage on magnetic

Figure 14.4 Multitasking of four programs in memory at one time in which the programs are executed in round-robin order.



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 to the CPU, the other memories are 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 Figure 14.2 for our example, virtual memory permits your program to use more pages of memory than can 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* (Figure 14.2). 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 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 can physically fit into RAM (Figure 14.4). 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 a 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 the least used application still in memory is simultaneously being paged out.

Table 14.1 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

14.3 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 the 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, Table 14.1 indicates how the operation $c = (a + b)/(d * f)$ is handled. 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 has something to do (during which time the other two sit idle).

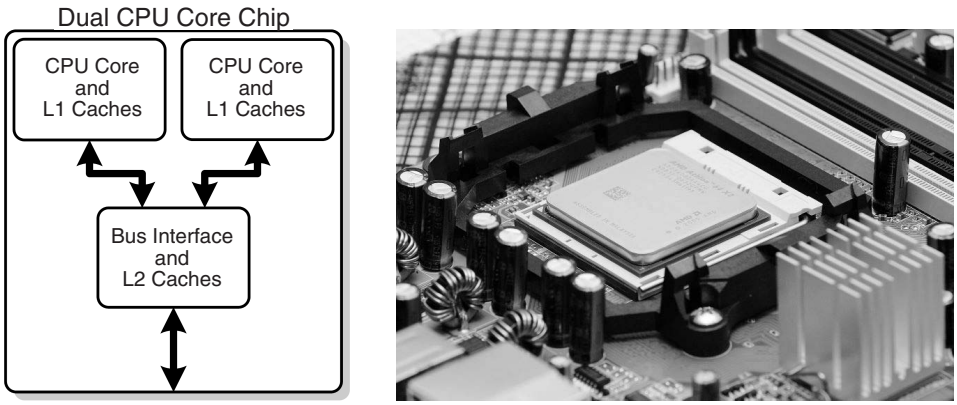
14.4 CPU DESIGN: REDUCED INSTRUCTION SET COMPUTER

Reduced instruction set computer (RISC) architecture (also called *superscalar*) is a design philosophy for CPUs 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 *complex instruction set computer* (CISC), architecture. 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 chip designers and containing machine language instructions that set the operating characteristics of the computer. There were more than 1000 instructions available, and many were similar to higher-level programming languages like *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 about 30 low-level instructions accounted for 99% of the use with only 10 of these instructions accounting for 80% 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

Figure 14.5 *Left:* A generic view of the Intel core-2 dual-core processor, with CPU-local level-1 caches and a shared, on-die level-2 cache (courtesy of D. Schmitz). *Right:* The AMD Athlon 64 X2 3600 dual-core CPU (Wikimedia Commons).



execution time of a program:

$$\text{CPU time} = \text{no.instructions} \times \text{cycles/instruction} \times \text{cycle time}. \tag{14.1}$$

Here "CPU time" is the time required by a program, "no. 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 one CPU cycle. After viewing (14.1) we can understand the CISC philosophy, which tries to reduce CPU time by reducing no. instructions, as well as the RISC philosophy, which tries to reduce CPU time by reducing cycles/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 the following:

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.

14.5 CPU DESIGN: MULTIPLE-CORE PROCESSORS

The year preceding the publication of this book has seen a rapid increase in the inclusion of dual-core, or even quad-core, chips as the computational engine of computers. As seen in Figure 14.5, a dual-core chip has two CPUs in one integrated circuit with a shared interconnect and a shared level-2 cache. This type of configuration with two or more identical processors connected to a single shared main memory is called *symmetric multiprocessing*, or SMP.. It is likely that by the time you read this book, 16-core or greater chips will be available.

Although multicore chips were designed for game playing and single precision, they should also be useful in scientific computing if new tools, algorithms, and programming methods are employed. These chips attain more speed with less heat and more energy efficiency than single-core chips, whose heat generation limits them to clock speeds of less than 4 GHz.

Table 14.2 Computation of Matrix $[C] = [A] + [B]$

Step 1	*Step 2*	$\cdots$	*Step 99*
$c(1) = a(1) + b(1)$	$c(2) = a(2) + b(2)$	$\cdots$	$c(99) = a(99) + b(99)$

Table 14.3 Vector Processing of Matrix $[A] + [B] = [C]$

Step 1	Step 2	$\cdots$	Step Z
$c(1) = a(1) + b(1)$			
	$c(2) = a(2) + b(2)$		
		$\cdots$	
			$c(Z) = a(Z) + b(Z)$

In contrast to multiple single-core chips, multicore chips use fewer transistors per CPU and are thus simpler to make and cooler to run.

Parallelism is built into a multicore chip because each core can run a different task. However, since the cores usually share the same communication channel and level-2 cache, there is the possibility of a communication bottleneck if both CPUs use the bus at the same time. Usually the user need not worry about this, but the writers of compilers and software must so that your code will run in parallel. Modern Intel compilers make use of each multiple core and even have MPI treat each core as a separate processor.

14.6 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 (Table 14.2). There is actually much behind-the-scenes 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 numerical values of these two elements in a CPU register, and the *storage* in memory of the sum in $c(i)$. 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 can also handle single-subscripted matrices, that is, mathematical vectors.) In the vector processing of $[A] + [B] = [C]$, the successive fetching of and addition of the elements A and B are grouped together and overlaid, and $Z \simeq 64$–256 elements (the *section size*) are processed with one command, as seen in Table 14.3. Depending on the array size, this method 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 ~ 64–256.

Vector processing probably had its heyday during the time when computer manufacturers produced large mainframe computers designed for the scientific and military communities. 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 holds in store?

14.7 UNIT II. PARALLEL COMPUTING

There is little question that advances in the hardware for parallel computing are impressive. Unfortunately, the software that accompanies the hardware often seems stuck in the 1960s. In our view, message passing has too many details for application scientists to worry about and requires coding at a much, or more, elementary level than we prefer. However, the increasing occurrence of clusters in which the nodes are symmetric multiprocessors has led to the development of sophisticated compilers that follow simpler programming models; for example, *partitioned global address space* compilers such as *Co-Array Fortran*, *Unified Parallel C*, and *Titanium*. In these approaches the programmer views a global array of data and then manipulates these data as if they were contiguous. Of course the data really are distributed, but the software takes care of that outside the programmer's view. Although the program may not be as efficient a use of the processors as hand coding, it is a lot easier, and as the number of processors becomes very large, one can live with a greater degree of inefficiency. In any case, if each node of the computer has a number of processors with a shared memory and there are a number of nodes, then some type of a hybrid programming model will be needed.

Problem: Start with the program you wrote to generate the bifurcation plot for bug dynamics in Chapter 12, "Discrete & Continuous Nonlinear Dynamics," and modify it so that different ranges for the growth parameter μ are computed simultaneously on multiple CPUs. Although this small a problem is not worth investing your time in to obtain a shorter turnaround time, it is worth investing your time into it gain some experience in parallel computing. In general, parallel computing holds the promise of permitting you to obtain 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.

14.8 PARALLEL SEMANTICS (THEORY)

We saw earlier 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 [Quinn 04]. It is a computing environment in which some number 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:

$$[B] = [A][B] \quad \Rightarrow \quad B_{i,j} = \sum_{k=1}^{N} A_{i,k} B_{k,j}. \tag{14.2}$$

Because the computation of $B_{i,j}$ for particular values of i and j is independent of the computation of all the other values, each $B_{i,j}$ can be computed in parallel, or each row or column of $[B]$ can be computed in parallel. However, because $B_{k,j}$ on the RHS of (14.2) 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 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 $[C] = [A][B]$ is 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 CPU 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 in handling instructions and data. From this viewpoint there are three types of machines:

- **Single-instruction, single-data (SISD):** These are the classic (von Neumann) 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 types 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 described in §14.13. 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.

The running of independent programs on a parallel computer is similar to the multitasking feature used by Unix and PCs. In multitasking (Figure 14.6 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 runs at a single time, but if other programs are in memory, then it does not take long to switch to them. In multiprocessing (Figure 14.6 right) these jobs may all run at the same time, either in different parts of memory or in the memory of different computers. Clearly, multiprocessing becomes complicated if separate processors are operating on different parts of the *same* program because then synchronization and load balance (keeping all the processors equally busy) are concerns.

In addition to instructions and data streams, another way of categorizing 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. An illustration is six Linux PCs sharing the same files across a network but with a different central memory system for each PC. Each computer can be operating on a different, 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 in the program.[1]

[1] Some experts define our medium grain as coarse grain yet this distinction changes with time.

Figure 14.6 *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.

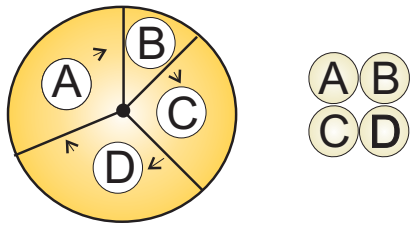
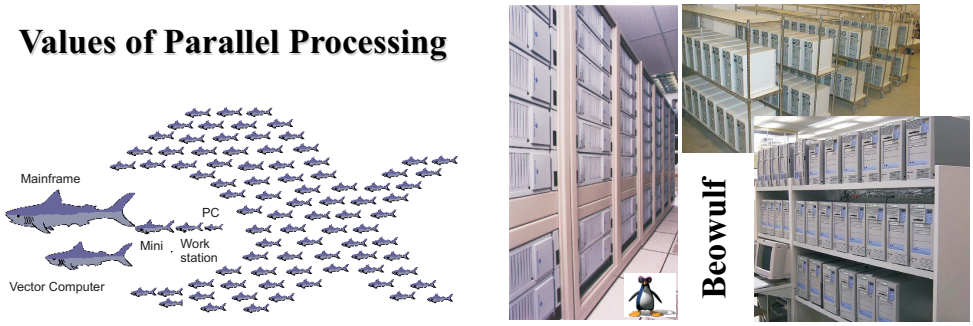


Figure 14.7 Two views of modern parallel computing (courtesy of Yuefan Deng).



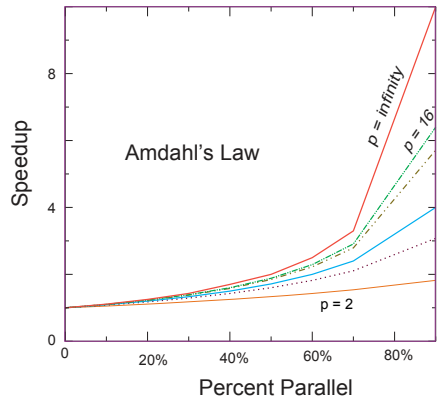
- **Fine-grain parallel:** As the granularity decreases and the number of nodes increases, there is an increased requirement for fast communication among the nodes. For this reason fine-grain systems tend to be custom-designed machines. The communication may be via a central bus or via shared memory for a small number of nodes, or through some form of high-speed network for massively parallel machines. In the 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.

14.9 DISTRIBUTED MEMORY PROGRAMMING

An approach to concurrent processing that, because it is built from commodity PCs, has gained dominant 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*[2] are examples of distributed memory computers (Figure 14.7). 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. In a Beowulf cluster, the components are commodity ones designed for a general market, as are the communication network and its high-speed switch (special interconnects are used by major commercial manufacturers, 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 are not usually called a Beowulf.

[2]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 computers from common elements that have surpassed the performance of major corporations and their proprietary, multi-million-dollar supercomputers.

Figure 14.8 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.



The literature contains frequent arguments concerning the differences among clusters, commodity clusters, Beowulfs, constellations, massively parallel systems, and so forth [Dong 05]. Even though we recognize that there are major differences between the clusters on the top 500 list of computers and the ones that 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 message-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 the nodes. When all the 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 the nodes. Often a separate host program, which starts the programs on the nodes, reads the input files and organizes the output.

14.10 PARALLEL PERFORMANCE

Imagine a cafeteria line in which all the servers appear to be working hard and fast yet the ketchup dispenser has some relish 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 "relish" may be the issuing and communicating of instructions. Because the computation cannot advance until all the instructions have been received, this one step may slow down or stop the entire process.

As we soon will demonstrate, the speedup of a program will not be significant unless you can get ∼90% of it to run in parallel, and even then most of the speedup will probably be obtained with only a small number of processors. This means that you need to have a computationally intense problem to make parallelization worthwhile, and that is one of the reasons why some proponents of parallel computers with thousands of processors suggest that you not apply the new machines to old problems but rather look for new problems that 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 [Amd 67, Quinn 04]. Let

$$p = \text{no. of CPUs} \quad T_1 = \text{1-CPU time}, \quad T_p = p\text{-CPU time}. \tag{14.3}$$

The maximum speedup S_p attainable with parallel processing is thus

$$S_p^{\max} = \frac{T_1}{T_p} \to p. \tag{14.4}$$

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 take time, and it is rare to attain a 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 be the fraction of the program that potentially may run on multiple processors. The fraction $1-f$ of the code that cannot be run in parallel must be run via serial processing and thus takes time:

$$T_s = (1-f)T_1 \quad \text{(serial time)}. \tag{14.5}$$

The time T_p spent on the p parallel processors is related to T_s by

$$T_p = f\,\frac{T_1}{p}. \tag{14.6}$$

That being so, the speedup S_p as a function of f and the number of processors is

$$S_p = \frac{T_1}{T_s + T_p} = \frac{1}{1 - f + f/p} \quad \text{(Amdahl's law)}. \tag{14.7}$$

Some theoretical speedups are shown in Figure 14.8 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 the 90% of your in- parallel figure comes 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.

14.10.1 Communication Overhead

As discouraging as Amdahl's law may seem, it actually *overestimates* speedup because it ignores the *overhead* for parallel computation. Here we look at communication overhead. Assume a completely parallel code so that its speedup is

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{T_1/p} = p. \tag{14.8}$$

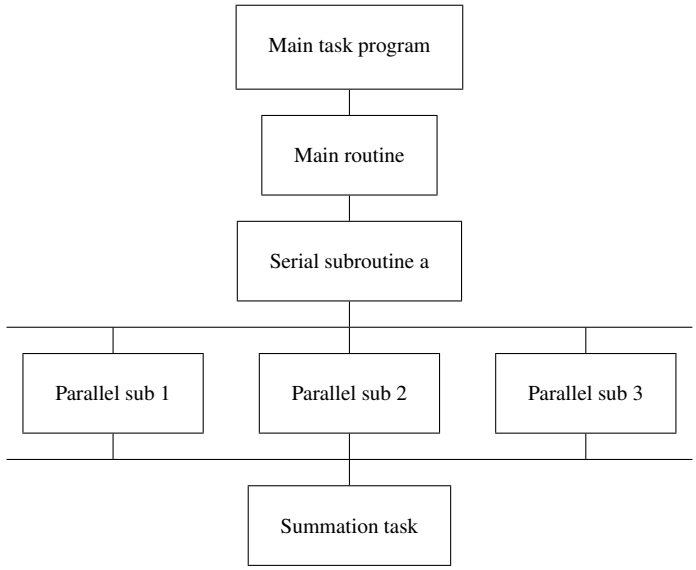
The denominator assumes that it takes no time for the processors to communicate. However, it take a finite time, called *latency*, to get data out of memory and into the cache or onto the communication network. When we add in this latency, as well as other times that make up the *communication time* T_c, the speedup decreases to

$$S_p \simeq \frac{T_1}{T_1/p + T_c} < p \quad \text{(with communication time)}. \tag{14.9}$$

For the speedup to be unaffected by communication time, we need to have

$$\frac{T_1}{p} \gg T_c \;\Rightarrow\; p \ll \frac{T_1}{T_c}. \tag{14.10}$$

Table 14.4 A typical organization of a program containing both serial and parallel tasks.



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, the slowest step in a process is usually the rate-determining step, and the increasing speed of CPUs means that this slowest step is more and more often access to or communication among 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.

14.11 PARALLELIZATION STRATEGY

A typical organization of a program containing both serial and parallel tasks is given in Table14.4. The user organizes the work into units called *tasks*, with each task assigning work (*threads*) to a processor. The main task controls the overall execution as well as the subtasks that run independent parts of the program (called *parallel subroutines*, *slaves*, *guests*, or *subtasks*). These parallel subroutines can be distinctive subprograms, multiple copies of the same subprogram, or even **`for`** loops.

It is the programmer's responsibility to ensure that the breakup of a code into parallel subroutines is mathematically and scientifically valid and is an equivalent formulation of the original program. As a case in point, if the most intensive part of a program is the evaluation of a large Hamiltonian matrix, you may want to evaluate each row on a different processor. Consequently, the key to parallel programming is to identify the parts of the program that may benefit from parallel execution. To do that the programmer should understand the program's data structures (see below), know in what order the steps in the computation must be performed,

and know how to coordinate the results generated by different processors.

The programmer helps speed up the execution by keeping many processors simultaneously busy and by avoiding storage conflicts among different parallel subprograms. You do this *load balancing* by dividing your program into subtasks of approximately equal numerical intensity that will run simultaneously on different processors. The rule of thumb is to make the task with the largest granularity (workload) dominant by forcing it to execute first and to keep all the processors busy by having the number of tasks an integer multiple of the number of processors. This is not always possible.

The individual parallel threads can have *shared* or *local* data. The shared data may be used by all the machines, while the local data are private to only one thread. To avoid storage conflicts, design your program so that parallel subtasks use data that are independent of the data in the main task and in other parallel tasks. This means that these data should not be modified *or even examined* by different tasks simultaneously. In organizing these multiple tasks, reduce communication *overhead costs* by limiting communication and synchronization. These costs tend to be high for fine-grain programming where much coordination is necessary. However, *do not* eliminate communications that are necessary to ensure the scientific or mathematical validity of the results; bad science can do harm!

14.12 PRACTICAL ASPECTS OF MIMD MESSAGE PASSING

It makes sense to run only the most numerically intensive codes on parallel machines. Frequently these are very large programs assembled over a number of years or decades by a number of people. It should come as no surprise, then, that the programming languages for parallel machines are primarily Fortran90, which has explicit structures for the compiler to parallelize, and C. [In the past we have not attained good speedup with Java in parallel, are told that *F-MPJ* (http://jfs.des.udc.es/) and *MPJ Express* (http://mpj-express.org/) have fixed those problems.] Effective parallel programming becomes more challenging as the number of processors increases. Computer scientists suggest that it is best *not* to attempt to modify a serial code but instead to rewrite it from scratch using algorithms and subroutine libraries best suited to parallel architecture. However, this may involve months or years of work, and surveys find that $\sim 70\%$ of computational scientists revise existing codes [Pan 96].

Most parallel computations at present are done on a multiple-instruction, multiple-data computers via message passing. MPI is the most common message-passing interface. Here we outline some practical concerns based on user experience [Dong 05, Pan 96].

Parallelism carries a price tag: There is a steep learning curve requiring intensive effort.Failures may occur for a variety of reasons, especially because parallel environments tend to change often and get "locked up" by a programming error. In addition, with multiple computers and multiple operating systems involved, the familiar techniques for debugging may not be effective.

Preconditions for parallelism: If your program is run thousands of times between changes, with execution time in days, and you must significantly increase the resolution of the output or study more complex systems, then parallelism is worth considering. Otherwise, and to the extent of the difference, parallelizing a code may not be worth the time investment.

The problem affects parallelism: You must analyze your problem in terms of how and when data are used, how much computation is required for each use, and the type of problem architecture:

- **Perfectly parallel:** The same application is run simultaneously on different data sets, with the calculation for each data set independent (e.g., running multiple versions of a Monte Carlo simulation, each with different seeds, or analyzing data from independent detectors). In this case it would be straightforward to parallelize with a respectable performance to be expected.
- **Fully synchronous:** The same operation applied in parallel to multiple parts of the same data set, with some waiting necessary (e.g., determining positions and velocities of particles simultaneously in a molecular dynamics simulation). Significant effort is required, and unless you balance the computational intensity, the speedup may not be worth the effort.
- **Loosely synchronous:** Different processors do small pieces of the computation but with intermittent data sharing (e.g., diffusion of groundwater from one location to another). In this case it would be difficult to parallelize and probably not worth the effort.
- **Pipeline parallel:** Data from earlier steps processed by later steps, with some overlapping of processing possible (e.g., processing data into images and then into animations). Much work may be involved, and unless you balance the computational intensity, the speedup may not be worth the effort.

14.12.1 High-Level View of Message Passing

Although it is true that parallel computing programs may become very complicated, the basic ideas are quite simple. All you need is a regular programming language like C or Fortran, plus four communication statements:[3]

`send`: One processor sends a message to the network. It is not necessary to indicate who will receive the message, but it must have a name.
`receive`: One processor receives a message from the network. This processor does not have to know who sent the message, but it has to know the message's name.
`myid`: An integer that uniquely identifies each processor.
`numnodes`: An integer giving the total number of nodes in the system.

Once you have made the decision to run your program on a computer cluster, you will have to learn the specifics of a message-passing system such as MPI. Here we give a broader view. When you write a message-passing program, you intersperse calls to the message-passing library with your regular Fortran or C program. The basic steps are

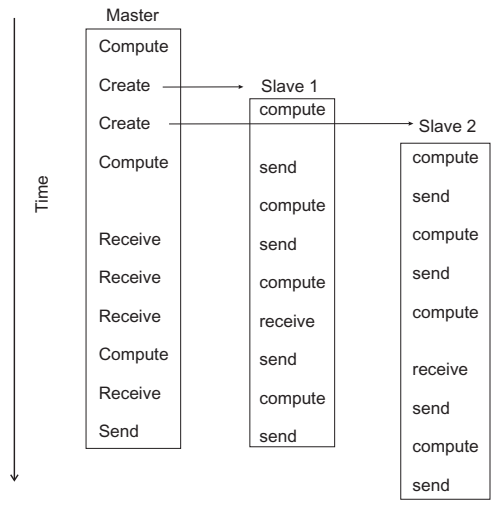
1. Submit your job from the command line or a job control system.
2. Have your job start additional processes.
3. Have these processes exchange data and coordinate their activities.
4. Collect these data and have the processes stop themselves.

We show this graphically in Figure 14.9 where at the top we see a *master* process create two *slave* processes and then assign work for them to do (arrows). The processes then communicate with each other via message passing, output their data to files, and finally terminate.

What can go wrong: Figure 14.9 also illustrates some of the difficulties:

[3] Personal communication, Yuefan Deng.

Figure 14.9 A master process and two slave processes passing messages. Notice how this program has more sends than receives and consequently may lead to results that depend on order of execution, or may even lock up.



- The programmer is responsible for getting the processes to cooperate and for dividing the work correctly.
- The programmer is responsible for ensuring that the processes have the correct data to process and that the data are distributed equitably.
- The commands are at a lower level than those of a compiled language, and this introduces more details for you to worry about.
- Because multiple computers and multiple operating systems are involved, the user may not receive or understand the error messages produced.
- It is possible for messages to be sent or received not in the planned order.
- A *race condition* may occur in which the program results depend upon the specific ordering of messages. There is no guarantee that slave 1 will get its work done before slave 2, even though slave 1 may have started working earlier (Figure 14.9).
- Note in Figure 14.9 how different processors must wait for signals from other processors; this is clearly a waste of time and has potential for deadlock.
- Processes may *deadlock*, that is, wait for a message that never arrives.

14.13 EXAMPLE OF A SUPERCOMPUTER: IBM BLUE GENE/L

Whatever figures we give to describe the latest supercomputer will be obsolete by the time you read them. Nevertheless, for the sake of completeness and to set the scale for the present, we do it anyway. At the time of this writing, the fastest computer, in some aggregate sense, is the IBM Blue Gene series [Gara 05]. The name reflects its origin in a computer originally intended for gene research that is now sold as a general-purpose supercomputer (after approximately $600 million in development costs).

A building-block view of Blue Gene is given in Figure 14.10. In many ways this is a computer built by committee, with compromises made in order to balance cost, cooling, computing speed, use of existing technologies, communication speed, and so forth. As a case in point, the CPUs have dual cores, with one for computing and the other for communication. This reflects the importance of communication for distributed-memory computing (there are both on- and off-chip distributed memories). And while the CPU is fast at 5.6 GFLOPs, there are faster ones available, but they would generate so much heat that it would not be

Figure 14.10 The building blocks of Blue Gene (adapted from [Gara 05]).

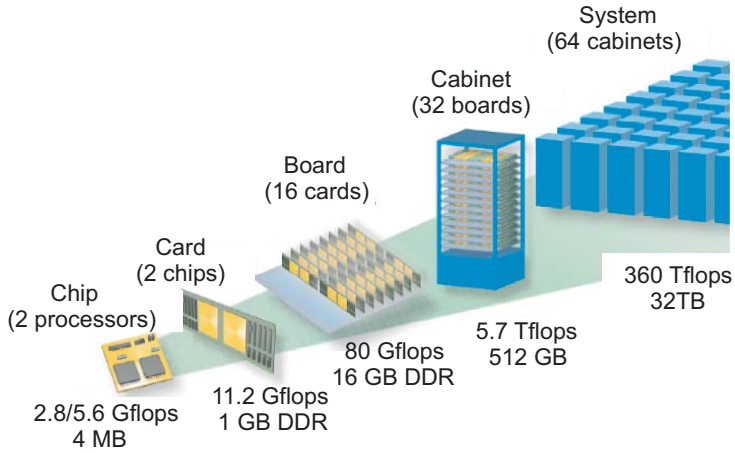
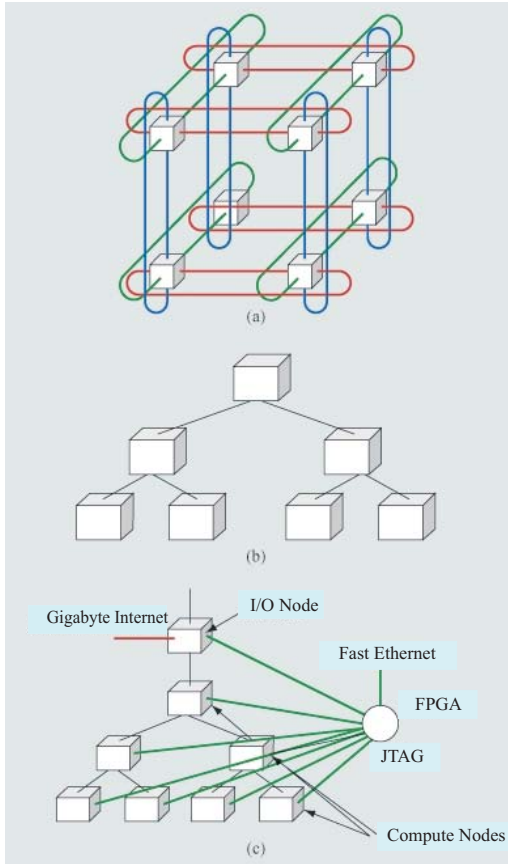


Figure 14.11 *(a)* A 3-D torus connecting $2 \times 2 \times 2$ memory blocks. *(b)* The global collective memory system. *(c)* The control and GB-Ethernet memory system (adapted from [Gara 05]).



possible to obtain the extreme scalability up to 2^{16} = 65,536 dual-processor nodes. The next objective is to balance a low cost/performance ratio with a high performance/ watt ratio.

Observe that at the lowest level Blue Gene contains two CPUs (dual cores) on a chip, with two chips on a card, with 16 cards on a board, with 32 boards in a cabinet, and up to 64 cabinets for a grand total of 65,536 CPUs (Figure 14.10). And if a way can be found to make all these chips work together for the common good on a single problem, they would turn out a peak performance of 360×10^{12} floating-point operations per second (360 tFLOPs). Each processor runs a Linux operating system (imagine what the cost in both time and money would be for Windows!) and utilizes the hardware by running a distributed memory MPI with C, C++, and Fortran90 compilers.

Blue Gene has three separate communication networks (Figure 14.11). At the heart of the memory system is a $64 \times 32 \times 32$ 3-D torus that connects all the memory blocks; Figure 14.11a shows the torus for $2 \times 2 \times 2$ memory blocks. The links are made by special link chips that also compute; they provide both direct neighbor–neighbor communications as well as cut through communication across the network. The result of this sophisticated network is that there is approximately the same effective bandwidth and latencies (response times) between all nodes, yet in order to obtain high speed, it is necessary to keep communication local. For node-to-node communication a rate of 1.4 Gb/s $= 1/10^{-9}$ s $= 1$ ns is obtained.

The latency ranges from 100 ns for 1 hop, to 6.4 μs for 64 hops between processors. The collective network in Figure 14.11b is used to communicate with all the processors simultaneously, what is known as a *broadcast*, and does so at 4 b/cycle. Finally, the control and gigabit ethernet network (Figure 14.11c) is used for I/O to communicate with the switch (the hardware communication center) and with ethernet devices. Its total capacity is greater than 1 tb ($= 10^{12}$)/s.

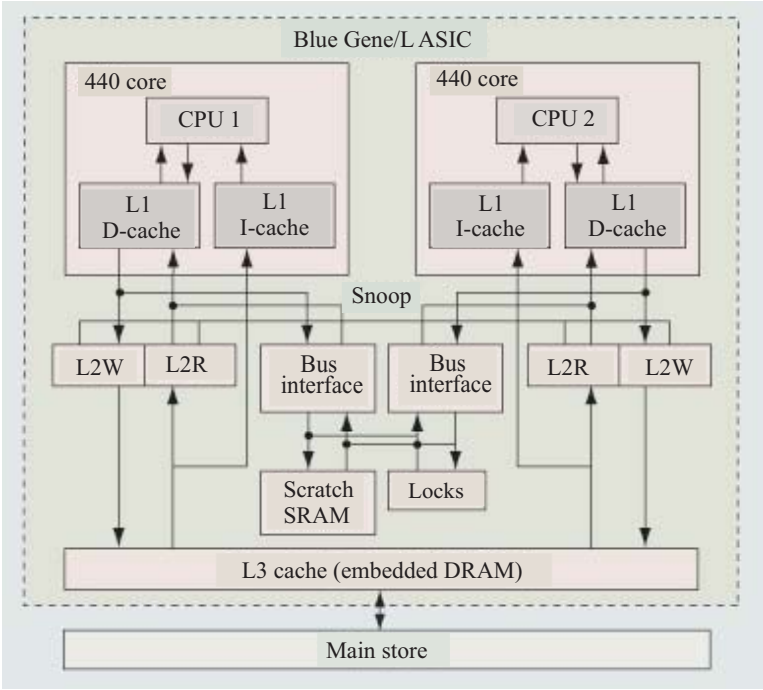
The computing heart of Blue Gene is its integrated circuit and the associated memory system (Figure 14.12). This is essentially an entire computer system on a chip containing, among other things,

- two PowerPC 440s with attached floating-point units (for rapid processing of floating-point numbers); one CPU is for computing, and one is for I/O.
- a RISC architecture CPU with seven stages, three pipelines, and 32-b, 64-way associative cache lines,
- variable memory page size,
- embedded dynamic memory controllers,
- a gigabit ethernet adapter,
- a total of 512 MB/node (32 tB when summed over nodes),
- level-1 (L1) cache of 32 KB, L2 cache of 2 KB, and L3 cache of 4 MB.

14.14 EXASCALE COMPUTING WITH MULTINODE-MULTICORES

The current architecture of top-end supercomputers uses a very large numbers of nodes, with each node containing a chip set that includes multiple cores (up to 32 at present) as well as a graphical processing unit (GPU) attached to the chip set (Figure 14.13). In the near future we expect to see laptop computers capable of teraflops (10^{12} floating point operations per second), deskside computers capable of petaflops, and supercomputers at the exascale in terms of both flops and memory.

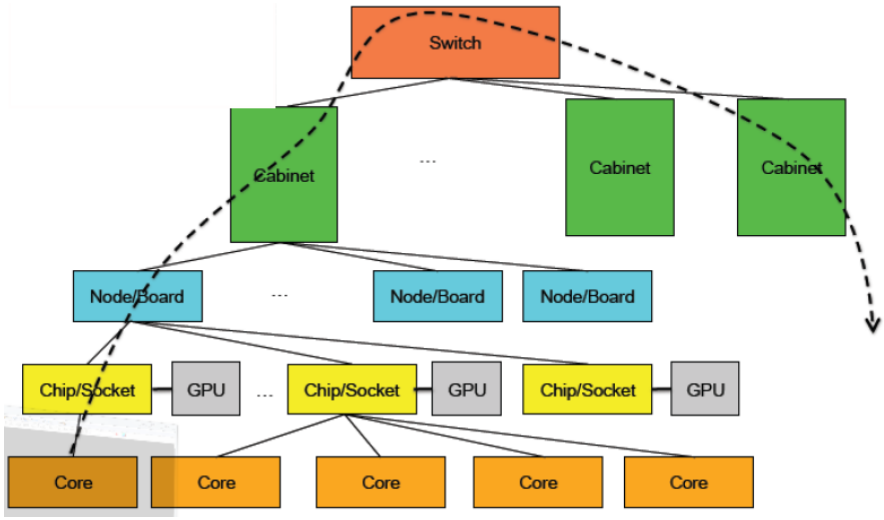
Figure 14.12 The single-node memory system (adapted from [Gara 05]).



Look again at the schematic in Figure 14.13. As in Blue Gene, there really are large numbers of chip boards and large numbers of cabinets. Here we show just one node and one cabinet and not the full number of cores. The dashed line in Figure 14.13 represents communications, and it is seen to permeate all components of the computer. Indeed, communications have become such an essential part of modern supercomputers, which may contain 100's of 1000's of CPUs, that the network interface "card" may be directly on the chip board. Because a computer of this sort contains shared memory at the node level and distributed memory at the cabinet or higher levels, programming for the requisite data transfer among the multiple elements is the essential challenge.

The GPU component in Figure 14.13 extends this type of supercomputer's architecture beyond that of the previously-discussed IBM Blue Gene. The GPU is an electronic circuit designed to accelerate the creation of visual images as they are displayed on a graphical output device. A GPU's efficiency arises from its ability to create many different parts of an image in parallel, an important ability since there are millions of pixels to display simultaneously. Indeed, these units can process 100s of millions of polygons in a second. Because GPUs are designed to assist the video processing on commodity devices such as personal computers, game machines, and mobile phones, they have become inexpensive, high performance, parallel computers in their own right. Yet because GPUs are designed to assist in video processing, their architecture and their programming are different from that of the general purpose CPUs usually used for scientific algorithms, and it takes some work to use them for scientific computing. Programming of GPUs requires specialized tools specific to the GPU architecture being used, and are beyond the discussion of this chapter. For example, CUDA programming refers to programming for the architecture developed by Nvidia, and is probably the most popular approach at present. Although possible to program at the basic level, there are now extensions and wrappers developed for popular programming languages such as C, Fortran, Python, Java and Perl. However, the general principles involved are just an extension of those used already discussed, and after we have you work through some examples of those general principles, in §14.17 we give some practical tips for programming multinode-multicore-GPU computers.

Figure 14.13 A schematic of an exascale computer in which, in addition to each chip having multiple cores, a graphical processing unit is attached to each chip. (Adapted from [Dong 11].)



14.15 UNIT III. HPC PROGRAM OPTIMIZATION

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 agree that using the optimization options of 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. Here are some entertaining, yet insightful, views [Har 96] reflecting the conclusions of those who have reflected upon this issue:

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.

— W.A. Wulf

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

— Donald Knuth

The best is the enemy of the good.

— Voltaire

Rule 1: Do not do it.

Rule 2 (for experts only): Do not do it yet.

Do not optimize as you go: Write your program without regard to possible optimizations, concentrating instead on making sure that the code is clean, correct, and understandable. If it's too big or too slow when you've finished, then you can consider optimizing it.

Remember the 80/20 rule: In many fields you can get 80% of the result with 20% of the effort (also called the 90/10 rule—it depends on who you talk to). Whenever you're about to optimize code, use profiling to find out where that 80% of execution time is going, so you know where to concentrate your effort.

Always run "before" and "after" benchmarks: How else will you know that your optimizations actually made a difference? If your optimized code turns out to be only slightly faster or smaller than the original version, undo your changes and go back to the original, clear code.

Use the right algorithms and data structures: Do not use an $\mathcal{O}(n^2)$ DFT algorithm to do a Fourier transform of a thousand elements when there's an $\mathcal{O}(n \log n)$ FFT available. Similarly, do not store a thousand items in an array that requires an $\mathcal{O}(n)$ search when you could use an $\mathcal{O}(\log n)$ binary tree or an $\mathcal{O}(1)$ hash table.

14.15.1 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 can't have the entire RAM at once, and so there will be memory access *conflicts*, in which case the performance of all the programs will suffer. The basic rules for programming for virtual memory are:

1. Do not waste your time worrying 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 their 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.

14.15.2 Optimizing Programs; Python *versus* Fortran/C

Many of the optimization techniques developed for Fortran and C are also relevant for Python applications. Yet while Python is a good language for scientific programming and is as universal and portable as Java, at present Python code runs slower than Fortran, C or even Java code. 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 Python not being designed for speed. Since modern computers are so fast, whether a program takes 1s or 3s 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 similar to that of Python) 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 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's 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 in accessing arrays in memory. They also are careful to keep 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 or Python cannot be compiled to produce an highly efficient code, and indeed such compilers are being developed and becoming available. However, such code is optimized for a particular computer architecture and so is not portable. In contrast, the byte code (`.class` file and `.pyc` in Python) produced is designed to be interpreted or recompiled by the *Java* or *Python Virtual Machine* (just another program). When you change from Unix to Windows, for example, the Java Virtual Machine program changes, but the byte code is the same. This is the essence of Java's portability.

In order to improve the performance of Java and Python, some computers and browsers run a *Just-in-Time* (JIT) compilers. If a JIT is present, the Virtual Machine feeds your byte code **`Prog.class`**, **`Prog.pyc`** to the JIT so that it can be recompiled into native code explicitly tailored to the machine you are using. Although there is an extra step involved here, the total time it takes to run your program is usually 10–30 times faster with a JIT as compared to line-by-line interpretation. Because the JIT is an integral part of the Virtual Machine on each operating system, this usually happens automatically.

In the experiments below you will investigate techniques to optimize both Fortran and Java or Python programs and to compare the speeds of both languages for the same computation. If you run your Java or Python code on a variety of machines (easy to do with them), you should also be able to compare the speed of one computer to that of another. Note that a knowledge of Fortran is not required for these exercises, you should be able to look at the code and figure out what changes may be needed.

Good and Bad Virtual Memory Use (Experiment)

To see the effect of using virtual memory, run these simple pseudocode examples on your computer (Listings 14.1–14.4). Use a command such as **`time`** to measure the time 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.

Listing 14.1 **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 (Listing 14.1) that each iteration of the **`for`** loop requires the data and code for all the functions as well as access to all the 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 it into separate components (Listing 14.2):

Listing 14.2 **GOOD program, separate loops.**

```
for j = 1, n; { for i = 1, n;        f12(i,j) = force12(pion(i), pion(j)) }
for j = 1, n; { for i = 1, n;        f21(i,j) = force21(pion(i), pion(j)) }
for j = 1, n; { for i = 1, n;        ftot = f12(i,j) + f21(i,j) }
```

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 (Listing 14.3):

Listing 14.3 **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 was tacked onto the end of **Common**. All the data comprising the variables **zed**, **ylt**, and **part** fit onto 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 B of data in **med2**. While it is difficult for a Fortran or C programmer to ensure the placement of variables within page boundaries, you will improve your chances by grouping data elements together (Listing 14.4):

Listing 14.4 **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
```

14.15.3 Empirical Performance of Hardware

In this section you conduct an experiment in which you run a complete program in several languages and on as many computers as are available. 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 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 *optimization options* to the compile command:

```
> f90 --O tune.f90
```

Here **--O** turns on optimization (**O** is the capital letter "oh," not zero). The actual optimization that is turned on differs from compiler to compiler. Fortran and C compilers have a bevy of such

options and directives that let you truly customize the resulting compiled code. Sometimes optimization options make the code run faster, sometimes not, and sometimes the faster-running code gives the wrong answers (but does so quickly).

Because computational scientists may spend a good fraction of their time running compiled codes, the compiler options tend to become 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.

The **Forte/Sun** Fortran compiler options include

`--O`	Use the 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 the same source file
`--O5`	Attempt aggressive optimizations (with profile feedback)

For the **Visual Fortran (Compaq, Intel)** compiler under windows, options are entered as **`/optimize`** and for optimization are

`/optimize:0`	Disable most optimizations
`/optimize:1`	Local optimizations in the source program unit
`/optimize:2`	Global optimization, including **`/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`	Activate loop transformation optimizations

The **gnu compilers** **`gcc, g77, g90`** accept **`--O`** options as well as

`--malign--double`	Align doubles on 64-bit boundaries
`--ffloat--store`	For codes using IEEE-854 extended precision
`--fforce--mem, --fforce--addr`	Improves loop optimization
`--fno--inline`	Do not compile statement functions inline
`--ffast--math`	Try non-IEEE handling of floats
`--funsafe--math--optimizations`	Speeds up float operations; 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`	Unrolls iterative **`DO`** loops
`--funroll--all--loops`	Unrolls **`DO WHILE`** loops

14.15.4 Python *versus* Fortran/C

The various versions of the program **tune** that are given in the Codes directory solve the matrix eigenvalue problem

$$\mathbf{Hc} = E\mathbf{c} \tag{14.11}$$

for the eigenvalues E and eigenvectors $\mathbf{c}$ of a Hamiltonian matrix $\mathbf{H}$. Here the individual Hamiltonian matrix elements are assigned the values

$$H_{i,j} = \begin{cases} i, & \text{for } i = j, \\ 0.3^{|i-j|}, & \text{for } i \neq j, \end{cases} = \begin{bmatrix} 1 & 0.3 & 0.14 & 0.027 & \dots \\ 0.3 & 2 & 0.3 & 0.9 & \dots \\ 0.14 & 0.3 & 3 & 0.3 & \dots \\ \ddots & & & & \end{bmatrix}. \tag{14.12}$$

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.

We find the solution to (14.11) via a variation of the *power* or *Davidson method*. We start with an arbitrary first guess for the eigenvector $\mathbf{c}$ and use it to calculate the energy corresponding to this eigenvector,[4]

$$\mathbf{c}_0 \simeq \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \qquad E \simeq \frac{\mathbf{c}_0^\dagger \mathbf{H} \mathbf{c_0}}{\mathbf{c}_0^\dagger \mathbf{c}_0}, \tag{14.13}$$

where $\mathbf{c}_0^\dagger$ is the row vector adjoint of $\mathbf{c}_0$. Because $\mathbf{H}$ is nearly diagonal with diagonal elements that increases as we move along the diagonal, this guess should be close to the eigenvector with the smallest eigenvalue. The heart of the algorithm is the guess that an improved eigenvector has the kth component

$$\mathbf{c}_1|_k \simeq \mathbf{c}_0|_k + \frac{[\mathbf{H} - E\mathbf{I}]\mathbf{c}_0|_k}{E - H_{k,k}}, \tag{14.14}$$

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 (14.14) each time. For the present case, six places of precision in the eigenvalue are usually obtained after 11 iterations. Here are the steps to follow:

- Vary the value of **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 (14.14) 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

[4] Note that the codes refer to the eigenvector c_0 as **coef**.

> **f90 tune.f90** — Fortran compilation

> **cc --lm tune.c** — C compilation, **gcc** also likely

> **time a.out** — 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**.

- As indicated in §14.15.3, you can ask the compiler to produce a version of your program optimized for speed by including the appropriate compiler option:

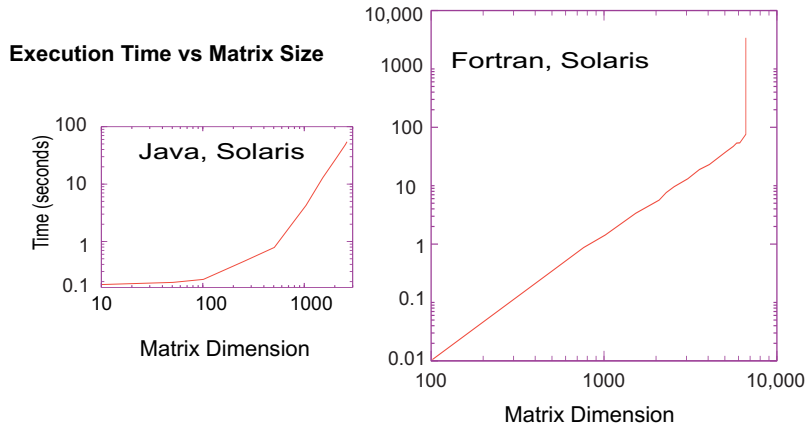
  > **f90 --O tune.f90**

  Execute and time the optimized code, checking that it still gives the same answer, and note any speedup 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.py** in Listing 14.5 is the Python equivalent of the Fortran program **tune.f90**.
- To get an idea of what **Tune.py** 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.py**. You do not have to issue the **time** command since we built a timer into the Python program. Check that you still get the same answer as you did with Fortran and note how much longer it takes with Python. You might be surprised how much slower Python is than Fortran.
- 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* to be sharing the computer with any other users. On Unix systems, the **who --a** command shows you the other users (we leave it up to you to figure out how to negotiate with them).
- To get some idea of what aspect of our little program is making it so slow, compile and run **Tune.py** for the series of matrix sizes **ldim** = 10, 100, 250, 500, 750, 1025, 2500, and 3000. You may get an error message that Python is out of memory at 3000. This is because you have not turned on the use of virtual memory.

Listing 14.5 **Tune.py** is meant to be numerically intensive enough 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.py Basic tuning program showing memory allocation

import datetime;  from numpy import zeros;  from math import (sqrt, pow)

Ldim = 251;          iter = 0;          step = 0.
diag  = zeros((Ldim, Ldim), float);     coef = zeros( (Ldim), float)
sigma = zeros((Ldim), float);           ham = zeros( (Ldim, Ldim), float)
t0 = datetime.datetime.now()                                  # Initialize time
for i in range(1, Ldim):                                    # Set up Hamiltonian
    for j in range(1, Ldim):
        if (abs(j - i) >10): ham[j, i] = 0.
        else : ham[j, i] = pow(0.3, abs(j - i) )
    ham[i,i] = i ;              coef[i] = 0.;
coef[1] = 1.;                   err = 1.;          iter = 0 ;
print("iter  ener      err ")
while (iter  < 15 and err > 1.e-6):        # Compute current E & normalize
```

Figure 14.14 Running time *versus* dimension for an eigenvalue search using `Tune.py` and `tune.f90`.




```
    iter = iter + 1; ener = 0. ; ovlp = 0.;
    for i in range(1, Ldim):
        ovlp = ovlp + coef[i]*coef[i]
        sigma[i] = 0.
        for j in range(1, Ldim): sigma[i] = sigma[i] + coef[j]*ham[j][i]
        ener = ener + coef[i]*sigma[i]
    ener = ener/ovlp
    for i in range(1, Ldim):
        coef[i] = coef[i]/sqrt(ovlp)
        sigma[i] = sigma[i]/sqrt(ovlp)
    err = 0.;
    for i in range(2, Ldim):                                    # Update
        step = (sigma[i] - ener*coef[i])/(ener - ham[i, i])
        coef[i] = coef[i] + step
        err = err + step*step
    err = sqrt(err)
    print(" %2d %9.7f %9.7f "%(iter, ener, err))
delta_t = datetime.datetime.now() - t0                    # Elapsed time
print(" time = ", delta_t)
```

- Make a graph of run time *versus* matrix size. It should be similar to Figure 14.14, although if there is more than one user on your computer while you run, you may get erratic results. Note that as our matrix becomes larger than $\sim 1000 \times 1000$ in size, the curve sharply increases in slope with execution time, in our case increasing like the *third* power of the dimension. Since the number of elements to compute increases as 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 2-D 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`** $= 10, 100, 250, 500, 1025, 3000, 4000, 6000, \ldots$. 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 a 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 2-D array **`ham [i][j]`** is slowing down the program, we have modified **`Tune.py`** into **`Tune4.py`** in Listing 14.6.
- Look at **`Tune4.py`** and note where the nested **`for`** loop over **`i`** and **`j`** now takes step of $\Delta i = 2$ rather the unit steps in **`Tune.py`**. If things work as expected, the better memory access of **`Tune4.py`** should cut the run time nearly in half. Compile and execute **`Tune4.py`**. Record the answer in your table.

Listing 14.6 **Tune4.py** does some loop unrolling by explicitly writing out two steps of a `for` loop (steps of 2.) This results in better memory access and faster execution.

```
# Tune4.py  Model tuning program

import datetime
from numpy import zeros
from math import (sqrt,pow)
from math import pi
from sys import version

if int(version[0])>2: raw_input=input   # raw_input deprecated in Python3

Ldim = 200;          iter1 = 0;          step = 0.
ham = zeros( (Ldim, Ldim), float);       diag = zeros( (Ldim), float)
coef = zeros( (Ldim), float);            sigma = zeros( (Ldim), float)
t0 = datetime.datetime.now()                              # Initialize time

for i in range(1, Ldim):                              # Set up Hamiltonian
        for j in range(1, Ldim):
            if abs(j - i) >10: ham[j, i] = 0.
            else : ham[j, i] = pow(0.3, abs(j - i) )
for i in range(1, Ldim):
    ham[i, i] = i
    coef[i]   = 0.
    diag[i]   = ham[i, i]
coef[1] = 1.;        err = 1.;           iter = 0 ;
print("iter      ener           err ")

while (iter1  < 15 and err > 1.e-6): # Compute current energy & normalize
    iter1 = iter1 + 1
    ener  = 0.
    ovlp1 = 0.
    ovlp2 = 0.
    for i in range(1, Ldim - 1, 2):
        ovlp1 = ovlp1 + coef[i]   * coef[i]
        ovlp2 = ovlp2 + coef[i+1] * coef[i+1]
        t1 = 0.
        t2 = 0.
        for j in range(1, Ldim):
            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./sqrt(ovlp)
    coef[1] = fact*coef[1]
    err = 0.                                          # Update & error norm
    for i in range(2, Ldim):
        t       = fact*coef[i]
        u       = fact*sigma[i] - ener*t
        step    = u/(ener - diag[i])
        coef[i] = t + step
        err     = err +  step*step
    err = sqrt(err)
    print (" %2d  %15.13f  %15.13f "%(iter1, ener, err))
delta_t = datetime.datetime.now()  -  t0                  # Elapsed time
print (" time = ",  delta_t)
print("press a key to finish")
s = raw_input()
```

- In order to cut the number of calls to the 2-D array in half, we employed a technique know as *loop unrolling* in which we explicitly wrote out some of the lines of code that 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 it evidently, 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.py`** *by hand*. Hand in your manual trial.

14.16 PROGRAMMING FOR THE 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. For 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 Figures 14.2 and 14.15, 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 the cache. If they are not, the manager reads the data from memory into the cache, and then the CPU deals with the data directly in the cache. The cache manager's view of RAM is shown in Figure 14.15.

When considering how a matrix operation uses memory, it is important to consider the *stride* of that operation, that is, the number of array elements that are stepped through as the operation repeats. For instance, summing the diagonal elements of a matrix to form the trace

$$\mathrm{Tr}\, A = \sum_{i=1}^{N} a(i,i) \tag{14.15}$$

involves a large stride because the diagonal elements are stored far apart for large N. However, the sum

$$c(i) = x(i) + x(i+1) \tag{14.16}$$

has stride 1 because adjacent elements of x are involved. The basic rule in programming for a cache is

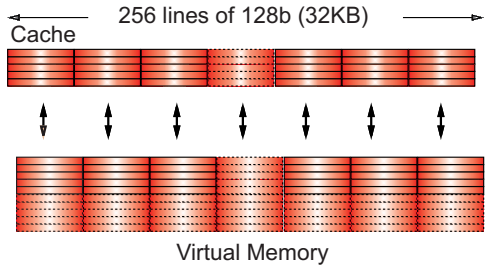
- Keep the stride low, preferably at 1, which in practice means:
- Vary the leftmost index first on Fortran arrays.
- Vary the rightmost index first on Python and C arrays.

14.16.1 Exercise 1: Cache Misses

We have said a number of times that your program will be slowed down if the data it needs are in virtual memory and not in RAM. Likewise, your program will also be slowed down if the data required by the CPU are not in the cache. For high-performance computing, you should write programs that keep as much of the data being processed as possible in the 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 Python 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 the cache from other elements of the computer's architecture, you should now estimate its importance by comparing the time it takes to step through the matrix elements row by row to the time it takes to step through the matrix elements column by column.

By actually running on machines available to you, check that the two simple codes in Listing 14.7 with the same number of arithmetic operations take significantly different times to

Figure 14.15 The cache manager's view of RAM. Each 128-b cache line is read into one of four lines in cache.



run because one of them must make large jumps through memory with the memory locations addressed not yet read into the cache:

```
    x(j) = m(1,j)                    // Sequential column reference
```

Listing 14.7 **Sequential column and row references.**

```
for j = 1, 9999;
    x(j) = m(j,1)                    // Sequential row reference
```

14.16.2 Exercise 2: Cache Flow

Test the importance of cache flow on your machine by comparing the time it takes to run the two simple programs in Listings 14.8 and 14.9. 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.

Listing 14.8 **GOOD f90, BAD Python/C Program; minimum, maximum stride.**

```
Dimension Vec(idim,jdim)                          // Stride 1 fetch (f90)
        for j = 1, jdim; { for i=1, idim;     Ans = Ans + Vec(i,j)*Vec(i,j)}
```

Listing 14.9 **BAD f90, GOOD Python/C Program; maximum, minimum stride.**

```
Dimension Vec(idim, jdim)                         // Stride jdim fetch (f90)
        for i = 1, idim; {for j=1, jdim;   Ans = Ans + Vec(i,j)*Vec(i,j)}
```

Loop A steps through the matrix **Vec** in column order. Loop B steps through in row order. By changing the size of the columns (the rightmost Fortran index), we change the step size (*stride*) taken through memory. Both loops take us through all the elements of the matrix, but the stride is different. By increasing the stride in any language, we use fewer elements already present in the cache, require additional swapping and loading of the cache, and thereby slow down the whole process.

14.16.3 Exercise 3: Large-Matrix Multiplication

As you increase the dimensions 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:

$$[C] = [A] \times [B], \tag{14.17}$$

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \times b_{kj}. \tag{14.18}$$

Listing 14.10 **BAD f90, GOOD Python/C Program; maximum, minimum stride.**

```
for i = 1, N; {                                    // Row
    for j = 1, N; {                                // Column
    c(i,j) = 0.0                                   // Initialize
    for k = 1, N; {
        c(i,j) = c(i,j) + a(i,k)*b(k,j) }}}        // Accumulate
```

This involves all the concerns with different kinds of memory. The natural way to code (14.17) follows from the definition of matrix multiplication (14.18), that is, as a sum over a row of A times a column of B. Try out the two code in Listings 14.10 and 14.11 on your computer. In Fortran, the first code has B with stride 1, but C with stride N. This is corrected in the second code by performing the initialization in another loop. In Python 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!

Listing 14.11 **GOOD f90, BAD Python/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) }}}
```

14.17 PRACTICAL TIPS FOR MULTICORE, GPU PROGRAMMING

We have already described some basics elements of exascale computing in § 14.14. Some practical tips for programming multinode-multicore-GPU computers follow along the same lines as we have been discussing, but with an even greater emphasis on minimizing communication costs[5]. This means that the "faster" of two algorithms may be the one that takes more steps, but requires less communications. Because the effort in programming the GPU directly can be quite high, we suggest that you will let compiler extensions and wrappers deal with the detailed GPU programming as this generally makes the best use of an application programmer's time.

Exascale computers and computing are expected to be *distruptive technologies* in which previous models for computing and computers change in a rather drastic step. This is in contrast to the more innovative technology of just increasing the clock speed of the CPU, which was the practice until the power consumption and associated heat production imposed a roadblock. Accordingly we should expect that software and algorithms will be changing (and we will have to rewrite our applications), much as it did when supercomputers changed from large vector machines with proprietary CPUs to cluster computing using commodity CPUs and message passing. Here are some of the major points to consider:

Exacascale data movement is expensive The time for a floating point operation and for a data transfer can be similar, although if the transfer is not local, as Figures 14.11 and 14.13 show happens quite often, then communication will be the rate-limiting step.

[5]Much of the material in this section comes from talks we have heard by John Dongarra [Dong 11].

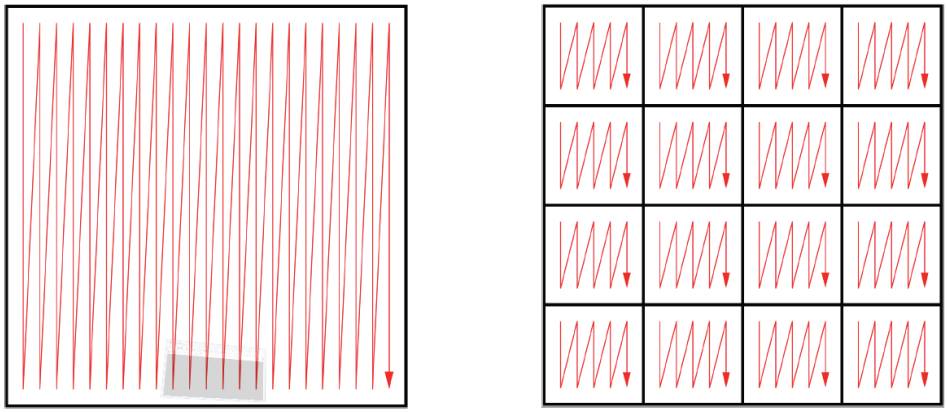


Figure 14.16 A schematic of how the contiguous elements of a matrix must be tiled for parallel processing (adapted from [Dong 11]).

Exacascale flop/s are cheap and abundant GPUs and local multicore chips provide many very fast flops for us to use, and we can expect even more of these elements to be included in future computer. So do not worry about flops as much as communication.

Synchronization-reducing algorithms are essential Having many processors stop in order to synchronize with each other, while essential in ensuring that the proper calculation is being performed, can slow down processing to a halt (literally). It is better to find or derive an algorithm that reduces the need for synchronization.

Break the fork-join model This model for parallel computing takes a queue of incoming jobs, divides them into subjobs for service on a variety of servers, and then combines them to produce the final result. Clearly, this type of model can lead to completed subjobs on various parts of the computer that must wait for other subjobs to complete before recombination. A big waste of resources.

Communication-reducing algorithms As already discussed, it is best to use methods that lessen the need for communication among processors, even if more flops are needed.

Use mixed precision methods Most GPUs do not have native double precision (or even full single precision) and correspondingly slow down by a factor-of-two or more when made to perform double precision calculation, or when made to move double-precision data. The preferred approach then is to use single precision. One way to do this is to employ perturbation theory in which the calculation focuses on the small (single precision) correction to a known, or separately calculated, large (double precision) basic solution. The rule-of-thumb then is to use the lowest precision required to achieve the required accuracy.

Push for and use autotuning The computer manufactures have advanced the hardware to incredible speeds, but have not produced concordant advances in the software that scientist need to employ in order to make good use of these machines. It often takes people-years to rewrite a program for these new machines, and that is a tremendous investment that not many scientists can afford. We need smarter software to deal with such complicated machines, and tools that permit us to optimize experimentally our programs for these machines.

Fault resilient algorithms Computers containing millions or billions of components do make mistakes at times. It does not make sense to have to start a calculation over, or hold a calculation up, when some minor failure such as a bit flip occurs. The algorithms should be able to recover from these types of failures.

Reproducibility of results Science is at its heart the search for scientific truth, and there should be only one answer when solving a problem whose solution is mathematically unique. However, approximations in the name of speed are sometimes made and this can lead to results whose exact reproducibility cannot be guaranteed. (Of course exact reproducibility is not to be expected for Monte Carlo calculation involving chance.)

Data layout is critical As we discussed with Figures 14.15, 14.2 and 14.16, much of HPC deals with matrices and their arrangements in memory. With parallel computing we must arrange the data into tiles such that each data tile is contiguous in memory. Then the tiles can be processed in a fine-grained computation. As we have seen in the exercises, the best size for the tiles depends upon the size of the caches being used, and these are generally small.

Chapter Fifteen

Thermodynamic Simulations & Feynman Quantum Path Integration

In Unit I of this chapter we describe how magnetic materials are simulated by using the Metropolis algorithm to solve the Ising model. This extends the techniques studied in Chapter 5, "Monte Carlo Simulations," to thermodynamics. Not only do thermodynamic simulations have important practical applications, but they also give us insight into what is "dynamic" in thermodynamics. In Unit II we describe a new Monte Carlo algorithm known as Wang–Landau sampling that in the last few years has shown itself to be far more efficient than the 50-year-old Metropolis algorithm. Wang–Landau sampling is an active subject in present research, and it is nice to see it fitting well into an elementary textbook. Unit III applies the Metropolis algorithm to Feynman's path integral formulation of quantum mechanics [F&H 65]. The theory, while the most advanced to be found in this book, forms the basis for field- theoretic calculations of quantum chromodynamics, some of the most fundamental and most time-consuming computations in existence. Basic discussions can be found in [Mann 83, MacK 85, M&N 87], with a recent review in [Potv 93].

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Ising Model Magnets	pdf	15.1-.6	Feynman Quantum Paths I	pdf	15.7
Feynman Quantum Paths II	pdf	15.8			

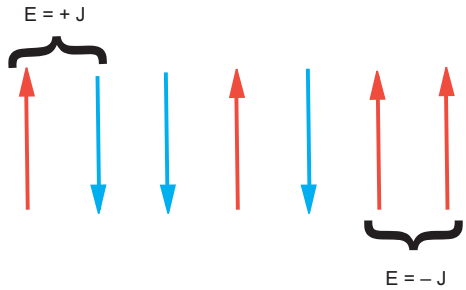
Applets (See Too Animations in Python Codes) Applet

Name	*Sections*	*Name*	*Sections*
Feynman Path Integrals	15.7–15.8		

15.1 UNIT I. MAGNETS VIA THE METROPOLIS ALGORITHM

Ferromagnets contain finite-size *domains* in which the spins of all the atoms point in the same direction. When an external magnetic field is applied to these materials, the different domains align and the materials become "magnetized." Yet as the temperature is raised, the total magnetism decreases, and at the Curie temperature the system goes through a *phase transition* beyond which all magnetization vanishes. Your **problem** is to explain the thermal behavior of ferromagnets.

Figure 15.1 The 1-D lattice of N spins used in the Ising model of magnetism. The interaction energy between nearest-neighbor pairs $E = \pm J$ is shown for aligned and opposing spins.



15.2 AN ISING CHAIN (MODEL)

As our model we consider N magnetic dipoles fixed in place on the links of a linear chain (Figure 15.1). (It is a straightforward generalization to handle 2-D and 3-D lattices.) Because the particles are fixed, their positions and momenta are not dynamic 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:

$$s_i \equiv s_{z,i} = \pm\frac{1}{2}. \tag{15.1}$$

Each configuration of the N particles is described by a quantum state vector

$$|\alpha_j\rangle = |s_1, s_2, \ldots, s_N\rangle = \left\{\pm\frac{1}{2},\ \pm\frac{1}{2},\ \ldots\right\}, \quad j = 1, \ldots, 2^N. \tag{15.2}$$

Because the spin of each particle can assume any one of *two* values, there are 2^N different possible states for the N particles in the system. Since fixed particles cannot be interchanged, we do not need to concern ourselves with the symmetry of the wave function.

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 magnetic *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:

$$V_i = -J\mathbf{s}_i \cdot \mathbf{s}_{i+1} - g\mu_b\, \mathbf{s}_i \cdot \mathbf{B}. \tag{15.3}$$

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 a particle's angular momentum and magnetic moment. The constant $\mu_b = e\hbar/(2m_e c)$ is the Bohr magneton, the basic measure for magnetic moments.

Even for small numbers of particles, the 2^N possible spin configurations gets to be very large ($2^{20} > 10^6$), and it is expensive for the computer to examine them all. Realistic samples with $\sim 10^{23}$ particles are beyond imagination. Consequently, statistical approaches are usually assumed, even for moderate values of N. Just how large N must be for this to be accurate is one of the things we want you to explore with your simulations.

The energy of this system in state α_k is the expectation value of the sum of the potential V over the spins of the particles:

$$E_{\alpha_k} = \left\langle \alpha_k \middle| \sum_i V_i \middle| \alpha_k \right\rangle = -J \sum_{i=1}^{N-1} s_i s_{i+1} - B\mu_b \sum_{i=1}^{N} s_i. \tag{15.4}$$

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 their spontaneous reversal. Indeed, natural magnetic materials have multiple finite domains with all the spins aligned, but with the different domains pointing in different directions. The instabilities in which domains change direction are called For simplicity we assume $B = 0$, which means that the spins interact just with each other. However, be cognizant of the fact that this means there is no preferred direction in space, and so you may have to be careful how you calculate observables. For example, you may need to take an absolute value of the total spin when calculating the magnetization, that is, to calculate $\langle|\sum_i s_i|\rangle$ rather than $\langle\sum_i s_i\rangle$.

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 the 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.

The simple 1-D Ising model has its limitations. Although the model is accurate in describing a system in thermal equilibrium, it is not accurate in describing the *approach* to thermal equilibrium (nonequilibrium thermodynamics is a difficult subject for which the theory is not complete). Second, as part of our algorithm we postulate that only one spin is flipped at a time, while real magnetic materials tend to flip many spins at a time. Other limitations are straightforward to improve, for example, the addition of longer-range interactions, the motion of the centers, higher-multiplicity spin states, and two and three dimensions.

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 1-D Ising model predicts realistic temperature dependences for the thermodynamic quantities, the model is too simple to support a phase transition. However, the 2-D and 3-D Ising models do support the Curie temperature phase transition [Yang 52].

15.3 STATISTICAL MECHANICS (THEORY)

Statistical mechanics starts with the elementary interactions among a system's particles and constructs the macroscopic thermodynamic properties such as specific heats. The essential assumption is that all configurations of the system consistent with the constraints are possible. In some simulations, such as the molecular dynamics ones in Chapter 16, "Simulating Matter with Molecular Dynamics," the problem is set up such that the *energy* of the system is fixed. The states of this type of system are described by what is called a *microcanonical ensemble*. In contrast, for the thermodynamic simulations we study in this chapter, the temperature, volume, and number of particles remain fixed, and so we have what is called a *canonical ensemble*.

When we say that an object is *at* temperature T, we mean that the object's atoms are in thermodynamic equilibrium at temperature T such that each atom has an average energy proportional to T. Although this may be an equilibrium state, it is a dynamic one in which the object's energy fluctuates as it exchanges energy with its environment (it is thermo*dynamics* after all). Indeed, one of the most illuminating aspects of the simulation we shall develop is its visualization of the continual and random interchange of energy that occurs at equilibrium.

The energy E_{α_j} of state α_j in a canonical ensemble is not constant but is distributed with probabilities $P(\alpha_j)$ given by the Boltzmann distribution:

$$\mathcal{P}(E_{\alpha_j}, T) = \frac{e^{-E_{\alpha_j}/k_B T}}{Z(T)}, \quad Z(T) = \sum_{\alpha_j} e^{-E_{\alpha_j}/k_B T}. \tag{15.5}$$

Here k is Boltzmann's constant, T is the temperature, and $Z(T)$ is the partition function, a weighted sum over states. Note that the sums in (15.5) are over the individual *states* or *configurations* of the system. Another formulation, such as the Wang–Landau algorithm in Unit II, sums over the *energies* of the states of the system and includes a density-of-states factor $g(E_i)$ to account for degenerate states with the same energy. While the present sum over states is a simpler way to express the problem (one less function), we shall see that the sum over energies is more efficient numerically. In fact, in this unit we even ignore the partition function $Z(T)$ because it cancels out when dealing with the *ratio* of probabilities.

15.3.1 Analytic Solutions

For very large numbers of particles, the thermodynamic properties of the 1-D Ising model can be solved analytically and determine [P&B 94]

$$U = \langle E \rangle \tag{15.6}$$

$$\frac{U}{J} = -N \tanh \frac{J}{k_B T} = -N \frac{e^{J/k_B T} - e^{-J/k_B T}}{e^{J/k_B T} + e^{-J/k_B T}} = \begin{cases} N, & k_B T \to 0, \\ 0, & k_B T \to \infty. \end{cases} \tag{15.7}$$

The analytic results for the specific heat per particle and the magnetization are

$$C(k_B T) = \frac{1}{N}\frac{dU}{dT} = \frac{(J/k_B T)^2}{\cosh^2(J/k_B T)} \tag{15.8}$$

$$M(k_B T) = \frac{N e^{J/k_B T} \sinh(B/k_B T)}{\sqrt{e^{2J/k_B T} \sinh^2(B/k_B T) + e^{-2J/k_B T}}}. \tag{15.9}$$

The **2-D Ising model** has an analytic solution, but it was not easy to derive [Yang 52, Huang 87]. Whereas the internal energy and heat capacity are expressed in terms of elliptic integrals, the spontaneous magnetization per particle has the rather simple form

$$\mathcal{M}(T) = \begin{cases} 0, & T > T_c \\ \frac{(1+z^2)^{1/4}(1-6z^2+z^4)^{1/8}}{\sqrt{1-z^2}}, & T < T_c, \end{cases} \tag{15.10}$$

$$kT_c \simeq 2.269185J, \quad z = e^{-2J/k_B T}, \tag{15.11}$$

where the temperature is measured in units of the Curie temperature T_c.

15.4 METROPOLIS ALGORITHM

In trying to devise an algorithm that simulates thermal equilibrium, it is important to understand that the Boltzmann distribution (15.5) does not require a system to remain in the state of lowest energy but says that it is less likely for the system to be found in a higher-energy state than in a lower-energy one. Of course, as $T \to 0$, only the lowest energy state will be populated.

For finite temperatures we expect the energy to fluctuate by approximately $k_B T$ about the equilibrium value.

In their simulation of neutron transmission through matter, Metropolis, Rosenbluth, Teller, and Teller [Metp 53] invented an algorithm to improve the Monte 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 §6.7.1 and the von Neumann rejection technique discussed in §6.7.3. There we showed how to make Monte Carlo integration more efficient by sampling random points predominantly where the integrand is large and how to generate random points with an arbitrary probability distribution. Now we would like to have spins flip randomly, have a system that can reach any energy in a finite number of steps (*ergodic* sampling), have a distribution of energies described by a Boltzmann distribution, yet have systems that equilibrate quickly enough to compute in reasonable times.

The Metropolis algorithm is implemented via a number of steps. We start with a fixed temperature and an initial spin configuration and apply the algorithm until a thermal equilibrium is reached (equilibration). Continued application of the algorithm generates the statistical fluctuations about equilibrium from which we deduce the thermodynamic quantities such as the magnetization $M(T)$. Then the temperature is changed, and the whole process is repeated in order to deduce the T dependence of the thermodynamic quantities. The accuracy of the deduced temperature dependences provides convincing evidence for the validity of the algorithm. Because the possible 2^N configurations of N particles can be a very large number, the amount of computer time needed can be very long. Typically, a small number of iterations $\simeq 10N$ is adequate for equilibration.

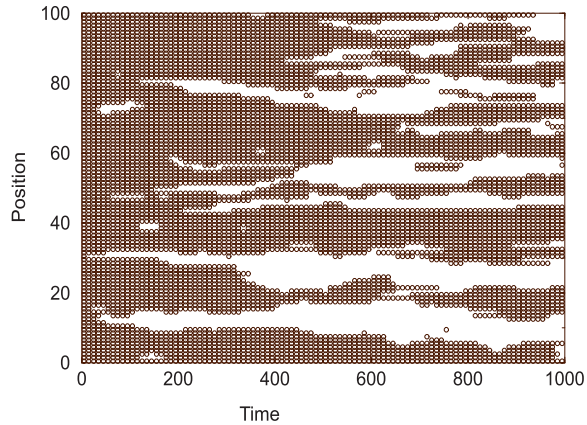
The explicit steps of the Metropolis algorithm are as follows.

1. Start with an arbitrary spin configuration $\alpha_k = \{s_1, s_2, \ldots, s_N\}$.
2. Generate a trial configuration α_{k+1} by
   a. picking a particle i randomly and
   b. flipping its spin.[1]
3. Calculate the energy $E_{\alpha_{\rm tr}}$ of the trial configuration.
4. If $E_{\alpha_{\rm tr}} \le E_{\alpha_k}$, accept the trial by setting $\alpha_{k+1} = \alpha_{\rm tr}$.
5. If $E_{\alpha_{\rm tr}} > E_{\alpha_k}$, accept with relative probability $\mathcal{R} = \exp(-\Delta E/k_B T)$:
   a. Choose a uniform random number $0 \le r_i \le 1$.
   b. Set $\alpha_{k+1} = \begin{cases} \alpha_{\rm tr}, & \text{if } \mathcal{R} \ge r_j \text{ (accept)}, \\ \alpha_k, & \text{if } \mathcal{R} < r_j \text{ (reject)}. \end{cases}$

The heart of this algorithm is its generation of a random spin configuration α_j (15.2) with

[1] Large-scale, practical computations make a full sweep in which every spin is updated once and then use this as the new trial configuration. This is found to be more efficient and useful in removing some autocorrelations. (We thank G. Schneider for this observation.)

Figure 15.2 An Ising model simulation using a 1-D 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.



probability

$$\mathcal{P}(E_{\alpha_j}, T) \propto e^{-E_{\alpha_j}/k_B T}. \tag{15.12}$$

The technique is a variation of von Neumann rejection (stone throwing in § 6.5) 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

$$\mathcal{R} = \frac{\mathcal{P}_{\text{tr}}}{\mathcal{P}_i} = e^{-\Delta E/k_B T}, \quad \Delta E = E_{\alpha_{\text{tr}}} - E_{\alpha_i}. \tag{15.13}$$

If the trial configuration has a lower energy ($\Delta E \leq 0$), the relative probability will be greater than 1 and we will accept the trial configuration as the new initial configuration without further ado. However, if the trial configuration has a higher energy ($\Delta E > 0$), we will not reject it out of hand but instead accept it with relative probability $\mathcal{R} = \exp(-\Delta E/k_B T) < 1$. 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 $E_{\alpha_{\text{tr}}} \to \infty$, in which case $\mathcal{P} \to 0$ and nothing is accepted.) When the trial configuration is rejected, the next configuration is identical to the preceding one.

How do you start? One possibility is to start with random values of the spins (a "hot" start). Another possibility (Figure 15.2) is a "cold" start in which you start with all spins parallel ($J > 0$) or antiparallel ($J < 0$). 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.

15.4.1 Metropolis Algorithm Implementation

1. Write a program that implements the Metropolis algorithm, that is, that produces a new configuration α_{k+1} from the present configuration α_k. (Alternatively, use the program `IsingViz.py` shown in Listing 15.1.)
2. Make the key data structure in your program an array `s[N]` containing the values of the spins s_i. For debugging, print out $+$ and $-$ to give the spin at each lattice point and

examine the pattern for different trial numbers.

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 $k_B T$ is in units of J and is the independent variable. Use $k_B T = 1$ 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); your simulation should resemble Figure 15.2.
   b. a random initial configuration (hot start).

15.4.2 Equilibration, Thermodynamic Properties (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.
2. Look for evidence of instabilities in which there is a spontaneous flipping of a large number of spins. This becomes more likely for larger $k_B T$ values.
3. Note how at thermal equilibrium the system is still quite dynamic, with spins flipping all the time. It is this energy exchange that determines the thermodynamic properties.
4. You may well find that simulations at small $k_B T$ (say, $k_B T \simeq 0.1$ for $N = 200$) are slow to equilibrate. Higher $k_B T$ 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 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.

Listing 15.1 **Ising.py** implements the Metropolis algorithm for a 1-D Ising chain.

```
# Ising.py: Ising model

import random
from visual.graph import *

# Display for the arrows
scene = display(x=0,y=0,width=700,height=200, range=40,title='Spins')
engraph = gdisplay(y=200,width=700,height=300, title='E of Spin System',\
        xtitle='iteration', ytitle='E',xmax=500, xmin=0, ymax=5, ymin=-5)
enplot = gcurve(color=color.yellow)                    # for the energy plot
N     = 30                                                 # number of spins
B     = 1.                                                  # magnetic field
mu    = .33                        # g mu (giromag. times Bohrs magneton)
J     = .20                                                # Exchange energy
k     = 1.                                               # Boltmann constant
T     = 100.                                                   # Temperature
state = zeros((N))                 # spins state some up(1) some down (0)
S     = zeros((N) ,float)
test  = state                                                  # a test state
random.seed()                                          # Seed random generator

def energy ( S) :                                      # Method to calc energy
    FirstTerm = 0.
    SecondTerm = 0.                                                # Sum energy
    for i in range(0,N-2):  FirstTerm += S[i]*S[i + 1]
```

```
    FirstTerm *= -J
    for i in range(0,N-1):   SecondTerm += S[i]
    SecondTerm *= -B*mu;
    return (FirstTerm + SecondTerm);

ES = energy(state)                                  # State, test's energy

def spstate(state):                         # Plots spins according to state
    for obj in scene.objects: obj.visible=0      #  erase previous arrows
    j=0
    for i in range(-N,N,2):                 # 30 spins numbered from 0 to 29
        if state[j]==-1:  ypos = 5                          # case spin down
        else:             ypos = 0
        if  5*state[j]<0: arrowcol = (1,1,1)   # white arrow if spin down
        else:             arrowcol =(0.7,0.8,0)
        arrow(pos=(i,ypos,0),axis=(0,5*state[j],0),color=arrowcol)# arrow
        j +=1

for  i in range(0 ,N):  state[i] = -1     # initial state, all spins down

for obj in scene.objects:   obj.visible=0
spstate(state)                        # plots initial state: all spins down
ES = energy(state)                    # finds the energy of the spin system
                                      # Here is the Metropolis algorithm
for  j in range (1,500):              # Change state and test
      rate(3)                         # to be able to see the flipping
      test = state                    # test is the previous spin state
      r = int(N*random.random());     # Flip spin randomly
      test[r] *= -1                   # flips temporarily that spin
      ET = energy(test)               # finds energy of the test configur.
      p = math.exp((ES-ET)/(k*T))     # test with Boltzmann factor
      enplot.plot(pos=(j,ES))         # adds a segment to the curve of E
      if p >= random.random():        # to see if trial config. is accepted
           state = test
           spstate(state)
           ES = ET
```

Thermodynamic Properties: For a given spin configuration α_j, the energy and magnetization are given by

$$E_{\alpha_j} = -J \sum_{i=1}^{N-1} s_i s_{i+1}, \quad \mathcal{M}_j = \sum_{i=1}^{N} s_i. \tag{15.14}$$

The internal energy $U(T)$ is just the average value of the energy,

$$U(T) = \langle E \rangle, \tag{15.15}$$

where the average is taken over a system in equilibrium. At high temperatures we expect a random assortment of spins and so a vanishing magnetization. At low temperatures when all the spins are aligned, we expect $\mathcal{M}$ to approach $N/2$. Although the specific heat can be computed from the elementary definition

$$C = \frac{1}{N}\frac{dU}{dT}, \tag{15.16}$$

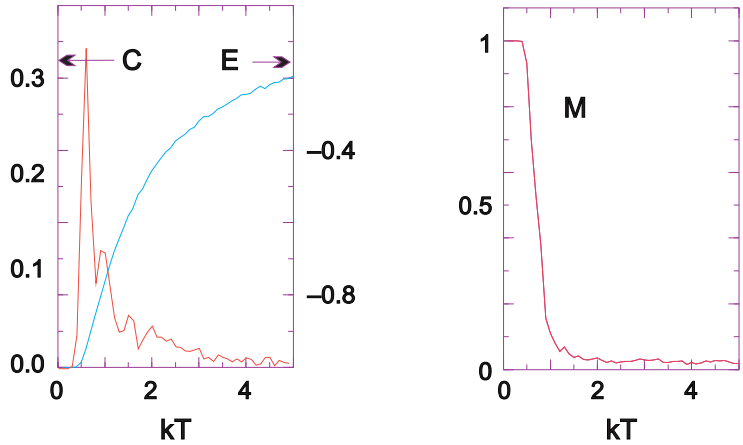
the numerical differentiation may be inaccurate since U has statistical fluctuations. A better way to calculate the specific heat is to first calculate the fluctuations in energy occurring during M trials and then determine the specific heat from the fluctuations:

$$U_2 = \frac{1}{M}\sum_{t=1}^{M}(E_t)^2, \tag{15.17}$$

$$C = \frac{1}{N^2}\frac{U_2 - (U)^2}{k_B T^2} = \frac{1}{N^2}\frac{\langle E^2\rangle - \langle E\rangle^2}{k_B T^2}. \tag{15.18}$$

1. Extend your program to calculate the internal energy U and the magnetization $\mathcal{M}$ for the chain. Do not recalculate entire sums when only one spin changes.

Figure 15.3 Simulation results from a 1-D Ising model of 100 spins for the energy, specific heat, and magnetization as a function of temperature.



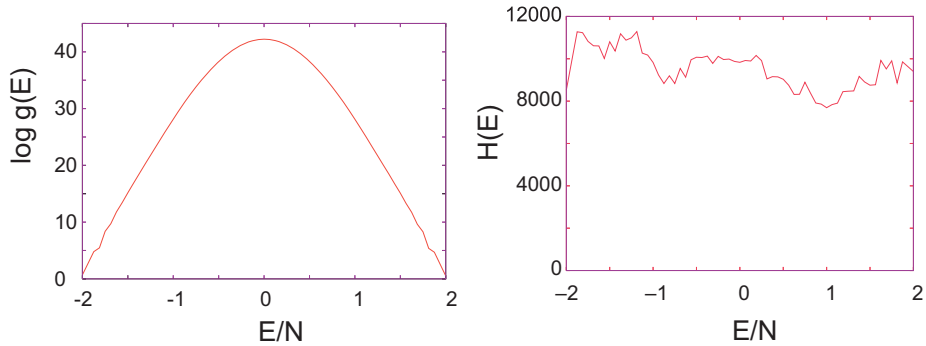
2. Make sure to wait for your system to equilibrate before you calculate thermodynamic quantities. (You can check that U is fluctuating about its average.) Your results should resemble Figure 15.3.
3. Reduce statistical fluctuations by running the simulation a number of times with different seeds and taking the average of the 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 $k_B T$ and compare it to the analytic result (15.7).
7. Make a plot of the magnetization $\mathcal{M}$ as a function of $k_B T$ and compare it to the analytic result. Does this agree with how you expect a heated magnet to behave?
8. Compute the energy fluctuations U_2 (15.17) and the specific heat C (15.18). Compare the simulated specific heat to the analytic result (15.8).

15.4.3 Beyond Nearest Neighbors, 1-D (Exploration)

- Extend the model so that the spin–spin interaction (15.3) 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 (15.3) extends to nearest neighbors in two dimensions, and for the truly ambitious, three dimensions. Continue using periodic boundary conditions and keep the number of particles small, at least to start [G,T&C 06].

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 that of the 1-D model?
4. Identify domains in the printout of spin configurations for small N.

Figure 15.4 Wang-Landau sampling used in the Ising model 2-D Ising model on an 8×8 lattice. *Left:* Logarithm of the density of states $\log g(E) \propto S$ *versus* the energy per particle. *Right:* The histogram $H(E)$ showing the number of states visited as a function of the energy per particle. The aim of WLS is to make this function flat.



5. Once your system appears to be behaving properly, calculate the heat capacity and magnetization of the 2-D Ising model with the same technique used for the 1-D model. Use a total number of particles of $100 \leq N \leq 2000$.
6. Look for a phase transition from ordered to unordered configurations by examining the heat capacity and magnetization as functions of temperature. The former should diverge, while the latter should vanish at the phase transition (Figure 15.5).

Exercise: Three fixed spin-$\frac{1}{2}$ particles interact with each other at temperature $T = 1/k_b$ such that the energy of the system is

$$E = -(s_1\, s_2 + s_2\, s_3).$$

The system starts in the configuration $\uparrow\downarrow\uparrow$. Do a simulation by hand that uses the Metropolis algorithm and the series of random numbers 0.5, 0.1, 0.9, 0.3 to determine the results of just two thermal fluctuations of these three spins. ▌

15.5 UNIT II. MAGNETS VIA WANG–LANDAU SAMPLING ⊙

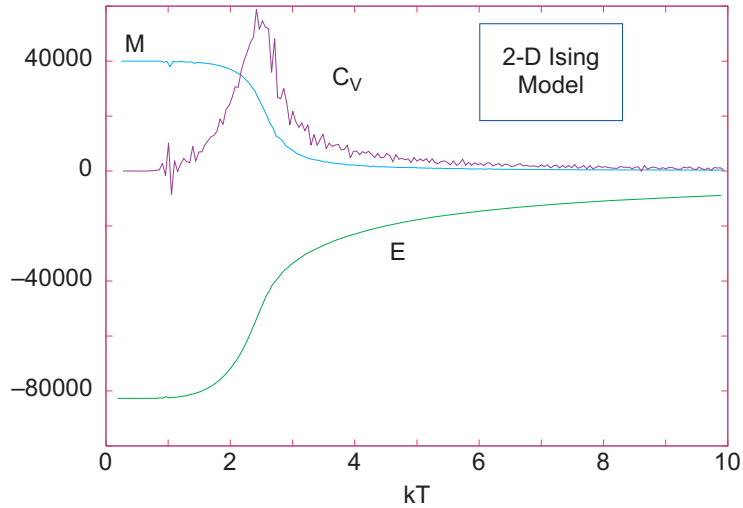
In Unit I we used a Boltzmann distribution to simulate the thermal properties of an Ising model. There we described the probabilities for explicit spin *states* α with energy E_α for a system at temperature T and summed over various configurations. An equivalent formulation describes the probability that the system will have the explicit *energy* E at temperature T:

$$\mathcal{P}(E_i, T) = g(E_i)\, \frac{e^{-E_i/k_B T}}{Z(T)}, \quad Z(T) = \sum_{E_i} g(E_i)\, e^{-E_i/k_B T}. \tag{15.19}$$

Here k_B is Boltzmann's constant, T is the temperature, $g(E_i)$ is the number of states of energy E_i ($i = 1, \ldots, M$), $Z(T)$ is the partition function, and the sum is still over all M states of the system but now with states of the same energy entering just once owing to $g(E_i)$ accounting for their degeneracy. Because we again apply the theory to the Ising model with its discrete spin states, the energy also assumes only discrete values. If the physical system had an energy that varied continuously, then the number of states in the interval $E \to E + dE$ would be given by $g(E)\, dE$ and $g(E)$ would be called the *density of states*. As a matter of convenience, we call $g(E_i)$ the density of states even when dealing with discrete systems, although the term "degeneracy factor" may be more precise.

Even as the Metropolis algorithm has been providing excellent service for more

Figure 15.5 The energy, specific heat, and magnetization as a function of temperature from a 2-D Ising model simulation with 40,000 spins. Evidence of a phase transition at the Curie temperature $kT = simeq2.5$ is seen in all three functions. The values of C and E have been scaled to fit on the same plot as M. (Courtesy of J. Wetzel.)



than 50 years, recent literature shows increasing use of Wang–Landau sampling (WLS) [WL 04, Clark]. Because WLS determines the density of states and the associated partition function, it is not a direct substitute for the Metropolis algorithm and its simulation of thermal fluctuations. However, we will see that WLS provides an equivalent simulation for the Ising model.[2] Nevertheless, there are cases where the Metropolis algorithm can be used but WLS cannot, computation of the ground-state function of the harmonic oscillator (Unit III) being an example.

The advantages of WLS is that it requires much shorter simulation times than the Metropolis algorithm and provides a direct determination of $g(E_i)$. For these reasons it has shown itself to be particularly useful for first-order phase transitions where systems spend long times trapped in metastable states, as well as in areas as diverse as spin systems, fluids, liquid crystals, polymers, and proteins. The time required for a simulation becomes crucial when large systems are modeled. Even a spin lattice as small as 8×8 has $2^{64} \simeq 1.84 \times 10^{19}$ configurations, and it would be too expensive to visit them all.

In Unit I we ignored the partition function when employing the Metropolis algorithm. Now we focus on the partition function $Z(T)$ and the density-of-states function $g(E)$. Because $g(E)$ is a function of energy but not temperature, once it has been deduced, $Z(T)$ and all thermodynamic quantities can be calculated from it without having to repeat the simulation for each temperature. For example, the internal energy and the entropy are calculated directly as

$$U(T) \stackrel{\text{def}}{=} \langle E \rangle = \frac{\sum_{E_i} E_i \, g(E_i) \, e^{-E_i/k_B T}}{\sum_{E_i} g(E_i) \, e^{-E_i/k_B T}}, \tag{15.20}$$

$$S = k_B \ln g(E_i). \tag{15.21}$$

The density of states $g(E_i)$ will be determined by taking the equivalent of a random walk in *energy* space. We flip a randomly chosen spin, record the energy of the new configuration, and then keep walking by flipping more spins to change the energy. The table $H(E_i)$ of the number of times each energy E_i is attained is called the energy *histogram* (an example of why it is called a histogram is given in Figure 15.4 on the right). If the walk were continued for

[2] We thank Oscar A. Restrepo of the Universidad de Antioquia for letting us use some of his material here.

a very long time, the histogram $H(E_i)$ would converge to the density of states $g(E_i)$. Yet with 10^{19}–10^{30} steps required even for small systems, this direct approach is unrealistically inefficient because the walk would rarely ever get away from the most probable energies.

Clever idea number 1 behind the Wang–Landau algorithm is to explore more of the energy space by increasing the likelihood of walking into less probable configurations. This is done by increasing the acceptance of less likely configurations while simultaneously decreasing the acceptance of more likely ones. In other words, we want to accept more states for which the density $g(E_i)$ is small and reject more states for which $g(E_i)$ is large (fret not these words, the equations are simple). To accomplish this trick, we accept a new energy E_i with a probability inversely proportional to the (initially unknown) density of states,

$$\mathcal{P}(E_i) = \frac{1}{g(E_i)}, \tag{15.22}$$

and then build up a histogram of visited states via a random walk.

The problem with clever idea number 1 is that $g(E_i)$ is unknown. WLS's clever idea 2 is to determine the unknown $g(E_i)$ simultaneously with the construction of the random walk. This is accomplished by improving the value of $g(E_i)$ via the multiplication $g(E_i) \to f\, g(E_i)$, where $f > 1$ is an empirical factor. When this works, the resulting histogram $H(E_i)$ becomes "flatter" because making the small $g(E_i)$ values larger makes it more likely to reach states with small $g(E_i)$ values. As the histogram gets flatter, we keep decreasing the multiplicative factor f until it is satisfactory close to 1. At that point we have a flat histogram and a determination of $g(E_i)$.

At this point you may be asking yourself, "Why does a flat histogram mean that we have determined $g(E_i)$?" Flat means that all energies are visited equally, in contrast to the peaked histogram that would be obtained normally without the $1/g(E_i)$ weighting factor. Thus, if by including this weighting factor we produce a flat histogram, then we have perfectly counteracted the actual peaking in $g(E_i)$, which means that we have arrived at the correct $g(E_i)$.

15.6 WANG–LANDAU SAMPLING

The steps in WLS are similar to those in the Metropolis algorithm, but now with use of the density-of-states function $g(E_i)$ rather than a Boltzmann factor:

1. Start with an arbitrary spin configuration $\alpha_k = \{s_1, s_2, \ldots, s_N\}$ and with arbitrary values for the density of states $g(E_i) = 1$, $i = 1, \ldots, M$, where $M = 2^N$ is the number of states of the system.
2. Generate a trial configuration α_{k+1} by
   a. picking a particle i randomly and
   b. flipping i's spin.
3. Calculate the energy $E_{\alpha_{\text{tr}}}$ of the trial configuration.
4. If $g(E_{\alpha_{\text{tr}}}) \le g(E_{\alpha_k})$, accept the trial, that is, set $\alpha_{k+1} = \alpha_{\text{tr}}$.
5. If $g(E_{\alpha_{\text{tr}}}) > g(E_{\alpha_k})$, accept the trial with probability $\mathcal{P} = g(E_{\alpha_k})/(g(E_{\alpha_{\text{tr}}})$:
   a. choose a uniform random number $0 \le r_i \le 1$.
   b. set $\alpha_{k+1} = \begin{cases} \alpha_{\text{tr}}, & \text{if } \mathcal{P} \ge r_j \text{ (accept)}, \\ \alpha_k, & \text{if } \mathcal{P} < r_j \text{ (reject)}. \end{cases}$

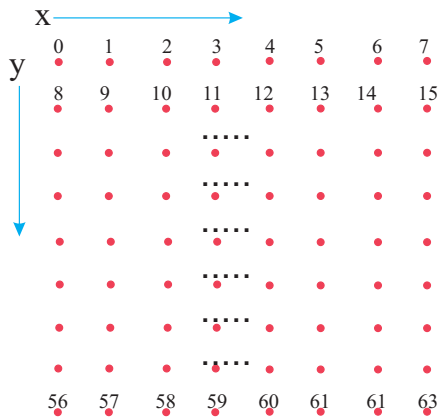
   This acceptance rule can be expressed succinctly as

$$\mathcal{P}(E_{\alpha_k} \to E_{\alpha_{\text{tr}}}) = \min\left[1, \frac{g(E_{\alpha_k})}{g(E_{\alpha_{\text{tr}}})}\right], \tag{15.23}$$

which manifestly always accepts low-density (improbable) states.

6. One we have a new state, we modify the current density of states $g(E_i)$ via the multiplicative factor f:
$$g(E_{\alpha_{k+1}}) \to f\, g(E_{\alpha_{k+1}}), \tag{15.24}$$
and add 1 to the bin in the histogram corresponding to the new energy:
$$H(E_{\alpha_{k+1}}) \to H(E_{\alpha_{k+1}}) + 1. \tag{15.25}$$
7. The value of the multiplier f is empirical. We start with Euler's number $f = e = 2.71828$, which appears to strike a good balance between very large numbers of small steps (small f) and too rapid a set of jumps through energy space (large f). Because the entropy $S = k_B \ln g(E_i) \to k_B[\ln\, g(E_i) + \ln\, f]$, (15.24) corresponds to a uniform increase by k_B in entropy.
8. Even with reasonable values for f, the repeated multiplications in (15.24) lead to exponential growth in the magnitude of g. This may cause floating-point overflows and a concordant loss of information [in the end, the magnitude of $g(E_i)$ does not matter since the function is normalized]. These overflows are avoided by working with logarithms of the function values, in which case the update of the density of states (15.24) becomes
$$\ln\, g(E_i) \to \ln\, g(E_i) + \ln\, f. \tag{15.26}$$
9. The difficulty with storing $\ln\, g(E_i)$ is that we need the ratio of $g(E_i)$ values to calculate the probability in (15.23). This is circumvented by employing the identity $x = \exp(\ln\, x)$ to express the ratio as
$$\frac{g(E_{\alpha_k})}{g(E_{\alpha_{\rm tr}})} = \exp\left[\ln \frac{g(E_{\alpha_k})}{g(E_{\alpha_{\rm tr}})}\right] = \exp\left[\ln\, g(E_{\alpha_k})\right] - \exp\left[\ln\, g(E_{\alpha_{\rm tr}})\right]. \tag{15.27}$$
In turn, $g(E_k) = f \times g(E_k)$ is modified to $\ln\, g(E_k) \to \ln\, g(E_k) + \ln f$.
10. The random walk in E_i continues until a flat histogram of visited energy values is obtained. The flatness of the histogram is tested regularly (every 10,000 iterations), and the walk is terminated once the histogram is sufficiently flat. The value of f is then reduced so that the next walk provides a better approximation to $g(E_i)$. Flatness is measured by comparing the variance in $H(E_i)$ to its average. Although 90%–95% flatness can be achieved for small problems like ours, we demand only 80% (Figure 15.4):
$$\text{If} \quad \frac{H_{\rm max} - H_{\rm min}}{H_{\rm max} + H_{\rm min}} < 0.2, \text{stop, let } \; f \to \sqrt{f}\, (\ln\, f \to \ln\, f/2). \tag{15.28}$$
11. Then keep the generated $g(E_i)$ and reset the histogram values $h(E_i)$ to zero.
12. The walks are terminated and new ones initiated until no significant correction to the density of states is obtained. This is measured by requiring the multiplicative factor $f \simeq 1$ within some level of tolerance; for example, $f \leq 1 + 10^{-8}$. If the algorithm is successful, the histogram should be flat (Figure 15.4) within the bounds set by (15.28).
13. The final step in the simulation is normalization of the deduced density of states $g(E_i)$. For the Ising model with N up or down spins, a normalization condition follows from knowledge of the total number of states [Clark]:
$$\sum_{E_i} g(E_i) = 2^N \quad \Rightarrow \quad g^{(\rm norm)}(E_i) = \frac{2^N}{\sum_{E_i} g(E_i)} g(E_i). \tag{15.29}$$
Because the sum in (15.29) is most affected by those values of energy where $g(E_i)$ is large, it may not be precise for the low-E_i densities that contribute little to the sum. Accordingly, a more precise normalization, at least if your simulation has done a good job in occupying all energy states, is to require that there are just two grounds states

Figure 15.6 The numbering scheme used in our WLS implementation of the 2-D Ising model with an 8×8 lattice of spins.



with energies $E = -2N$ (one with all spins up and one with all spins down):

$$\sum_{E_i=-2N} g(E_i) = 2. \tag{15.30}$$

In either case it is good practice to normalize $g(E_i)$ with one condition and then use the other as a check.

15.6.1 WLS Ising Model Implementation

We assume an Ising model with spin–spin interactions between nearest neighbors located in an $L \times L$ lattice (Figure 15.6). To keep the notation simple, we set $J = 1$ so that

$$E = -\sum_{i \leftrightarrow j}^{N} \sigma_i \sigma_j, \tag{15.31}$$

where $\leftrightarrow$ indicates nearest neighbors. Rather than recalculate the energy each time a spin is flipped, only the difference in energy is computed. For example, for eight spins in a 1-D array,

$$-E_k = \sigma_0\sigma_1 + \sigma_1\sigma_2 + \sigma_2\sigma_3 + \sigma_3\sigma_4 + \sigma_4\sigma_5 + \sigma_5\sigma_6 + \sigma_6\sigma_7 + \sigma_7\sigma_0, \tag{15.32}$$

where the 0–7 interaction arises because we assume periodic boundary conditions. If spin 5 is flipped, the new energy is

$$-E_{k+1} = \sigma_0\sigma_1 + \sigma_1\sigma_2 + \sigma_2\sigma_3 + \sigma_3\sigma_4 - \sigma_4\sigma_5 - \sigma_5\sigma_6 + \sigma_6\sigma_7 + \sigma_7\sigma_0, \tag{15.33}$$

and the difference in energy is

$$\Delta E = E_{k+1} - E_k = 2(\sigma_4 + \sigma_6)\sigma_5. \tag{15.34}$$

This is cheaper to compute than calculating and then subtracting two energies.

When we advance to two dimensions with the 8×8 lattice in Figure 15.6, the change in energy when spin $\sigma_{i,j}$ flips is

$$\Delta E = 2\sigma_{i,j}(\sigma_{i+1,j} + \sigma_{i-1,j} + \sigma_{i,j+1} + \sigma_{i,j-1}), \tag{15.35}$$

which can assume the values $-8, -4, 0, 4$, and 8. There are two states of minimum energy $-2N$ for a 2-D system with N spins, and they correspond to all spins pointing in the same

direction (either up or down). The maximum energy is $+2N$, and it corresponds to alternating spin directions on neighboring sites. Each spin flip on the lattice changes the energy by four units between these limits, and so the values of the energies are

$$E_i = -2N, \quad -2N+4, \quad -2N+8, \ldots, 2N-8, \quad 2N-4, \quad 2N. \tag{15.36}$$

These energies can be stored in a uniform 1-D array via the simple mapping

$$E' = (E+2N)/4 \quad \Rightarrow \quad E' = 0,\ 1,\ 2,\ \ldots,\ N. \tag{15.37}$$

Listing 15.2 displays our implementation of Wang–Landau sampling to calculate the density of states and internal energy $U(T)$ (15.20). We used it to obtain the entropy $S(T)$ and the energy histogram $H(E_i)$ illustrated in Figure 15.4. Other thermodynamic functions can be obtained by replacing the E in (15.20) with the appropriate variable. The results look like those in Figure 15.5. A problem that may be encountered when calculating these variables is that the sums in (15.20) can become large enough to cause overflows, even though the ratio would not. You work around that by factoring out a common large factor; for example,

$$\sum_{E_i} X(E_i)\, g(E_i)\, e^{-E_i/k_BT} = e^{\lambda} \sum_{E_i} X(E_i)\, e^{\ln g(E_i) - E_i/k_BT - \lambda}, \tag{15.38}$$

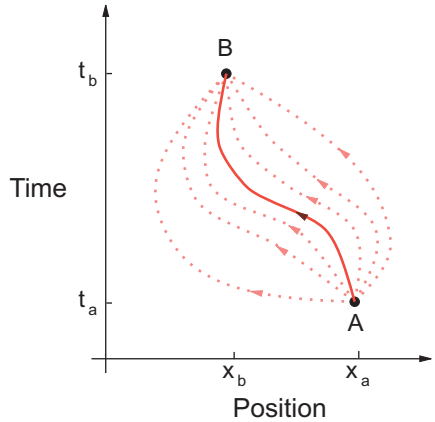
where λ is the largest value of $\ln\, g(E_i) - E_i/k_BT$ at each temperature. The factor e^{λ} does not actually need to be included in the calculation of the variable because it is present in both the numerator and denominator and so cancels out.

Listing 15.2 **WangLandau.py** simulates the 2-D Ising model using Wang–Landau sampling to compute the density of states and from that the internal energy.

```
# WangLandau.py: Wang Landau algorithm for 2-D spin system

""" Author in Java: Oscar A. Restrepo,
 Universidad de Antioquia, Medellin, Colombia
 Each time fac changes, a new histogrm is generated.
 Only the first Histogram plotted to reduce computational time"""

import random;   from visual.graph import *

L = 8;  N  = (L*L)

# Set up graphics
entgr = gdisplay(x=0,y=0,width=500,height=250,title='Density of States',\
      xtitle= 'E/N', ytitle='log g(E)', xmax=2., xmin=-2.,ymax=45,ymin=0)
entrp     = gcurve(color = color.yellow, display = entgr)
energygr = gdisplay(x=0, y=250, width=500, height=250, title='E vs T',\
        xtitle = 'T', ytitle='U(T)/N', xmax=8.,xmin=0, ymax =0.,ymin=-2.)
energ     = gcurve(color = color.cyan, display = energygr)
histogr  = display(x = 0, y = 500, width = 500, height = 300,\
       title = '1st histogram: H(E) vs. E/N, corresponds to log(f) = 1')
histo = curve(x = list(range(0, N+1)), color=color.red, display=histogr)
xaxis     = curve(pos = [( - N,  - 10), (N,  - 10)])
minE      = label(text = ' - 2', pos = ( - N + 3,  - 15), box = 0)
maxE      = label(text = '2', pos = (N - 3,  - 15), box = 0)
zeroE     = label(text = '0', pos = (0,  - 15), box = 0)
ticm      = curve(pos = [( - N,  - 10), ( - N,  - 13)])
tic0      = curve(pos = [(0,  - 10), (0,  - 13)])
ticM      = curve(pos = [(N,  - 10), (N,  - 13)])
enr       = label(text = 'E/N', pos = (N/2,  - 15), box = 0)

sp     = zeros( (L, L) )                              # Grid size, spins
hist   = zeros( (N + 1) )
prhist = zeros( (N + 1) )                                    # Histograms
S      = zeros( (N + 1), float)                      # Entropy = log g(E)

def iE(e):  return int((e + 2*N)/4)

def IntEnergy():
    exponent = 0.0
    for T in arange (0.2, 8.2, 0.2 ):                # Select lambda max
        Ener = - 2*N
        maxL = 0.0                                           # Initialize
        for i in range(0, N + 1):
            if S[i]!= 0 and (S[i] - Ener/T)>maxL:
```

```
                maxL = S[i] - Ener/T
                Ener = Ener + 4
        sumdeno = 0
        sumnume = 0
        Ener    = -2*N
        for i in range(0, N):
            if S[i] != 0:
                exponent = S[i] - Ener/T - maxL
            sumnume += Ener*exp(exponent)
            sumdeno += exp(exponent)
            Ener     = Ener +  4.0
        U = sumnume/sumdeno/N                    # internal energy U(T)/N
        energ.plot(pos = (T, U) )

def WL():                                        # Wang - Landau sampling
    Hinf   = 1.e10                          # initial values for Histogram
    Hsup   = 0.
    tol    = 1.e-3                       # tolerance, stops the algorithm
    ip     = zeros(L)
    im     = zeros(L)                             # BC R or down, L or up
    height = abs(Hsup - Hinf)/2.                   # Initialize histogram
    ave = (Hsup + Hinf)/2.                    # about average of histogram
    percent = height / ave
    for i in range(0, L):
        for j in range(0, L): sp[i, j] = 1                # Initial spins
    for i in range(0, L):
         ip[i] = i + 1
         im[i] = i - 1                                # Case plus, minus
    ip[L - 1] = 0
    im[0]     = L - 1                                          # Borders
    Eold = - 2*N                                     # Initialize energy
    for  j in range(0, N + 1): S[j] = 0             # Entropy initialized
    iter = 0
    fac  = 1
    while  fac > tol :
        i = int(N*random.random() )                  # Select random spin
        xg = i%L
        # Must be i//L, not i/L for Python 3:
        yg     = i//L                          # Localize x, y, grid point
        Enew   = Eold + 2*(sp[ip[xg],yg] + sp[im[xg],yg] + sp[xg,ip[yg]]
             + sp[xg, im[yg]] ) * sp[xg, yg]              # Change energy
        deltaS = S[iE(Enew)]  -  S[iE(Eold)]
        if  deltaS <= 0 or random.random() < exp( - deltaS):
           Eold = Enew;
           sp[xg, yg] *=  - 1                                # Flip spin
        S[iE(Eold)]   += fac;                             # Change entropy
        if iter%10000 == 0:           # Check flatness every 10000 sweeps
           for j in range( 0, N + 1):
              if  j == 0 :
                  Hsup = 0
                  Hinf = 1e10                   # Initialize new histogram
              if  hist[j] == 0 :  continue        # Energies never visited
              if  hist[j] > Hsup: Hsup = hist[j]
              if  hist[j] < Hinf: Hinf = hist[j]
           height = Hsup - Hinf
           ave = Hsup + Hinf
           percent = 1.0* height/ave        # 1.0 to make it float number
           if percent < 0.3 :                          # Histogram flat?
                print(" iter ", iter, "   log(f) ", fac)
                for j in range(0, N + 1):
                    prhist[j] = hist[j]                          # to plot
                    hist[j]   = 0                             # Save hist
                fac *= 0.5                    # Equivalent to log(sqrt(f))
        iter += 1
        hist[iE(Eold)]   += 1            # Change histogram, add 1, update
        if fac >= 0.5:                    # just show the first histogram
            # Speed up by using array calculations:
            histo.x = 2.0*arange(0,N+1) - N
            histo.y = 0.025*hist-10
deltaS = 0.0
print("wait because iter > 13 000 000")              # not always the same
WL()                                             # Call Wang Landau algorithm
deltaS = 0.0
for j in range(0, N + 1):
    order    = j*4  -  2*N
    deltaS   = S[j]  -  S[0] + log(2)
    if S[j] != 0 : entrp.plot(pos = (1.*order/N, deltaS))   # plot entropy
IntEnergy();
print("Done")
```

Figure 15.7 In the Feynman path-integral formulation of quantum mechanics a collection of paths connect 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.



15.6.2 WLS Ising Model Assessment

Repeat the assessment conducted in §15.4.2 for the thermodynamic properties of the Ising model but use WLS in place of the Metropolis algorithm.

15.7 UNIT III. FEYNMAN PATH INTEGRALS ⊙

Problem: As is well known, a classical particle attached to a linear spring undergoes simple harmonic motion with a position in space as a function of time given by $x(t) = A\sin(\omega_0 t + \phi)$. Your **problem** is to take this classical space-time trajectory $x(t)$ and use it to generate the quantum wave function $\psi(x,t)$ for a particle bound in a harmonic oscillator potential.

15.8 FEYNMAN'S SPACE-TIME PROPAGATION (THEORY)

Applet Feynman was looking for a formulation of quantum mechanics that gave a more direct connection to classical mechanics than does Schrödinger theory and that made the statistical nature of quantum mechanics evident from the start. He followed a suggestion by Dirac that Hamilton's principle of least action, which can be used to derive classical mechanics, may be the $\hbar \to 0$ limit of a quantum least-action principle. Seeing that Hamilton's principle deals with the paths of particles through space-time, Feynman posultated 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 expressed as [F&H 65]

$$\psi(x_b, t_b) = \int dx_a\, G(x_b, t_b; x_a, t_a)\psi(x_a, t_a), \tag{15.39}$$

where G is the *Green's function* or *propagator*

$$G(x_b, t_b; x_a, t_a) \equiv G(b, a) = \sqrt{\frac{m}{2\pi i(t_b - t_a)}}\exp\left[i\frac{m(x_b - x_a)^2}{2(t_b - t_a)}\right]. \tag{15.40}$$

Equation (15.39) 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 a new wavefront $\psi(x_b, t_b)$ is created by summation over and interference with all the other wavelets.

Feynman imagined that another way of interpreting (15.39) is as a form of Hamilton's principle in which the probability amplitude (wave function ψ) for a particle to be at B is equal to the sum over all *paths* through space-time originating at time A and ending at B (Figure 15.7). 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 can appreciate how different it is from Feynman's view.) The values for the probabilities of the paths derive from *Hamilton's classical principle of least action*:

> 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:
> $$\delta S[\bar{x}(t)] = S[\bar{x}(t) + \delta x(t)] - S[\bar{x}(t)] = 0, \tag{15.41}$$
> 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:

$$S[\bar{x}(t)] = \int_{t_a}^{t_b} dt\, L\left[x(t), \dot{x}(t)\right], \quad L = T\left[x, \dot{x}\right] - V[x]. \tag{15.42}$$

Here T is the kinetic energy, V is the potential energy, $\dot{x} = dx/dt$, and square brackets indicate a *functional*[3] of the function $x(t)$ and $\dot{x}(t)$.

Feynman observed that the classical action for a free particle ($V = 0$),

$$S[b, a] = \frac{m}{2}\left(\dot{x}\right)^2 (t_b - t_a) = \frac{m}{2}\frac{(x_b - x_a)^2}{t_b - t_a}, \tag{15.43}$$

is related to the free-particle propagator (15.40) by

$$G(b, a) = \sqrt{\frac{m}{2\pi i(t_b - t_a)}}\, e^{iS[b,a]/\hbar}. \tag{15.44}$$

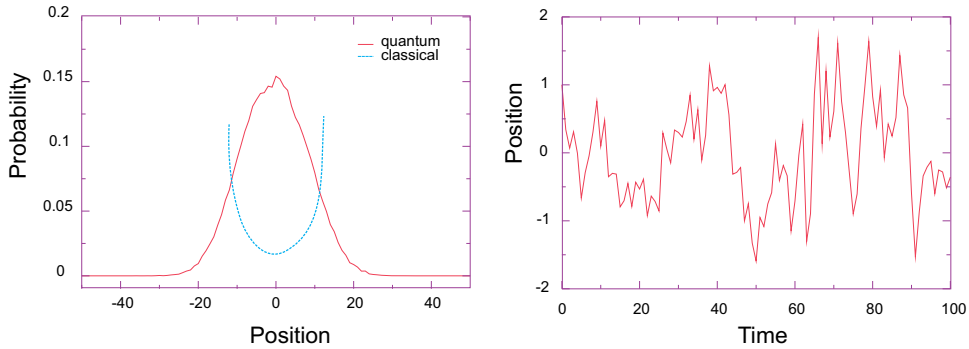
This is the much sought connection between quantum mechanics and Hamilton's principle. Feynman then postulated a reformulation of quantum mechanics that incorporated its statistical aspects by expressing $G(b, a)$ as the weighted sum over all *paths* connecting a to b,

$$G(b, a) = \sum_{\text{paths}} e^{iS[b,a]/\hbar} \quad \text{(path integral)}. \tag{15.45}$$

Here the classical action S (15.42) is evaluated along different paths (Figure 15.7), and the exponential of the action is summed over paths. The sum (15.45) 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 a sum are the same anyway). The essential connection between classical and quantum mechanics is the realization 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 (15.45), the main contributions come from those paths adjacent to the classical trajectory $\bar{x}$. In fact, because S is an extremum for the classical

[3] 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.

Figure 15.8 *Left:* The probability distribution for the harmonic oscillator ground state as determined with a path-integral calculation (the classical result has maxima at the two turning points). *Right:* A space-time quantum path resulting from applying the Metropolis algorithm.



trajectory, it is a constant to first order in the variation of paths, and so nearby paths have phases that vary smoothly and relatively slowly. In contrast, 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, $\hbar \to 0$, only the single classical trajectory contributes and (15.45) becomes Hamilton's principle of least action! In Figure 15.8 we show an example of a trajectory used in path-integral calculations.

15.8.1 Bound-State Wave Function (Theory)

Although you may be thinking that you have already seen enough expressions for the 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,

$$\tilde{H}\psi_n = E_n\psi_n,$$

each 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:

$$\psi(x,t) = \sum_{n=0}^{\infty} c_n\, e^{-iE_n t}\, \psi_n(x), \quad c_n = \int_{-\infty}^{+\infty} dx\, \psi_n^*(x)\psi(x,t=0), \tag{15.46}$$

where the value for the expansion coefficients c_n follows from the orthonormality of the ψ_n's. If we substitute this c_n back into the wave function expansion (15.46), we obtain the identity

$$\psi(x,t) = \int_{-\infty}^{+\infty} dx_0 \sum_n \psi_n^*(x_0)\psi_n(x)e^{-iE_n t}\psi(x_0,t=0). \tag{15.47}$$

Comparison with (15.39) yields the eigenfunction expansion for G:

$$G(x,t;x_0,t_0=0) = \sum_n \psi_n^*(x_0)\psi_n(x)e^{-iE_n t}. \tag{15.48}$$

We relate this to the bound-state wave function (recall that 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 (15.48) to negative imaginary time (permissable for

analytic functions):

$$G(x,-i\tau;x,0)=\sum_n |\psi_n(x)|^2 e^{-E_n\tau} = |\psi_0|^2 e^{-E_0\tau} + |\psi_1|^2 e^{-E_1\tau} + \cdots$$

$$\Rightarrow \quad |\psi_0(x)|^2 = \lim_{\tau\to\infty} e^{E_0\tau} G(x,-i\tau;x,0). \tag{15.49}$$

The limit here corresponds to long imaginary times τ, after which the parts of ψ with higher energies decay more quickly, leaving only the ground state ψ_0.

Equation (15.49) 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 Figure 15.8 showing some results of a computation. Although we start with a probability distribution that peaks near the classical turning points at the edges of the well, after a large number of iterations we end up with a distribution that resembles the expected Gaussian. On the right we see a trajectory that has been generated via statistical variations about the classical trajectory $x(t) = A\sin(\omega_0 t + \phi)$.

15.8.2 Lattice Path Integration (Algorithm)

Because both time and space are 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 (Figure 15.9). We divide the time between points A and B into N equal steps of size ε and label them with the index j:

$$\varepsilon \stackrel{\text{def}}{=} \frac{t_b - t_a}{N} \quad \Rightarrow \quad t_j = t_a + j\varepsilon \quad (j = 0, N). \tag{15.50}$$

Although it is more precise to use the actual positions $x(t_j)$ of the trajectory at the times t_j to determine the x_js (as in Figure 15.9), in practice we discretize space 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[4]:

$$\frac{dx_j}{dt} \simeq \frac{x_j - x_{j-1}}{t_j - t_{j-1}} = \frac{x_j - x_{j-1}}{\varepsilon}, \tag{15.51}$$

$$S_j \simeq L_j\,\Delta t \simeq \frac{1}{2} m \frac{(x_j - x_{j-1})^2}{\varepsilon} - V(x_j)\varepsilon, \tag{15.52}$$

where we have assumed that the Lagrangian is constant over each link.

Lattice path integration is based on the *composition theorem* for propagators:

$$G(b,a) = \int dx_j\, G(x_b,t_b;x_j,t_j) G(x_j,t_j;x_a,t_a) \quad (t_a < t_j,\ t_j < t_b). \tag{15.53}$$

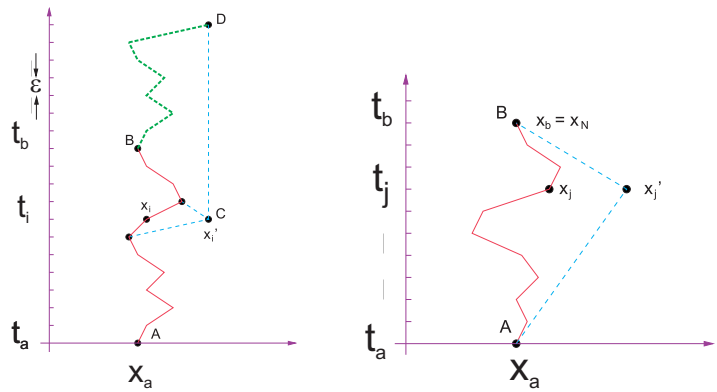
For a free particle this yields

$$G(b,a) = \sqrt{\frac{m}{2\pi i(t_b - t_j)}}\sqrt{\frac{m}{2\pi i(t_j - t_a)}} \int dx_j\, e^{i(S[b,j]+S[j,a])}$$

$$= \sqrt{\frac{m}{2\pi i(t_b - t_a)}} \int dx_j\, e^{iS[b,a]}, \tag{15.54}$$

[4]Even though Euler's rule has a large error, it is often use 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.

Figure 15.9 *Left:* A path through a space-time lattice that starts and ends at $x = x_a = x_b$. 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 path *AC*. *Right:* The dashed path joins the initial and final times in two equal time steps; the solid curve uses N steps each of size ε. The position of the curve at time t_j defines the position x_j.



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 in Figure 15.9, equation (15.53) becomes

$$G(b,a) = \int dx_1 \cdots dx_{N-1}\, e^{iS[b,a]}, \quad S[b,a] = \sum_{j=1}^{N} S_j, \tag{15.55}$$

where S_j is the value of the action for link j. At this point the integral over the *single* path shown in Figure 15.9 has become an N-term sum that becomes an infinite sum as the time step ε approaches zero.

To summarize, Feynman's path-integral postulate (15.45) means that we sum over all paths connecting A to B to obtain Green's function $G(b,a)$. This means that we must sum not only over the links in one path but *also* over all the different paths in order to produce the variation in paths 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 (15.45) is the sum over all paths connecting A to B, with each path weighted by the exponential of the action along that path, explicitly:

$$G(x,t;x_0,t_0) = \sum\!\!\!\!\!\!\int dx_1\, dx_2 \cdots\, dx_{N-1} e^{iS[x,x_0]}, \tag{15.56}$$

$$S[x,x_0] = \sum_{j=1}^{N-1} S[x_{j+1},x_j] \simeq \sum_{j=1}^{N-1} L\left(x_j,\dot{x_j}\right)\varepsilon, \tag{15.57}$$

where $L(x_j,\dot{x_j})$ is the average value of the Lagrangian on link j at time $t = j\varepsilon$. The computation is made simpler by assuming that the potential $V(x)$ is independent of velocity and does not depend on other x values (local potential). Next we observe that G is evaluated with a negative imaginary time in the expression (15.49) for the ground-state wave function. Accordingly,

we evaluate the Lagrangian with $t = -i\tau$:

$$L(x, \dot{x}) = T - V(x) = +\frac{1}{2}m\left(\frac{dx}{dt}\right)^2 - V(x), \tag{15.58}$$

$$\Rightarrow \quad L\left(x, \frac{dx}{-id\tau}\right) = -\frac{1}{2}m\left(\frac{dx}{d\tau}\right)^2 - V(x). \tag{15.59}$$

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 $t = \tau$:

$$H\left(x, \frac{dx}{d\tau}\right) = \frac{1}{2}m\left(\frac{dx}{d\tau}\right)^2 + V(x) = E, \tag{15.60}$$

$$\Rightarrow \quad L\left(x, \frac{dx}{-id\tau}\right) = -H\left(x, \frac{dx}{d\tau}\right). \tag{15.61}$$

In this way we rewrite the t-path integral of L as a τ-path integral of H and so express the action and Green's function in terms of the Hamiltonian:

$$S[j+1, j] = \int_{t_j}^{t_{j+1}} L(x,t)\, dt = -i\int_{\tau_j}^{\tau_{j+1}} H(x,\tau)\, d\tau, \tag{15.62}$$

$$\Rightarrow \quad G(x, -i\tau; x_0, 0) = \int dx_1 \ldots dx_{N-1}\, e^{-\int_0^\tau H(\tau')d\tau'}, \tag{15.63}$$

where the line integral of H is over an entire trajectory. Next we express the path integral in terms of the average energy of the particle on each link, $E_j = T_j + V_j$, and then sum over links[5] to obtain the summed energy $\mathcal{E}$:

$$\int H(\tau)\, d\tau \simeq \sum_j \varepsilon E_j = \varepsilon\mathcal{E}(\{x_j\}), \tag{15.64}$$

$$\mathcal{E}(\{x_j\}) \stackrel{\text{def}}{=} \sum_{j=1}^N \left[\frac{m}{2}\left(\frac{x_j - x_{j-1}}{\varepsilon}\right)^2 + V\left(\frac{x_j + x_{j-1}}{2}\right)\right]. \tag{15.65}$$

In (15.65) we have approximated each path link as a *straight line*, used Euler's derivative rule to obtain the velocity, and evaluated the potential at the midpoint of each link. We now substitute this G into our solution (15.49) for the ground-state wave function in which the initial and final points in space are the same:

$$\lim_{\tau\to\infty} \frac{G(x, -i\tau, x_0 = x, 0)}{\int dx\, G(x, -i\tau, x_0 = x, 0)} = \frac{\int dx_1 \cdots dx_{N-1} \exp\left[-\int_0^\tau H d\tau'\right]}{\int dx\, dx_1 \cdots dx_{N-1} \exp\left[-\int_0^\tau H d\tau'\right]}$$

$$\Rightarrow \quad |\psi_0(x)|^2 = \frac{1}{Z}\lim_{\tau\to\infty}\int dx_1 \cdots dx_{N-1}\, e^{-\varepsilon\mathcal{E}} \tag{15.66}$$

$$Z = \lim_{\tau\to\infty}\int dx\, dx_1 \cdots dx_{N-1} e^{-\varepsilon\mathcal{E}} \tag{15.67}$$

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 to the heat diffusion equation:

$$i\frac{\partial\psi}{\partial(-i\tau)} = \frac{-\nabla^2}{2m}\psi \quad \Rightarrow \quad \frac{\partial\psi}{\partial\tau} = \frac{\nabla^2}{2m}\psi. \tag{15.68}$$

[5] In some cases, such as for an infinite square well, this can cause problems if the trial link causes the energy to be infinite. In that case, one can modify the algorithm to use the potential at the beginning of a link

It is not surprising then that the sum over paths in Green's function has each path weighted by the Boltzmann factor $\mathcal{P} = e^{-\varepsilon\mathcal{E}}$ usually associated with thermodynamics. We make the connection complete by identifying the temperature with the inverse time step:

$$\mathcal{P} = e^{-\varepsilon\mathcal{E}} = e^{-\mathcal{E}/k_BT} \quad \Rightarrow \quad k_BT = \frac{1}{\varepsilon} \equiv \frac{\hbar}{\varepsilon}. \tag{15.69}$$

Consequently, the $\varepsilon \to 0$ limit, which makes time continuous, is a "high-temperature" limit. The $\tau \to \infty$ limit, which is required to project 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 Unit I required us to wait a long time while the system equilibrated, so the present simulation requires us to wait a long time so that all but the ground-state wave function has decayed. Alas, this is the solution to our **problem** of finding the ground-state wave function.

To summarize, we have expressed the Green's function as a path integral that requires integration of the Hamiltonian along paths and a summation over all the paths (15.66). We evaluate this path integral as the sum over all the trajectories in a space-time lattice. Each trial path occurs with a probability based on its action, and we use the Metropolis algorithm to include statistical fluctuation in the links, as if they are in thermal equilibrium. This is similar to our work with the Ising model in Unit I, 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 time the deduced wave function has to equilibrate to the ground state.

In general, Monte Carlo Green's function techniques work best if we start with a good guess at the correct answer and have the algorithm calculate variations on our guess. For the present problem this means that if we start 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 the weighting factor $\exp(\delta S/\hbar)$ fluctuates wildly (essentially averaging out to zero) and so loses its sensitivity.

A Time-Saving Trick

As we have formulated the computation, we pick a value of x and perform an expensive 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 must 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 (15.66), thereby fixing the initial position to be x_0, and then to integrate over all values for x_0:

$$|\psi_0(x)|^2 = \int dx_1 \cdots dx_N \, e^{-\varepsilon\mathcal{E}(x,x_1,\ldots)} \tag{15.70}$$

$$= \int dx_0 \cdots dx_N \delta(x - x_0) \, e^{-\varepsilon\mathcal{E}(x,x_1,\ldots)}. \tag{15.71}$$

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 (15.71), there will always be some x value for which the integral is nonzero, and we need to accumulate only the solution for various (discrete) x values to determine $|\psi_0(x)|^2$ for all x.

To understand how this works in practice, consider path AB in Figure 15.9 for which we have just calculated the summed energy. We 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 the wave function at that point.

15.8.3 Lattice Implementation

The program **QMC.py** in Listing 15.3 evaluates the integral (15.45) by finding the average of the integrand $\delta(x_0 - x)$ with paths distributed according to the weighting function $\exp[-\varepsilon\mathcal{E}(x_0, x_1, \ldots, x_N)]$. The physics enters via (15.73), the calculation of the summed energy $\mathcal{E}(x_0, x_1, \ldots, x_N)$. We evaluate the action integral for the harmonic oscillator potential

$$V(x) = \frac{1}{2}x^2 \tag{15.72}$$

and for a particle of mass $m = 1$. Using a convenient set of natural units, we 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 15.8 shows results from an application of the Metropolis algorithm. In this computation we started with an initial path close to the classical trajectory and then examined $\frac{1}{2}$ 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 maximum 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 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 $t_b - t_a$ equals the longer time $20T$, the system has had enough time to decay to its ground state and the wave function looks like the expected Gaussian distribution. In either case (Figure 15.8 right), 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 will be a very smooth trigonometric function (the classical trajectory), but the wave function, which is a measure of the fluctuations about the classical trajectory, will vanish! The explicit steps of the calculation are

1. Construct a grid of N time steps of length ε (Figure 15.9). 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 grid of M space points separated by steps of size δ. 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 τ_j, subject to the boundary conditions that the

initial and final positions always remain the same, $x_N = x_0 = x$.

5. Choose a path of straight-line links connecting the lattice points corresponding to the classical trajectory. Observe 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 $j = 0$:

$$\mathcal{E}(x_0, x_1, \ldots, x_N) \simeq \sum_{j=1}^{N} \left[\frac{m}{2} \left(\frac{x_j - x_{j-1}}{\varepsilon} \right)^2 + V\left(\frac{x_j + x_{j-1}}{2} \right) \right]. \tag{15.73}$$

7. Begin 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 Figure 15.9). This changes *two* links in the path.
8. For the coordinate that is changed, use the Metropolis algorithm to weigh the change with the Boltzmann factor.
9. For each lattice point, establish a running sum that represents 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 starting with a different seed. The average wave function from a number of intermediate-length runs is better than that from one very long run.

Listing 15.3 **QMC.py** determines the ground-state probability via a Feynman path integration using the Metropolis algorithm to simulate variations about the classical trajectory.

```
# QMC.py: Quantum MonteCarlo, Feynman path integration

import random
from visual.graph import *

N = 100;            M = 101;        xscale = 10.
path = zeros([M], float);           prob = zeros([M], float)   # Initialize

trajec = display(width = 300,height=500, title='Spacetime Trajectories')
trplot = curve(y = range(0, 100), color=color.magenta, display = trajec)

def trjaxs():                                                  # axis
   trax = curve(pos = [(-97,-100),(100,-100)], color = color.cyan, display = trajec)
   label(pos = (0,-110),  text = '0', box = 0, display = trajec)
   label(pos = (60,-110), text = 'x', box = 0, display = trajec)

wvgraph = display(x=340,y=150,width=500,height=300, title='Ground State')
wvplot = curve(x = range(0, 100), display = wvgraph)
wvfax = curve(color = color.cyan)

def wvfaxs():                                     # axis for probability
   wvfax = curve(pos =[(-600,-155),(800,-155)], display=wvgraph,color=color.cyan)
   curve(pos = [(0,-150), (0,400)], display=wvgraph, color=color.cyan)
   label(pos = (-80,450), text='Probability', box = 0, display = wvgraph)
   label(pos = (600,-220), text='x', box=0, display=wvgraph)
   label(pos = (0,-220), text='0', box=0, display=wvgraph)

trjaxs();           wvfaxs()                                   # plot axes

def energy(path):                                              # HO energy
    sums = 0.
    for i in range(0,N-2):sums += (path[i+1]-path[i])*(path[i+1]-path[i])
    sums += path[i+1]*path[i+1];
    return sums

def plotpath(path):                          # plot trajectory in xy scale
   for j in range (0, N):
       trplot.x[j] = 20*path[j]
       trplot.y[j] = 2*j - 100

def plotwvf(prob):                                             # plot prob
    for i in range (0, 100):
```

```
        wvplot.color = color.yellow
        wvplot.x[i] = 8*i - 400                    # convenient coordinates
        wvplot.y[i] = 4.0*prob[i] - 150               # for centered figure

oldE = energy(path)                                        # find E of path

while True:                                              # pick random element
    rate(10)                                             # slows the paintings
    element = int(N*random.random() )                   # Metropolis algorithm
    change = 2.0*(random.random() - 0.5)
    path[element] += change                                   # Change path
    newE = energy(path);                                       # Find new E
    if  newE > oldE and math.exp( - newE + oldE)<= random.random():
          path[element] -= change                                   # Reject
          plotpath(path)                    # plot resulting last trajectory
    elem = int(path[element]*16 + 50)        #     if path = 0, elem = 50

    # elem = m *path[element] + b is the linear transformation
    # if path=-3, elem=2 if path=3., elem=98 => b=50, m=16 linear TF.
    # this way x = 0 correspond to prob[50]

    if elem < 0: elem = 0,                       # negative case not allowed
    if elem > 100:  elem = 100                                # if exceed max
    prob[elem] += 1                  # increase probability for that x value
    plotwvf(prob)                                                 # plot prob
    oldE = newE
```

15.8.4 Assessment and Exploration

1. Plot some of the actual space-time paths used in the simulation along with 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 ε.
3. Because there are no sign changes in a ground-state wave function, you can ignore the phase, assume $\psi(x) = \sqrt{\psi^2(x)}$, and then estimate the energy via
$$E = \frac{\langle\psi|\, H\, |\psi\rangle}{\langle\psi|\psi\rangle} = \frac{\omega}{2\langle\psi|\psi\rangle}\int_{-\infty}^{+\infty}\psi^*(x)\left(-\frac{d^2}{dx^2}+x^2\right)\psi(x)\,dx,$$
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 your ψ for the gravitational potential (see quantum bouncer below):
$$V(x) = mg|x|, \quad x(t) = x_0 + v_0 t + \tfrac{1}{2}gt^2.$$

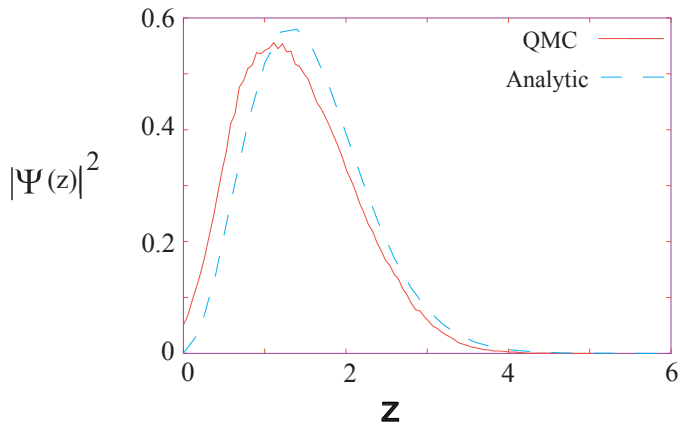
15.9 EXPLORATION: QUANTUM BOUNCER'S PATHS ⊙

Another problem for which the classical trajectory is well known is that of a *quantum bouncer*. Here we have a particle dropped in a uniform gravitational field, hitting a hard floor, and then bouncing. When treated quantum mechanically, quantized levels for the particle result [Gibbs 75, Good 92, Whine 92, Bana 99, Vall 00]. In 2002 an experiment to discern this gravitational effect at the quantum level was performed by Nesvizhevsky et al. [Nes 02] and described in [Schw 02]. It consisted of dropping ultracold neutrons from a height of 14 μm unto a neutron mirror and watching them bounce. It found a neutron ground state at 1.4 peV.

We start by determining the analytic solution to this problem for stationary states and

Figure 15.10 The analytic and quantum Monte Carlo solution for the quantum bouncer. The continuous line is the Airy function squared and the dashed line $|\psi_0(q)|^2$ () after a million trajectories.

{Slides}



then generalize it to include time dependence.[6] The time-independent Schrödinger equation for a particle in a uniform gravitation potential is

$$-\frac{\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + mxg\,\psi(x) = E\,\psi(x), \tag{15.74}$$

$$\psi(x \le 0) = 0, \quad \text{(boundary condition)}. \tag{15.75}$$

The boundary condition (15.75) is a consequence of the hard floor at $x = 0$. A change of variables converts (15.74) to a dimensionless form,

$$\frac{d^2\psi}{dz^2} - (z - z_E)\,\psi = 0, \tag{15.76}$$

$$z = x\left(\frac{2gm^2}{\hbar^2}\right)^{1/3}, \quad z_E = E\left(\frac{2}{\hbar^2 mg^2}\right)^{1/3}. \tag{15.77}$$

This equation has an analytic solution in terms of Airy functions Ai(z) [L 96]:

$$\psi(z) = N_n\ \mathrm{Ai}(z - z_E), \tag{15.78}$$

where N_n is a normalization constant and z_E is the scaled value of the energy. The boundary condition $\psi(0) = 0$ implies that

$$\psi(0) = N_E\ \mathrm{Ai}(-z_E) = 0, \tag{15.79}$$

which means that the allowed energies of the system are discrete and correspond to the zeros z_n of the Airy functions [Pres 00] at negative argument. To simplify the calculation, we take $\hbar = 1$, $g = 2$, and $m = \frac{1}{2}$, which leads to $z = x$ and $z_E = E$.

The time-dependent solution for the quantum bouncer is constructed by forming the infinite sum over all the discrete eigenstates, each with a time dependence appropriate to its

[6] Oscar A. Restrepo assisted in the preparation of this section.

energy:

$$\psi(z,t) = \sum_{n=1}^{\infty} C_n N_n \,\mathrm{Ai}(z - z_n) e^{-iE_n t/\hbar}, \tag{15.80}$$

where the C_n's are constants.

Figure 15.10 shows the results of solving for the quantum bouncer's ground-state probability $|\psi_0(z)|^2$ using Feynman's path integration. The time increment dt and the total time t were selected by trial and error in such a way as to make $|\psi(0)|^2 \simeq 0$ (the boundary condition). To account for the fact that the potential is infinite for negative x values, we selected trajectories that have positive x values over all their links. This incorporates the fact that the particle can never penetrate the floor. Our program is given in Listing 15.4, and it yields the results in Figure 15.10 after using 10^6 trajectories and a time step $\varepsilon = d\tau = 0.05$. Both results were normalized via a trapezoid integration. As can be seen, the agreement between the analytic result and the path integration is satisfactory.

Listing 15.4 **QMCbouncer.py** uses Feynman path integration to compute the path of a quantum particle in a gravitational field.

```
# QMCbouncer.py:        g.s. wavefunction via path integration

import random
from visual.graph import *

# Parameters
N = 100;  dt = 0.05;          g = 2.0;        h = 0.00;       maxel = 0
path = zeros([101], float);  arr = path; prob = zeros([201],float) # Init

trajec = display(width = 300, height=500,title = 'Spacetime Trajectory')
trplot = curve(y = range(0, 100), color=color.magenta, display = trajec)

def trjaxs():                                  # plot axis for trajectories
  trax=curve(pos=[(-97,-100),(100,-100)],color=color.cyan,display=trajec)
  curve(pos = [(-65, -100),(-65, 100)], color=color.cyan,display=trajec)
  label(pos = (-65,110), text = 't', box = 0, display = trajec)
  label(pos = (-85, -110), text = '0', box = 0, display = trajec)
  label(pos = (60, -110), text = 'x', box = 0, display = trajec)

wvgraph = display(x=350, y=80, width=500, height=300, title = 'GS Prob')
wvplot  = curve(x = range(0, 50), display = wvgraph) # wave function plot
wvfax   = curve(color = color.cyan)

def wvfaxs():                                  # plot axis for wavefunction
   wvfax = curve(pos =[(-200,-155),(800,-155)],display=wvgraph,color=color.cyan)
   curve(pos = [(-200,-150),(-200,400)],display=wvgraph,color=color.cyan)
   label(pos = (-70, 420),text = 'Probability', box = 0, display=wvgraph)
   label(pos = (600, -220),text = 'x', box = 0, display = wvgraph)
   label(pos = (-200, -220),text = '0', box = 0, display = wvgraph)

trjaxs();  wvfaxs()                                            # plot axes

def energy (arr):                              # Method for Energy of path
    esum = 0.
    for i in range(0,N):
      esum += 0.5*((arr[i+1]-arr[i])/dt)**2+g*(arr[i]+arr[i+1])/2
    return esum

def plotpath(path):                          # Method to plot xy trajectory
   for j in range (0, N):
       trplot.x[j] = 20*path[j] - 65
       trplot.y[j] = 2*j - 100

def plotwvf(prob):                           # Method to plot wave function
    for i in range (0, 50):
       wvplot.color = color.yellow
       wvplot.x[i] = 20*i - 200
       wvplot.y[i] = 0.5*prob[i] - 150

oldE = energy(path)                                            # Initial E
counter = 1                                          # for ea 100 iterations
norm = 0.                                          # wavefunction is plotted
maxx = 0.0

while 1:                                                   # "Infinite" loop
    rate(100)
```

```
    element = int(N*random.random() )
    if element != 0 and element!= N:                    # Ends not allowed
        change = ( (random.random() - 0.5)*20.)/10.  #  -1 =<random  =< 1
        if  path[element] + change > 0.:                # No negative paths
          path[element]   += change                    # change temporarily
          newE = energy(path)                           # New trajectory E
          if newE > oldE and exp( - newE + oldE) <= random.random() :
              path[element]   -= change                     # Link rejected
              plotpath(path)
          ele = int(path[element]*1250./100.)               # Scale changed
          if  ele >= maxel:  maxel = ele              # Scale change 0 to N
          if  element != 0:  prob[ele]   += 1
          oldE = newE;
    if counter%100 == 0:                        # plot wavefunction every 100
        for  i in range(0, N):                        # max x value of path
            if  path[i] >= maxx:  maxx = path[i]
        h = maxx/maxel                                          # space step
        firstlast = h*0.5*(prob[0] +  prob[maxel])   # for trap. extremes
        for  i in range(0, maxel + 1):  norm = norm + prob[i]      # norm
        norm = norm*h + firstlast                                # Trap rule
        plotwvf(prob)                                  # plot probability
    counter   += 1
```

Chapter Sixteen

Simulating Matter with Molecular Dynamics

Problem: Determine whether a collection of argon molecules placed in a box will form an ordered structure at low temperature.

You may have seen in your introductory classes that the ideal gas law can be derived from first principles if gas molecules are treated as billiard balls bouncing off the walls but not interacting with each other. We want to extend this model so that we can solve for the motion of every molecule in a box interacting with every other molecule via a potential. We picked argon because it is an inert element with a closed shell of electrons and so can be modeled as almost-hard spheres.

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Molecular Dynamics I	pdf	16	Molecular Dynamics II	pdf	16

Animation Links Applet

4 particles in 2D (gif)	25 particles in 2D (m4v)	100 particles in 2D (m4v)
Aquaporin H_2O permeation (mpg)	MD Applet by D. Wolf	

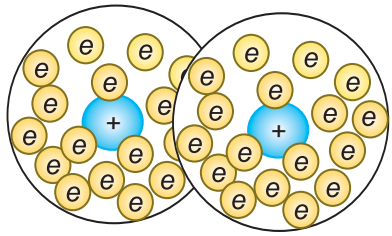
16.1 MOLECULAR DYNAMICS (THEORY)

Molecular dynamics (MD) is a powerful simulation technique for studying the physical and chemical properties of solids, liquids, amorphous materials, and biological molecules. Even though we know that quantum mechanics is the proper theory for molecular interactions, MD uses Newton's laws as the basis of the technique and focuses on bulk properties, which do not depend much on small-r behaviors. In 1985 Car and Parrinello showed how MD can be extended to include quantum mechanics by applying density functional theory to calculate the force [C&P 85]. This technique, known as *quantum MD*, is an active area of research but is beyond the realm of the present chapter.[1] For those with more interest in the subject, there are full texts [A&T 87, Rap 95, Hock 88] on MD and good discussions [G,T&C 06, Thij 99, Fos 96], as well as primers [Erco] and codes, [NAMD, Mold, ALCMD] available on-line.

MD's solution of Newton's laws is conceptually simple, yet when applied to a very large number of particles becomes the "high school physics problem from hell." Some approximations must be made in order not to have to solve the 10^{23}–10^{25} equations of motion describing a realistic sample but instead to limit the problem to $\sim 10^6$ particles for protein simulations and $\sim 10^8$ particles for materials simulations. If we have some success, then it is a good bet that the

[1] We thank Satoru S. Kano for pointing this out to us.

Figure 16.1 The molecule–molecule effective interaction arises from the many-body interaction of the electrons and nucleus in one molecule (circle) with the electrons and nucleus in another molecule (other circle). Note, the size of the nucleus at the center of each molecule is highly exaggerated is size, and real electrons are just points.



model will improve if we incorporate more particles or more quantum mechanics, something that becomes easier as computing power continues to increase.

In a number of ways, MD simulations are similar to the thermal Monte Carlo simulations we studied in Chapter 15, "Thermodynamic Simulations & Feynman Quantum Path Integration." Both typically involve a large number N of interacting particles that start out in some set configuration and then equilibrate into some dynamic state on the computer. However, in MD we have what statistical mechanics calls a *microcanonical ensemble* in which the energy E and volume V of the N particles are fixed. We then use Newton's laws to generate the dynamics of the system. In contrast, Monte Carlo simulations do not start with first principles but instead incorporate an element of chance and have the system in contact with a heat bath at a fixed temperature rather than keeping the energy E fixed. This is called a *canonical ensemble*.

Because a system of molecules is dynamic, the velocities and positions of the molecules change continuously, and so we will need to follow the motion of each molecule in time to determine its effect on the other molecules, which are also moving. After the simulation has run long enough to stabilize, we will compute time averages of the dynamic quantities in order to deduce the thermodynamic properties. We apply Newton's laws with the assumption that the net force on each molecule is the sum of the two-body forces with all other $(N-1)$ molecules:

$$m\frac{d^2\mathbf{r}_i}{dt^2} = \mathbf{F}_i(\mathbf{r}_0, \ldots, \mathbf{r}_{N-1}) \tag{16.1}$$

$$m\frac{d^2\mathbf{r}_i}{dt^2} = \sum_{i<j=0}^{N-1} \mathbf{f}_{ij}, \quad i = 0, \ldots, (N-1). \tag{16.2}$$

In writing these equations we have ignored the fact that the force between argon atoms really arises from the particle–particle interactions of the 18 electrons and the nucleus that constitute each atom (Figure 16.1). Although it may be possible to ignore this internal structure when deducing the long-range properties of inert elements, it matters for systems such as polyatomic molecules that display rotational, vibrational, and electronic degrees of freedom as the temperature is raised.[2]

We assume that the force on molecule i derives from a potential and that the potential is

[2]We thank Saturo Kano for clarifying this point.

Table 16.1 Parameter Values and Scales for the Lennard-Jones Potential.

Quantity	Mass	Length	Energy	Time	Temperature
Unit	m	σ	ϵ	$\sqrt{m\sigma^2/\epsilon}$	ϵ/k_B
Value	6.7×10^{-26} kg	3.4×10^{-10} m	1.65×10^{-21} J	4.5×10^{-12} s	119 K

the sum of central molecule–molecule potentials:

$$\mathbf{F}_i(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{N-1}) = -\nabla_{\mathbf{r}_i} U(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{N-1}), \tag{16.3}$$

$$U(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{N-1}) = \sum_{i<j} u(r_{ij}) = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} u(r_{ij}), \tag{16.4}$$

$$\Rightarrow \quad \mathbf{f}_{ij} = -\frac{du(r_{ij})}{dr_{ij}} \left(\frac{x_i - x_j}{r_{ij}} \hat{\mathbf{e}}_x + \frac{y_i - y_j}{r_{ij}} \hat{\mathbf{e}}_y + \frac{z_i - z_j}{r_{ij}} \hat{\mathbf{e}}_z \right).$$

Here $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = r_{ji}$ is the distance between the centers of molecules i and j, and the limits on the sums are such that no interaction is counted twice. Because we have assumed a *conservative* potential, the total energy of the system, that is, the potential plus kinetic energies summed over all particles, should be conserved over time. Nonetheless, in a practical computation we "cut the potential off" [assume $u(r_{ij}) = 0$] when the molecules are far apart. Because the derivative of the potential produces an infinite force at this cutoff point, energy will no longer be precisely conserved. Yet because the cutoff radius is large, the cutoff occurs only when the forces are minuscule, and so the violation of energy conservation should be small relative to approximation and round-off errors.

In a first-principles calculation, the potential between any two argon atoms arises from the sum over approximately 1000 electron–electron and electron–nucleus Coulomb interactions. A more practical calculation would derive an effective potential based on a form of many-body theory, such as Hartree–Fock or density functional theory. Our approach is simpler yet. We use the Lennard–Jones potential,

$$u(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{16.5}$$

$$\mathbf{f}(r) = -\frac{du}{dr}\frac{\mathbf{r}}{r} = \frac{48\epsilon}{r^2} \left[\left(\frac{\sigma}{r}\right)^{12} - \frac{1}{2}\left(\frac{\sigma}{r}\right)^{6} \right] \mathbf{r}. \tag{16.6}$$

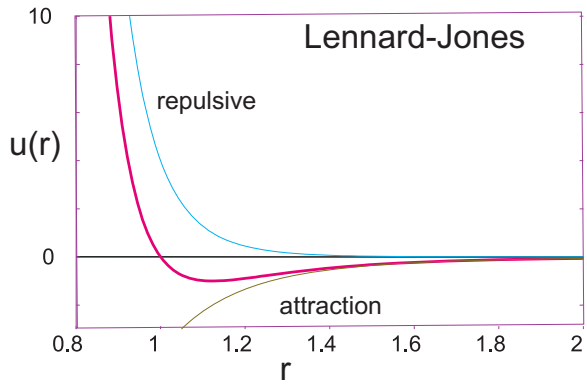
Here the parameter ϵ governs the strength of the interaction, and σ determines the length scale. Both are deduced by fits to data, which is why this is called a "phenomenological" potential.

Some typical values for the parameters, and corresponding scales for the variables, are given in Table 16.1. In order to make the program simpler and to avoid under- and overflows, it is helpful to measure all variables in the natural units formed by these constants. The interparticle potential and force then take the forms

$$u(r) = 4\left[\frac{1}{r^{12}} - \frac{1}{r^6}\right], \qquad f(r) = \frac{48}{r}\left[\frac{1}{r^{12}} - \frac{1}{2r^6}\right]. \tag{16.7}$$

The Lennard-Jones potential is seen in Figure 16.2 to be the sum of a long-range attractive interaction $\propto 1/r^6$ and a short-range repulsive one $\propto 1/r^{12}$. The change from repulsion to attraction occurs at $r = \sigma$. The minimum of the potential occurs at $r = 2^{1/6}\sigma = 1.1225\sigma$,

Figure 16.2 The Lennard-Jones effective potential used in many MD simulations. Note the sign change at $r = 1$ and the minimum at $r \simeq 1.1225$ (natural units). Note too that because the r axis does not extend to $r = 0$, the infinitely high central repulsion is not shown.



which would be the atom–atom spacing in a solid bound by this potential. The repulsive $1/r^{12}$ term in the Lennard-Jones potential (16.5) arises when the electron clouds from two atoms overlap, in which case the Coulomb interaction and the Pauli exclusion principle keep the electrons apart. The $1/r^{12}$ term dominates at short distances and makes atoms behave like hard spheres. The precise value of 12 is not of theoretical significance (although it's being large is) and was probably chosen because it is 2×6.

The $1/r^6$ term dominates at large distances and models the weak *van der Waals* induced dipole–dipole attraction between two molecules.[3] The attraction arises from fluctuations in which at some instant in time a molecule on the right tends to be more positive on the left side, like a dipole $\Leftarrow$. This in turn attracts the negative charge in a molecule on its left, thereby inducing a dipole $\Leftarrow$ in it. As long as the molecules stay close to each other, the polarities continue to fluctuate in synchronization $\Leftarrow\Leftarrow$ so that the attraction is maintained. The resultant dipole–dipole attraction behaves like $1/r^6$, and although much weaker than a Coulomb force, it is responsible for the binding of neutral, inert elements, such as argon for which the Coulomb force vanishes.

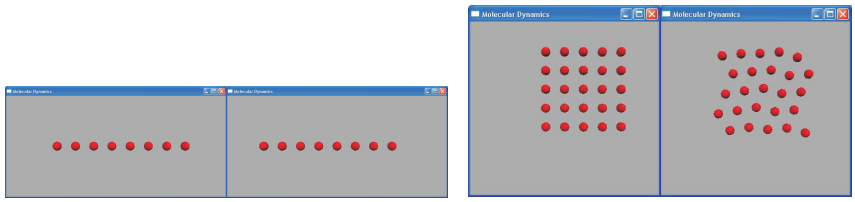
16.1.1 Connection to Thermodynamic Variables

We assume that the number of particles is large enough to use statistical mechanics to relate the results of our simulation to the thermodynamic quantities (the simulation is valid for any number of particles, but the use of statistics requires large numbers). The equipartition theorem tells us that for molecules in thermal equilibrium at temperature T, each molecular degree of freedom has an energy $k_B T/2$ on the average associated with it, where $k_B = 1.38 \times 10^{-23}$ J/K is Boltzmann's constant. A simulation provides the kinetic energy of translation[4]:

$$\mathrm{KE} = \frac{1}{2} \left\langle \sum_{i=0}^{N-1} v_i^2 \right\rangle . \tag{16.8}$$

[3] There are also van der Waals forces that cause dispersion, but we are not considering those here.
[4] Unless the temperature is very high, argon atoms, being inert spheres, have no rotational energy.

Figure 16.3 *Left:* Two frames from an animation showing the results of a 1-D MD simulation that starts with uniformly spaced atoms. Note how an image atom has moved in from the left after an atom leaves from the right. *Right:* Two frames from the animation of a 2-D MD simulation showing the initial and an equilibrated state. Note how the atoms start off in a simple cubic arrangement but then equilibrate to a face-centered-cubic lattice. In both cases the atoms remain confined due to the interatomic forces.



The time average of KE (three degrees of freedom) is related to temperature by

$$\langle \mathrm{KE} \rangle = N\frac{3}{2}k_B T \quad \Rightarrow \quad T = \frac{2\langle \mathrm{KE} \rangle}{3k_B N}. \tag{16.9}$$

The system's pressure P is determined by a version of the *Virial theorem*,

$$PV = Nk_B T + \frac{w}{3}, \quad w = \left\langle \sum_{i<j}^{N-1} \mathbf{r}_{ij} \cdot \mathbf{f}_{ij} \right\rangle, \tag{16.10}$$

where the Virial w is a weighted average of the forces. Note that because ideal gases have no interaction forces, their Virial vanishes and we have the ideal gas law. The pressure is thus

$$P = \frac{\rho}{3N}\left(2\langle \mathrm{KE} \rangle + w\right), \tag{16.11}$$

where $\rho = N/V$ is the density of the particles.

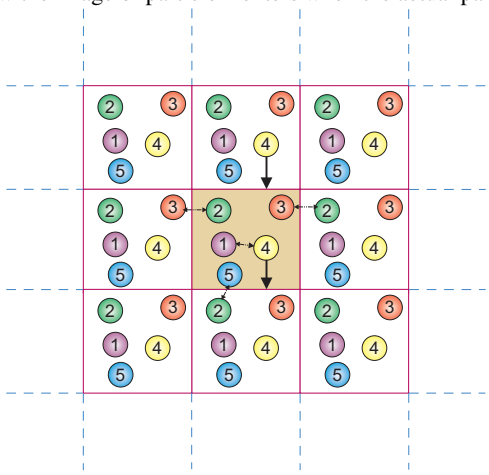
16.1.2 Setting Initial Velocities

Even though we start the system off with a velocity distribution characteristic of some temperature, since the system is not in equilibrium initially (some of the assigned KE goes into PE), this is not the true temperature of the system [Thij 99]. Note that this initial randomization is the only place where chance enters into our MD simulation, and it is there to speed the simulation along. Once started, the time evolution is determined by Newton's laws, in contrast to Monte Carlo simulations which are inherently stochastic. We produce a Gaussian (Maxwellian) velocity distribution with the methods discussed in Chapter 5, "Monte Carlo Simulations." In our sample code we take the average $\frac{1}{12}\sum_{i=1}^{12} r_i$ of uniform random numbers $0 \le r_i \le 1$ to produce a Gaussian distribution with mean $\langle r \rangle = 0.5$. We then subtract this mean value to obtain a distribution about 0.

16.1.3 Periodic Boundary Conditions and Potential Cutoff

It is easy to believe that a simulation of 10^{23} molecules should predict bulk properties well, but with typical MD simulations employing only 10^3–10^6 particles, one must be clever to make less seem like more. Furthermore, since computers are finite, the molecules in the simulation are constrained to lie within a finite box, which inevitably introduces artificial *surface effects* from the walls. Surface effects are particularly significant when the number of particles is small because then a large fraction of the molecules reside near the walls. For example, if 1000

Figure 16.4 The infinite space generated by imposing periodic boundary conditions on the particles within the simulation volume (shaded box). The two-headed arrows indicate how a particle interacts with the nearest version of another particle, be that within the simulation volume or an image. The vertical arrows indicate how the image of particle 4 enters when the actual particle 4 exits.



particles are arranged in a $10 \times 10 \times 10 \times 10$ cube, there are 10^3–$8^3 = 488$ particles one unit from the surface, that is, 49% of the molecules, while for 10^6 particles this fraction falls to 6%.

The imposition of *periodic boundary conditions* (PBCs) strives to minimize the shortcomings of both the small numbers of particles and of artificial boundaries. Even though we limit our simulation to an $L_x \times L_y \times L_z$ box, we imagine this box being replicated to infinity in all directions (Figure 16.4). Accordingly, after each time-integration step we examine the position of each particle and check if it has left the simulation region. If it has, then we bring an *image* of the particle back through the opposite boundary (Figure 16.4):

$$x \;\Rightarrow\; \begin{cases} x + L_x, & \text{if } x \le 0, \\ x - L_x, & \text{if } x > L_x. \end{cases} \tag{16.12}$$

Consequently, each box looks the same and has continuous properties at the edges. As shown by the one-headed arrows in Figure 16.4, if a particle exits the simulation volume, its image enters from the other side, and so balance is maintained.

In principle a molecule interacts with all others molecules and their images, so even though there is a finite number of atoms in the interaction volume, there is an effective infinite number of interactions [Erco]. Nonetheless, because the Lennard–Jones potential falls off so rapidly for large r, $V(r = 3\sigma) \simeq V(1.13\sigma)/200$, far-off molecules do not contribute significantly to the motion of a molecule, and we pick a value $r_{\text{cut}} \simeq 2.5\sigma$ beyond which we ignore the effect of the potential:

$$u(r) = \begin{cases} 4\left(r^{-12} - r^{-6}\right), & \text{for } r < r_{\text{cut}}, \\ 0, & \text{for } r > r_{\text{cut}}. \end{cases} \tag{16.13}$$

Accordingly, if the simulation region is large enough for $u(r > L_i/2) \simeq 0$, an atom interacts with only the *nearest image* of another atom.

The only problem with the cutoff potential (16.13) is that since the derivative du/dr is singular at $r = r_{\text{cut}}$, the potential is no longer conservative and thus energy conservation is no longer ensured. However, since the forces are already very small at r_{cut}, the violation will also be very small.

16.2 VERLET AND VELOCITY-VERLET ALGORITHMS

A realistic MD simulation may require integration of the 3-D equations of motion for 10^{10} time steps for each of 10^3–10^6 particles. Although we could use our standard **rk4** ODE solver for this, time is saved by using a simple rule embedded in the program. The Verlet algorithm uses the central-difference approximation (Chapter 7, "Differentiation & Searching") for the second derivative to advance the solutions by a single time step h for all N particles simultaneously:

$$\mathbf{F}_i[\mathbf{r}(t), t] = \frac{d^2\mathbf{r}_i}{dt^2} \simeq \frac{\mathbf{r}_i(t+h) + \mathbf{r}_i(t-h) - 2\mathbf{r}_i(t)}{h^2}, \tag{16.14}$$

$$\Rightarrow \quad \mathbf{r}_i(t+h) \simeq 2\mathbf{r}_i(t) - \mathbf{r}_i(t-h) + h^2\mathbf{F}_i(t) + \mathrm{O}(h^4), \tag{16.15}$$

where we have set $m = 1$. (Improved algorithms may vary the time step depending upon the speed of the particle.) Notice that even though the atom–atom force does not have an explicit time dependence, we include a t dependence in it as a way of indicating its dependence upon the atoms' positions at a particular time. Because this is really an implicit time dependence, energy remains conserved.

Part of the efficiency of the Verlet algorithm (16.15) is that it solves for the position of each particle without requiring a separate solution for the particle's velocity. However, once we have deduced the position for various times, we can use the central-difference approximation for the first derivative of $\mathbf{r}_i$ to obtain the velocity:

$$\mathbf{v}_i(t) = \frac{d\mathbf{r}_i}{dt} \simeq \frac{\mathbf{r}_i(t+h) - \mathbf{r}_i(t-h)}{2h} + \mathrm{O}(h^2). \tag{16.16}$$

Note, finally, that because the Verlet algorithm needs $\mathbf{r}$ from two previous steps, it is not self-starting and so we start it with the forward difference,

$$\mathbf{r}(t = -h) \simeq \mathbf{r}(0) - h\mathbf{v}(0) + \frac{h^2}{2}\mathbf{F}(0). \tag{16.17}$$

Velocity-Verlet Algorithm: Another version of the Verlet algorithm, which we recommend because of its increased stability, uses a forward-difference approximation for the derivative to advance *both* the position and velocity simultaneously:

$$\mathbf{r}_i(t+h) \simeq \mathbf{r}_i(t) + h\mathbf{v}_i(t) + \frac{h^2}{2}\mathbf{F}_i(t) + \mathrm{O}(h^3), \tag{16.18}$$

$$\mathbf{v}_i(t+h) \simeq \mathbf{v}_i(t) + h\,\overline{\mathbf{a}(t)} + \mathrm{O}(h^2) \tag{16.19}$$

$$\simeq \mathbf{v}_i(t) + h\left[\frac{\mathbf{F}_i(t+h) + \mathbf{F}_i(t)}{2}\right] + \mathrm{O}(h^2). \tag{16.20}$$

Although this algorithm appears to be of lower order than (16.15), the use of updated positions when calculating velocities, and the subsequent use of these velocities, make both algorithms of similar precision.

Of interest is that (16.20) approximates the average force during a time step as $[\mathbf{F}_i(t+h) + \mathbf{F}_i(t)]/2$. Updating the velocity is a little tricky because we need the force at time $t+h$, which depends on the particle positions at $t+h$. Consequently, we must update all the particle positions and forces to $t+h$ before we update any velocities, while saving the forces at the earlier time for use in (16.20). As soon as the positions are updated, we impose periodic boundary conditions to ensure that we have not lost any particles, and then we calculate the forces.

16.3 1-D IMPLEMENTATION AND EXERCISE

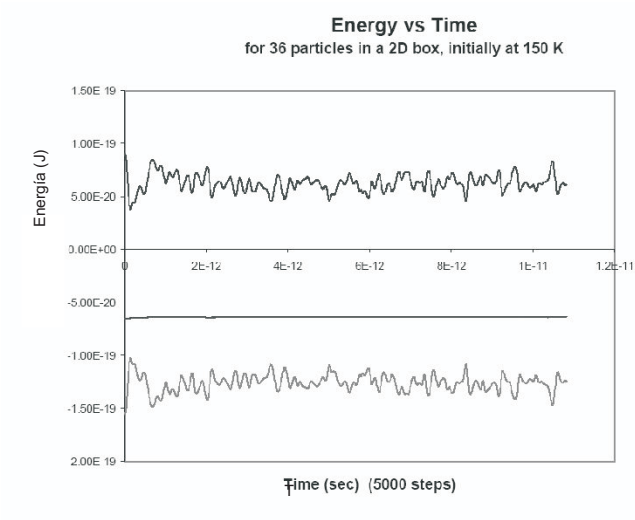
Applet In the table of links you will find a number of 2-D animations (movies) of solutions to the MD equations. Some frames from these animations are shown in Figure 16.3. We recommend that you look at them in order to better visualize what the particles do during an MD simulation. In particular, these simulations use a potential and temperature that should lead to a solid or liquid system, and so you should see the particles binding together.

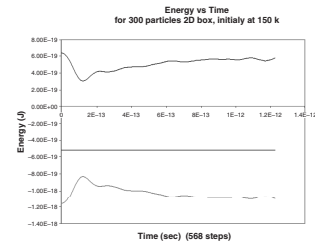
Listing 16.1 **MD.py** performs a 1-D MD simulation with too small a number of large time steps for just a few particles. To be realistic the user should change the parameters and the number of random numbers added to form the Gaussian distribution.

```
# MD.py Molecular dynamics in 2D

from visual.graph import *
import random
scene = display(x=0,y=0,width=350,height=350, title='Molecular Dynamics',
                range=10)
sceneK = gdisplay(x=0,y=350,width=600,height=150,title='Average KE',
         ymin=0.0,ymax=5.0,xmin=0,xmax=500,xtitle='time',ytitle='KE avg')
Kavegraph=gcurve(color= color.red)
sceneT = gdisplay(x=0,y=500,width=600,height=150,title='Average PE',
          ymin=-60,ymax=0.,xmin=0,xmax=500,xtitle='time',ytitle='PE avg')
Tcurve = gcurve(color=color.cyan)
Natom = 25
Nmax =  25
Tinit = 2.0

dens = 1.0                                          # density (1.20 for fcc)
t1 = 0
x  = zeros( (Nmax),     float)                         # x position of atoms
y  = zeros( (Nmax),     float)                                  # y position
vx = zeros( (Nmax),     float)                            # vel. x of atoms
vy = zeros( (Nmax),     float)                  # y component velocity atoms
fx = zeros( (Nmax, 2), float)                         # x component of force
fy = zeros( (Nmax, 2), float)                         # y component of force
L = int(1.*Natom**0.5)                           # side of square with atoms
atoms=[]

def twelveran():                     # computes average of 12 random numbers
    s=0.0
    for i in range (1,13):
        s += random.random()
    return s/12.0-0.5

def initialposvel():                # initial positions, velocities of atoms
    i = -1
    for ix in range(0, L):                  # x->   0  1  2  3  4
        for iy in range(0, L):              # y=0   0  5  10 15 20
            i = i + 1                       # y=1   1  6  11 16 21
            x[i]  = ix                      # y=2   2  7  12 17 22
            y[i]  = iy                      # y=3   3  8  13 18 23
            vx[i] = twelveran()             # y=4   4  9  14 19 24
            vy[i] = twelveran()             # numbering of 25 atoms
            vx[i] = vx[i]*sqrt(Tinit)
            vy[i] = vy[i]*sqrt(Tinit)
    for j in range(0,Natom):
        xc = 2*x[j] - 4
        yc = 2*y[j] - 4
        atoms.append(sphere(pos=(xc,yc), radius=0.5,color=color.red))

def sign(a, b):
    if (b >=  0.0):
        return abs(a)
    else:
        return  - abs(a)

def Forces(t, w, PE, PEorW):    # sets the forces on each of 25 particles
    # invr2 = 0.
    r2cut = 9.                          # of PE or W==1 computes PE else, w
    PE = 0.
    for i in range(0, Natom):
        fx[i][t] = fy[i][t]  = 0.0  # to make sums, start in 0
    for i in range( 0, Natom-1 ):
          for j in range(i + 1, Natom):
              dx = x[i] - x[j]                          # atom separation x
              dy = y[i] - y[j]                          # atom separation y
              if (abs(dx) > 0.50*L):            # smallest r from part/image
                  dx = dx  -  sign(L, dx)       # interact with closer image
```

```
                if (abs(dy) > 0.50*L):                          # same for y
                    dy = dy - sign(L, dy)
                r2 = dx*dx + dy*dy                          # (distance)**2
                if (r2 < r2cut):                        # if less than rcut**2
                    if (r2 == 0.):                  # to avoid 0 denominator
                        r2 = 0.0001
                    invr2 = 1./r2                       # compute this factor
                    wij = 48.*(invr2**3 - 0.5) *invr2**3
                    fijx = wij*invr2*dx
                    fijy = wij*invr2*dy
                    fx[i][t] = fx[i][t] + fijx
                    fy[i][t] = fy[i][t] + fijy
                    fx[j][t] = fx[j][t] - fijx          # in opposite sense
                    fy[j][t] = fy[j][t] - fijy     # for next i iteration
                    PE = PE + 4.*(invr2**3)*((invr2**3) - 1.)
                    w = w + wij
    if (PEorW == 1):
        return PE
    else:
        return w

def timevolution():
    avT = 0.0
    avP = 0.0
    Pavg = 0.0
    avKE = 0.0
    avPE = 0.0
    t1 = 0
    PE = 0.0
    h = 0.031                                                   # step
    hover2 = h/2.0
    # initial KE & PE via Forces
    KE = 0.0
    w = 0.0
    initialposvel()
    for i in range(0, Natom):
        KE = KE+(vx[i]*vx[i]+vy[i]*vy[i])/2.0
  # System.out.println(""+t+" PE= "+PE+" KE = "+KE+" PE+KE = "+(PE+KE));
    PE = Forces(t1,w,PE,1)
    time =1
    while 1:
        # rate(100)
        for i in range(0, Natom):
            PE = Forces(t1,w,PE,1)
            x[i] = x[i] + h*(vx[i] + hover2*fx[i][t1])  # velocity Verlet
            y[i] = y[i] + h*(vy[i] + hover2*fy[i][t1]);
            if x[i] <= 0.:
                x[i] = x[i] + L            # periodic boundary conditions
            if x[i] >= L :
               x[i] = x[i] - L
            if y[i] <= 0.:
                y[i] = y[i] + L
            if y[i] >= L:
                y[i] = y[i] - L

            xc = 2*x[i] - 4
            yc = 2*y[i] - 4
            atoms[i].pos=(xc,yc)

        PE = 0.
        t2=1
        PE = Forces(t2, w, PE, 1)
        KE = 0.
        w = 0.
        for i in range(0 , Natom):
            vx[i] = vx[i] + hover2*(fx[i][t1] + fx[i][t2])
            vy[i] = vy[i] + hover2*(fy[i][t1] + fy[i][t2])
            KE = KE + (vx[i]*vx[i] + vy[i]*vy[i])/2
        w = Forces(t2, w, PE, 2)
        P=dens*(KE+w)
        T=KE/(Natom)
        # increment averages
        avT = avT + T                                       # Temperature
        avP = avP + P                                          # Pressure
        avKE = avKE + KE                                  # Kinetic energy
        avPE = avPE + PE                                # Potential energy
        time += 1
        t=time
        if (t==0):
            t=1
        Pavg  = avP /t
        eKavg = avKE /t
        ePavg = avPE /t
```

Figure 16.5 The kinetic, potential, and total energy for a 2-D MD simulation with 36 particles (*left*), and 300 particles (*right*), both with an initial temperature of 150 K. The potential energy is negative, the kinetic energy is positive, and the total energy is seen to be conserved (flat).





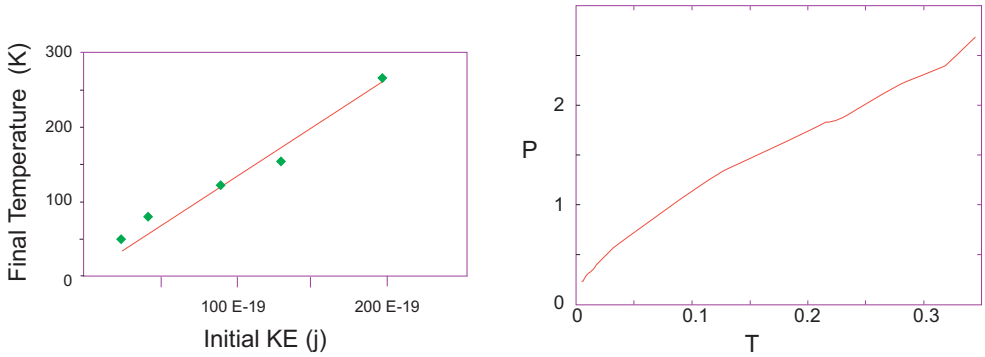

```
        Tavg  = avT /t
        pre = (int)(Pavg*1000)
        Pavg = pre/1000.0
        kener = (int)(eKavg*1000)
        eKavg = kener/1000.0
        Kavegraph.plot(pos=(t,eKavg))
        pener = (int)(ePavg*1000)
        ePavg = pener/1000.0
        tempe = (int)(Tavg*1000000)
        Tavg = tempe/1000000.0
        Tcurve.plot(pos=(t,ePavg),display=sceneT)

timevolution()
```

MD2D.py

The program **MD.py** implements an MD simulation in 1-D using the velocity–Verlet algorithm. Use it as a model and do the following:

1. Ensure that you can run and visualize the 1-D simulation.
2. Place the particles initially at the sites of a simple cubic lattice. The equilibrium configuration for a Lennard-Jones system at low temperature is a face-centered-cubic, and if your simulation is running properly, then the particles should migrate from SC to FCC. The particles will find their own ways from the SC. An FCC lattice has four quarters of a particle per unit cell, so an L^3 box with a lattice constant L/N contains (parts of) $4N^3$ = 32, 108, 256, . . . particles.
3. To save computing time, assign initial particle velocities corresponding to a fixed-temperature Maxwellian distribution.
4. Print the code and indicate on it which integration algorithm is used, where the periodic boundary conditions are imposed, where the nearest image interaction is evaluated, and where the potential is cut off.
5. A typical time step is $\Delta t = 10^{-14}$ s, which in our natural units equals 0.004. You probably will need to make 10^4–10^5 such steps to equilibrate, which corresponds to a total time of only 10^{-9} s (a lot can happen to a speedy molecule in 10^{-9} s). Choose the *largest* time step that provides stability and gives results similar to Figure 16.5.
6. The PE and KE change with time as the system equilibrates. Even after that, there will be fluctuations since this is a dynamic system. Evaluate the time-averaged energies for

Figure 16.6 *Left:* The temperature after equilibration as a function of initial kinetic energy for a 2-D MD simulation with 36 particles. The two are nearly proportional. *Right:* The pressure *versus* temperature for a simulation with several hundred particles. An ideal gas (noninteracting particles) would yield a straight line. (Courtesy of J. Wetzel.)



an equilibrated system.

7. Compare the final temperature of your system to the initial temperature. Change the initial temperature and look for a simple relation between it and the final temperature (Figure 16.6).

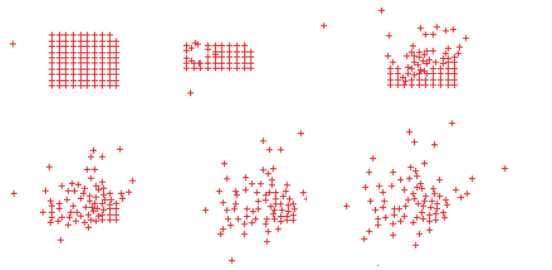
16.4 TRAJECTORY ANALYSIS

1. Modify your program so that it outputs the coordinates and velocities of some particles throughout the simulation. Note that you do not need as many time steps to follow a trajectory as you do to compute it and so you may want to use the *mod* operator `%100` for output.
2. Start your assessment with a 1-D simulation at zero temperature. The particles should remain in place without vibration. Increase the temperature and note how the particles begin to move about and interact.
3. Try starting off all your particles at the minima in the Lennard-Jones potential. The particles should remain bound within the potential until you raise the temperature.
4. Repeat the simulations for a 2-D system. The trajectories should resemble billiard ball–like collisions.
5. Create an animation of the time-dependent locations of several particles.
6. Calculate and plot as a function of temperature the root-mean-square displacement of molecules:
$$R_{\rm rms} = \sqrt{\left\langle |\mathbf{r}(t+\Delta t) - \mathbf{r}(t)|^2 \right\rangle}, \tag{16.21}$$
where the average is over all the particles in the box. Determine the approximate time dependence of $R_{\rm rms}$.
7. Test your system for time-reversal invariance. Stop it at a fixed time, reverse all the velocities, and see if the system retraces its trajectories back to the initial configuration.

16.5 QUIZ

1. We wish to make an MD simulation *by hand* of the positions of particles 1 and 2 that are in a 1-D box of side 8. For an origin located at the center of the box, the particles are initially at rest and at locations $x_i(0) = -x_2(0) = 1$. The particles are subject to the

Figure 16.7 A simulation of a projectile shot into a group of particles. The energy introduced by the projectile is seen to lead to evaporation of the particles. (Courtesy of J. Wetzel.)



force

$$F(x) = \begin{cases} 10, & \text{for } |x_1 - x_2| \leq 1, \\ -1, & \text{for } 1 \leq |x_1 - x_2| \leq 3, \\ 0, & \text{otherwise.} \end{cases} \tag{16.22}$$

Use a simple algorithm to determine the positions of the particles up until the time they leave the box. Make sure to apply periodic boundary conditions. *Hint:* Since the configuration is symmetric, you know the location of particle 2 by symmetry and do not need to solve for it. We suggest the Verlet algorithm (no velocities) with a forward-difference algorithm to initialize it. To speed things along, use a time step of $h = 1/\sqrt{2}$.

Chapter Seventeen

PDEs for Electrostatics & Heat Flow

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Intro to PDEs	pdf	17.1	PDE Electrostatics I	pdf	17.2
Electrostatics II	pdf	17.4	Finite Elements Electrostatics	pdf	17.10
PDE Heat	pdf	17.16	Heat Crank-N	pdf	17.19

Applets Applet

Name	*Sections*
Heat Equation	17.16–17.18

17.1 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 variations. As time evolves, the changes in $U(x, y, z, t)$ at any one position affect the field at neighboring points. This means that the dynamic 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 two-independent variable PDE is

$$A\,\frac{\partial^2 U}{\partial x^2} + 2B\,\frac{\partial^2 U}{\partial x \partial y} + C\,\frac{\partial^2 U}{\partial y^2} + D\,\frac{\partial U}{\partial x} + E\,\frac{\partial U}{\partial y} = F, \tag{17.1}$$

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 [A&W 01], with the next two rows being examples:

We usually think of a parabolic equation as containing a first-order derivative in one variable and a second-order derivative in the other; a hyperbolic equation as containing second-order

Elliptic	*Parabolic*	*Hyperbolic*
$d = AC - B^2 > 0$	$d = AC - B^2 = 0$	$d = AC - B^2 < 0$
$\nabla^2 U(x) = -4\pi\rho(x)$	$\nabla^2 U(\mathbf{x}, t) = a\,\partial U/\partial t$	$\nabla^2 U(\mathbf{x}, t) = c^{-2}\partial^2 U/\partial t^2$
Poisson's	Heat	Wave

Table 17.1 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	*Unique & stable (1-D)*
Dirichlet closed surface	*Unique & stable*	Overspecified	Overspecified
Neumann open surface	Underspecified	Underspecified	*Unique & Stable (1-D)*
Neumann closed surface	*Unique & stable*	Overspecified	Overspecified
Cauchy open surface	Nonphysical	*Unique & stable*	Overspecified
Cauchy closed surface	Overspecified	Overspecified	Overspecified

derivatives of all the variables, with opposite signs 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.

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 and 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 the value of both the 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.[1]

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,

$$\frac{d\mathbf{y}(t)}{dt} = \mathbf{f}(\mathbf{y}, t), \tag{17.2}$$

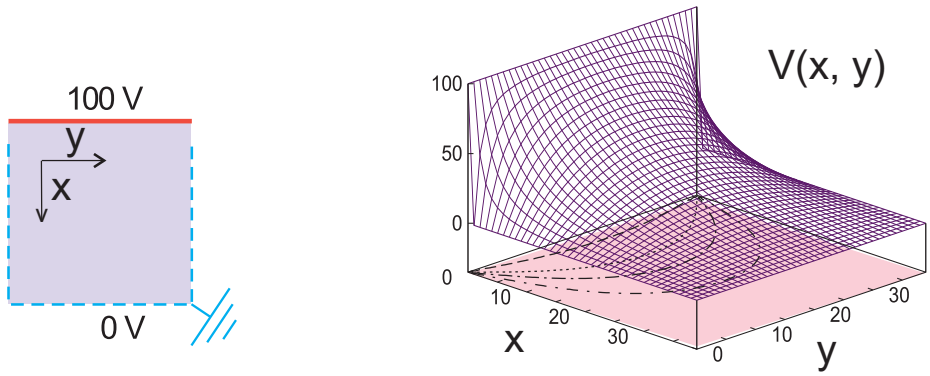
with t the single independent variable, we are able to use a standard algorithm, **`rk4`** in our case, to solve all such equations. Yet because PDEs have several independent variables, for example, $\rho(x, y, z, t)$, we would have to apply (17.2) simultaneously and independently to each variable, which would be very complicated. Second, since there are more equations to solve with PDEs than with ODEs, we need more information than just the two *initial conditions* $[x(0), \dot{x}(0)]$. In addition, because each PDE often has its own particular set of boundary conditions, we have to develop a special algorithm for each particular problem.

17.2 UNIT I. ELECTROSTATIC POTENTIALS

Your **problem** is to find the electric potential for all points *inside* the charge-free square shown in Figure 17.1. 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

[1] 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 [M&F 53] use the finite-difference form to derive the relations between boundary conditions and uniqueness for each type of equation shown in Table 17.1 [Jack 88].

Figure 17.1 *Left:* The shaded region of space within a square in which we determine the electric potential by solving Laplace's equation. There is a wire at the top kept at a constant 100 V and a grounded wire (dashed) at the sides and bottom. *Right:* The computed electric potential as a function of x and y. The projections onto the shaded xy plane are equipotential (contour) lines.



100 V.

17.2.1 Laplace's Elliptic PDE (Theory)

We consider the entire square in Figure 17.1 as our boundary with the voltages prescribed upon it. If we imagine infinitesimal insulators placed at the top corners of the box, then we will 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 Table 17.1, 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 [Jack 88]:

$$\nabla^2 U(\mathbf{x}) = -4\pi\rho(\mathbf{x}), \tag{17.3}$$

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*:

$$\nabla^2 U(\mathbf{x}) = 0. \tag{17.4}$$

Both these equations are elliptic PDEs of a form that occurs in various applications. We solve them in 2-D rectangular coordinates:

$$\frac{\partial^2 U(x,y)}{\partial x^2} + \frac{\partial^2 U(x,y)}{\partial y^2} = \begin{cases} 0, & \text{Laplace's equation,} \\ -4\pi\rho(\mathbf{x}), & \text{Poisson's equation.} \end{cases} \tag{17.5}$$

In both cases we see that the potential depends simultaneously on x and y. For Laplace's equation, the charges, which are the source of the field, enter indirectly by specifying the potential values in some region of space; for Poisson's equation they enter directly.

17.3 FOURIER SERIES SOLUTION OF A PDE

For the simple geometry of Figure 17.1 an analytic solution of Laplace's equation

$$\frac{\partial^2 U(x,y)}{\partial x^2} + \frac{\partial^2 U(x,y)}{\partial y^2} = 0 \tag{17.6}$$

exists in the form of an infinite series. If we assume that the solution is the product of independent functions of x and y and substitute the product into (17.6), we obtain

$$U(x,y) = X(x)Y(y) \quad \Rightarrow \quad \frac{d^2X(x)/dx^2}{X(x)} + \frac{d^2Y(y)/dy^2}{Y(y)} = 0. \tag{17.7}$$

Because $X(x)$ is a function of only x, and $Y(y)$ of only y, the derivatives in (17.7) are *ordinary* as opposed to *partial* derivatives. Since $X(x)$ and $Y(y)$ are assumed to be independent, the only way (17.7) can be valid for *all* values of x and y is for each term in (17.7) to be equal to a constant:

$$\frac{d^2Y(y)/dy^2}{Y(y)} = -\frac{d^2X(x)/dx^2}{X(x)} = k^2, \tag{17.8}$$

$$\Rightarrow \quad \frac{d^2X(x)}{dx^2} + k^2X(x) = 0, \quad \frac{d^2Y(y)}{dy^2} - k^2Y(y) = 0. \tag{17.9}$$

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 with these boundary conditions.

The solutions for $X(x)$ are periodic, and those for $Y(y)$ are exponential:

$$X(x) = A\sin kx + B\cos kx, \quad Y(y) = Ce^{ky} + De^{-ky}. \tag{17.10}$$

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

$$kL = n\pi, \quad n = 1, 2, \ldots . \tag{17.11}$$

Such being the case, for each value of n there is the solution

$$X_n(x) = A_n \sin\left(\frac{n\pi}{L}x\right). \tag{17.12}$$

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 $D = -C$:

$$Y_n(y) = C(e^{k_n y} - e^{-k_n y}) \equiv 2C\,\sinh\left(\frac{n\pi}{L}y\right). \tag{17.13}$$

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:

$$U(x,y) = \sum_{n=1}^{\infty} E_n\,\sin\left(\frac{n\pi}{L}x\right)\,\sinh\left(\frac{n\pi}{L}y\right). \tag{17.14}$$

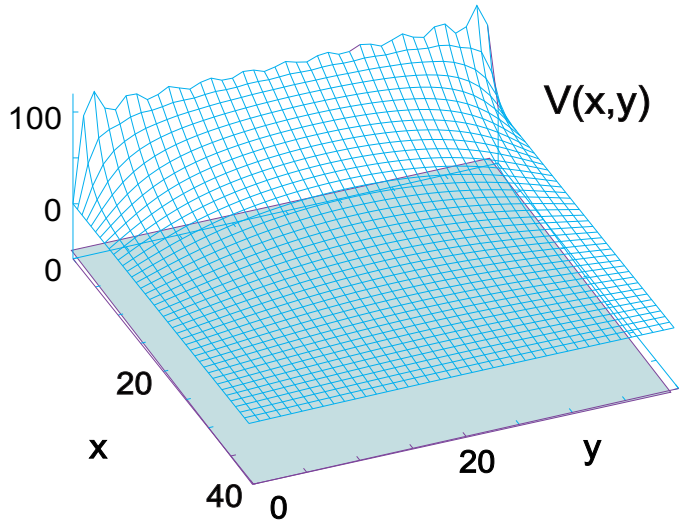
The E_n values are arbitrary constants and are fixed by requiring the solution to satisfy the remaining boundary condition at $y = L$, $U(x, y = L) = 100$ V:

$$\sum_{n=1}^{\infty} E_n\,\sin\frac{n\pi}{L}x\,\sinh n\pi = 100\,\text{V}. \tag{17.15}$$

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:

$$\sum_{n}^{\infty} E_n\,\sinh n\pi \int_0^L dx\,\sin\frac{n\pi}{L}x\,\sin\frac{m\pi}{L}x = \int_0^L dx\,100\,\sin\frac{m\pi}{L}x. \tag{17.16}$$

Figure 17.2 The analytic (Fourier series) solution of Laplace's equation summing 21 terms. Gibbs-overshoot leads to the oscillations near $x = 0$, and persist even if a large number of terms is summed.



The integral on the LHS is nonzero only for $n = m$, which yields

$$E_n = \begin{cases} 0, & \text{for } n \text{ even}, \\ \frac{4(100)}{n\pi \sinh n\pi}, & \text{for } n \text{ odd}. \end{cases} \tag{17.17}$$

Finally, we obtain the potential at any point (x, y) as

$$U(x,y) = \sum_{n=1,3,5,\ldots}^{\infty} \frac{400}{n\pi} \sin\left(\frac{n\pi x}{L}\right) \frac{\sinh(n\pi y/L)}{\sinh(n\pi)}. \tag{17.18}$$

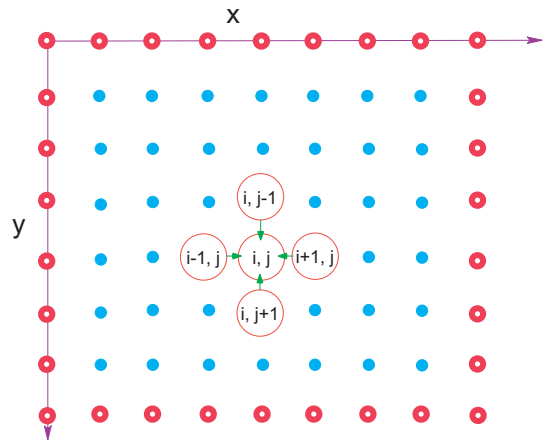
17.3.1 Polynomial Expansion As an Algorithm

It is worth pointing out that even though a product of separate functions of x and y is an acceptable form for a solution to Laplace's equation, this does not mean that the solution to realistic problems will have this form. Indeed, a realistic solution can be expressed as an infinite sum of such products, but the sum is no longer separable. Worse than that, as an algorithm, we must stop the sum at some point, yet the series converges so painfully slowly that many terms are needed, and so round-off error may become a problem. In addition, the $\sinh$ functions in (17.18) overflow 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:

$$\frac{\sinh(n\pi y/L)}{\sinh(n\pi)} = \frac{e^{n\pi(y/L-1)} - e^{-n\pi(y/L+1)}}{1 - e^{-2n\pi}} \to e^{n\pi(y/L-1)}. \tag{17.19}$$

A third problem with the "analytic" solution is that a Fourier series converges only in the *mean square* (Figure 17.2). This means that it converges to the *average* of the left- and right-hand limits in the regions where the solution is discontinuous [Krey 98], such as in the corners of the box. Explicitly, what you see in Figure 17.2 is a phenomenon known as the **Gibbs 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, where, in contrast, the numerical solution required only hundreds of iterations.

Figure 17.3 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.



17.4 SOLUTION: FINITE-DIFFERENCE METHOD

To solve our 2-D PDE numerically, we divide space up into a lattice (Figure 17.3) 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 (17.5), we follow the same path taken in § 7.1 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) and above and below (x, y):

$$U(x+\Delta x, y) = U(x, y) + \frac{\partial U}{\partial x}\Delta x + \frac{1}{2}\frac{\partial^2 U}{\partial x^2}(\Delta x)^2 + \cdots, \tag{17.20}$$

$$U(x-\Delta x, y) = U(x, y) - \frac{\partial U}{\partial x}\Delta x + \frac{1}{2}\frac{\partial^2 U}{\partial x^2}(\Delta x)^2 - \cdots. \tag{17.21}$$

All odd terms cancel when we add these equations, and we obtain a central-difference approximation for the second partial derivative good to order Δ^4:

$$\frac{\partial^2 U(x, y)}{\partial x^2} \simeq \frac{U(x+\Delta x, y) + U(x-\Delta x, y) - 2U(x, y)}{(\Delta x)^2} \tag{17.22}$$

$$\frac{\partial^2 U(x, y)}{\partial y^2} \simeq \frac{U(x, y+\Delta y) + U(x, y-\Delta y) - 2U(x, y)}{(\Delta y)^2} \tag{17.23}$$

Substituting both these approximations in Poisson's equation (17.5) leads to a finite-difference form of the PDE:

$$\frac{U(x+\Delta x, y) + U(x-\Delta x, y) - 2U(x, y)}{(\Delta x)^2} + \frac{U(x, y+\Delta y) + U(x, y-\Delta y) - 2U(x, y)}{(\Delta y)^2} = -4\pi\rho.$$

We assume that the x and y grids are of equal spacings $\Delta x = \Delta y = \Delta$, and so the algorithm

takes the simple form

$$U(x+\Delta,y)+U(x-\Delta,y)+U(x,y+\Delta)+U(x,y-\Delta)-4U(x,y)=-4\pi\rho. \quad (17.24)$$

The reader will notice that this equation shows 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, we obtain a set of $N_x \times N_y$ simultaneous linear algebraic equations for `U[i][j]` to solve. One approach is to solve these equations explicitly as a (big) matrix problem. This is attractive, as it is a direct solution, but it requires a great deal of memory and accounting. The approach we follow here is based on the algebraic solution of (17.24) for $U(x,y)$:

$$\begin{aligned} 4U(x,y) \simeq\, & U(x+\Delta,y)+U(x-\Delta,y)+U(x,y+\Delta)+U(x,y-\Delta) \\ & +4\pi\rho(x,y)\Delta^2, \end{aligned} \quad (17.25)$$

where we would omit the $\rho(x)$ term for Laplace's equation. In terms of discrete locations on our lattice, the x and y variables are

$$x=x_0+i\Delta, \quad y=y_0+j\Delta, \quad i,j=0,\ldots,N_{\text{max-1}}, \quad (17.26)$$

where we have placed our lattice in a square of side L. The finite-difference algorithm (17.25) becomes

$$U_{i,j}=\frac{1}{4}\left[U_{i+1,j}+U_{i-1,j}+U_{i,j+1}+U_{i,j-1}\right]+\pi\rho(i\Delta,j\Delta)\Delta^2. \quad (17.27)$$

This equation says that when we have a proper solution, it will be the average of the potential at the four nearest neighbors (Figure 17.3) plus a contribution from the local charge density. As an algorithm, (17.27) 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 an 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; otherwise 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 ∇^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 cylindrical or spherical coordinates in order to have grids that fit the geometry better.

17.4.1 Relaxation and Overrelaxation

There are a number of ways in which the algorithm (17.25) can be iterated so as to convert the boundary conditions to a solution. Its most basic form is the *Jacobi method* and is one in which the potential values are not changed until an entire sweep of applying (17.25) at each point is completed. This maintains the symmetry of the initial guess and boundary conditions. A rather obvious improvement on the Jacobi method employs the updated guesses for the potential in

(17.25) as soon as they are available. As a case in point, if the sweep starts in the upper-left-hand corner of Figure 17.3, then the leftmost `(i --1, j)` and topmost `(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: (Gauss–Seidel method)

$$U_{i,j}^{(\text{new})} = \frac{1}{4}\left[U_{i+1,j}^{(\text{old})} + U_{i-1,j}^{(\text{new})} + U_{i,j+1}^{(\text{old})} + U_{i,j-1}^{(\text{new})}\right] \tag{17.28}$$

This technique, known as the *Gauss–Seidel* (GS) *method*, usually leads to accelerated convergence, which in turn leads to less round-off 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, which one hopes is insignificant when convergence is reached.

A less obvious improvement in the relaxation technique, known as *successive overrelaxation* (SOR), starts by writing the algorithm (17.25) 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:

$$U_{i,j}^{(\text{new})} = U_{i,j}^{(\text{old})} + r_{i,j}. \tag{17.29}$$

While the Gauss–Seidel technique may still be used to incorporate the updated values in $U^{(\text{old})}$ to determine r, we rewrite the algorithm here in the general form:

$$\begin{aligned} r_{i,j} &\equiv U_{i,j}^{(\text{new})} - U_{i,j}^{(\text{old})} \\ &= \frac{1}{4}\left[U_{i+1,j}^{(\text{old})} + U_{i-1,j}^{(\text{new})} + U_{i,j+1}^{(\text{old})} + U_{i,j-1}^{(\text{new})}\right] - U_{i,j}^{(\text{old})}. \end{aligned} \tag{17.30}$$

The successive overrelaxation technique [Pres 94, Gar 00] proposes that if convergence is obtained by adding r to U, then more rapid convergence might be obtained by adding more or less of r:

$$U_{i,j}^{(\text{new})} = U_{i,j}^{(\text{old})} + \omega r_{i,j}, \quad \text{(SOR)}, \tag{17.31}$$

where ω is a parameter that amplifies or reduces the residual. The nonaccelerated relaxation algorithm (17.28) is obtained with $\omega = 1$, accelerated convergence (overrelaxation) is obtained with $\omega \geq 1$, and underrelaxation is obtained with $\omega < 1$. Values of $1 \leq \omega \leq 2$ 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 ω, we suggest that you explore different values for ω to see which one works best for your particular problem.

17.4.2 Lattice PDE Implementation

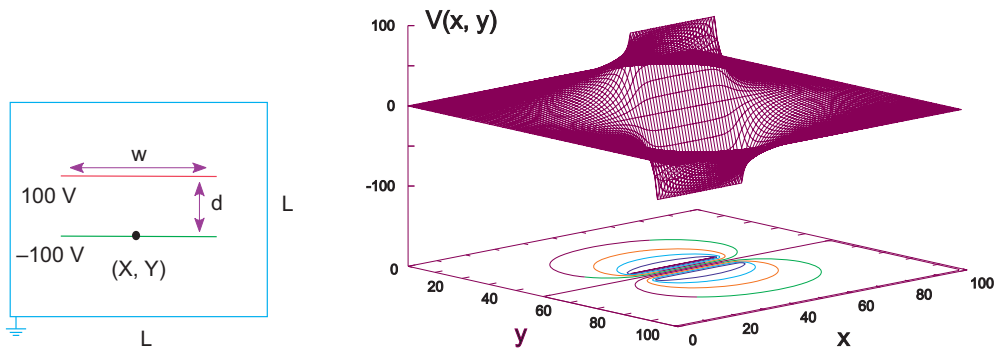
In Listing 17.1 we present the code **`LaplaceLine.py`** that solves the square-wire problem (Figure 17.1). Here we have kept the code simple by setting the length of the box $L = N_{\max}\Delta = 100$ and by taking $\Delta = 1$:

$$\begin{aligned} U(i, N_{\max}) &= 99 \quad \text{(top)}, & U(1, j) &= 0 \quad \text{(left)}, \\ U(N_{\max}, j) &= 0 \quad \text{(right)}, & U(i, 1) &= 0 \quad \text{(bottom)}, \end{aligned} \tag{17.32}$$

We run the algorithm (17.27) for a fixed 1000 iterations. A better code would vary Δ and the dimensions and would quit iterating once the solution converges to some tolerance. Study, compile, and execute the basic code.

Listing 17.1 **LaplaceLine.py** solves Laplace's equation via relaxation. The various parameters need to be adjusted for an accurate solution.

Figure 17.4 *Left:* A simple model of a parallel-plate capacitor within a box. A realistic model would have the plates close 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.




```
# LaplaceLine.py:  Solve Laplace's eqtn, 3D matplot, close shell to quit

from numpy import *
import matplotlib.pylab as p;
from mpl_toolkits.mplot3d import Axes3D

print("Initializing")
Nmax = 100; Niter = 70; V = zeros((Nmax, Nmax), float)            # float maybe Float

print( "Working hard, wait for the figure while I count to 60")
for k in range(0, Nmax-1):  V[k,0] = 100.0                              # line at 100V

for iter in range(Niter):                                     # iterations over algorithm
    if iter%10 == 0: print( iter)
    for i in range(1, Nmax-2):
        for j in range(1,Nmax-2): V[i,j] = 0.25*(V[i+1,j]+V[i-1,j]+V[i,j+1]+V[i,j-1])
x = range(0, Nmax-1, 2);  y = range(0, 50, 2)                   # plot every other point
X, Y = p.meshgrid(x,y)

def functz(V):                                                  # Function returns V(x, y)
    z = V[X,Y]
    return z

Z = functz(V)
fig = p.figure()                                                          # Create figure
ax = Axes3D(fig)                                                           # plot axes
ax.plot_wireframe(X, Y, Z, color = 'r')                                   # red wireframe
ax.set_xlabel('X')                                                           # label axes
ax.set_ylabel('Y')
ax.set_zlabel('Potential')
p.show()                                                     # display fig, close shell to quit
```

17.5 ASSESSMENT VIA SURFACE PLOT

After executing **`LaplaceLine.py`**, you should see a surface plot like Figure 17.1. Study this file in order to understand how to make surface plots with Matplotlib in Python. 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 3-D surface plots we show in this chapter were made with both *gnuplot*, and below we show how to do that.

`>` **`gnuplot`**	Start Gnuplot system from a shell
`gnuplot>` **`set hidden3d`**	Hide surface whose view is blocked
`gnuplot>` **`set unhidden3d`**	Show surface though hidden from view
`gnuplot>` **`splot 'Laplace.dat' with lines`**	Surface plot of Laplace.dat with lines

```
gnuplot> set view 65,45                                  Set x and y rotation viewing angles
gnuplot> replot                                          See effect of your change
gnuplot> set contour                                     Project contours onto xy plane
gnuplot> set cntrparam levels 10                         10 contour levels
gnuplot> set terminal PostScript                         Output PostScript for printing
gnuplot> set output "Laplace.ps"                         Output to file Laplace.ps
gnuplot> splot `Laplace.dat' w l                         Plot again, output to file
gnuplot> set terminal x11                                To see output on screen again
gnuplot> set title `Potential V(x,y) vs x,y'             Title graph
gnuplot> set xlabel `x Position'                         Label x axis
gnuplot> set ylabel `y Position'                         Label y axis
gnuplot> set zlabel `V(x,y)'; replot                     Label z axis and replot
gnuplot> help                                            Tell me more
gnuplot> set nosurface                                   Do not draw surface; leave contours
gnuplot> replot                                          Draw plot again
gnuplot> quit                                            Get out of Gnuplot
```

Because Gnuplot 4 and later versions permit 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* 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, set the terminal type back to **`x11`** again (for Unix's *X Windows System*) and then plot it again.

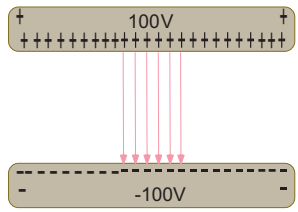
17.6 ALTERNATE CAPACITOR PROBLEMS

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 a more interesting one involving a realistic capacitor or nonplanar capacitors. We now describe the capacitor problem and then move on to the assessment and exploration.

Loading permission pending, laplace.mp4

Elementary textbooks solve the capacitor problem for the uniform field confined between two infinite plates. The field in a finite capacitor varies near the edges (edge effects) and extends beyond the edges of the capacitor (fringe fields). We model the realistic capacitor in a grounded box (Figure 17.4) as two plates (wires) of finite length. Write your simulation such that it is convenient to vary the grid spacing Δ and the geometry of the box and plate. We pose three versions of this problem, each displaying somewhat different physics. In each case the boundary condition $V = 0$ on the surrounding box must be imposed for all iterations in order to obtain a unique solution.

Figure 17.5 A guess as to how charge may rearrange itself on finite conducting plates.



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 composed of a line of dielectric material with uniform charge densities ρ on the top and $-\rho$ on the bottom. Solve Poisson's equation (17.3) in the region including the plates, and Laplace's equation elsewhere. Experiment until you find a numerical value for ρ that gives a potential similar to that shown in Figure 17.6 for plates with fixed voltages.
3. For the final version of this problem investigate how the charges on a capacitor with finite-thickness conducting plates (Figure 17.5) distribute themselves. Since the plates are conductors, they are still equipotential surfaces at 100 and -100 V, only now they have a thickness of at least 2Δ (so we can see the difference between the potential near the top and the bottom surfaces of the plates). Such being the case, we solve Laplace's equation (17.4) much as before to determine $U(x, y)$. Once we have $U(x, y)$, we substitute it into Poisson's equation (17.3) 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, some charge may end up on the edges and outer surfaces of the plates (Figure 17.4).
4. The numerical solution to our PDE can be applied to arbitrary boundary conditions. Two boundary conditions to explore are triangular and sinusoidal:

$$U(x) = \begin{cases} 200x/w, & x \le w/2, \\ 100(1 - x/w), & x \ge w/2, \end{cases} \qquad U(x) = 100 \sin\left(\frac{2\pi x}{w}\right).$$

5. **Square conductors:** You have designed a piece of equipment consisting of a small metal box at 100 V within a larger grounded one (Figure 17.8). 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 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.
6. **Cracked cylindrical capacitor:** You have designed the cylindrical capacitor containing a long outer cylinder surrounding a thin inner cylinder (Figure 17.8 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 that since our algorithm is based

Figure 17.6 *Left:* A Gnuplot visualization of the computed electric potential for a capacitor with finite width plates. *Right:* An OpenDX visualization of the charge distribution along one plate determined by evaluating $\nabla^2 V(x, y)$ (courtesy of J. Wetzel). Note the "lightening rod" effect of charge accumulating at corners and points.

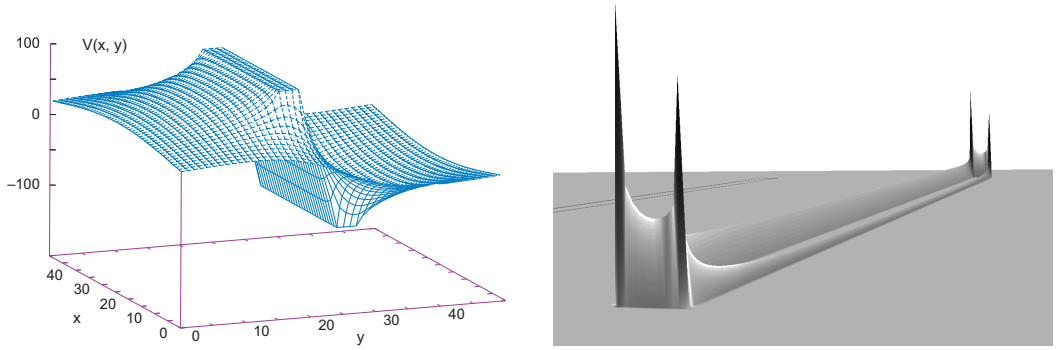
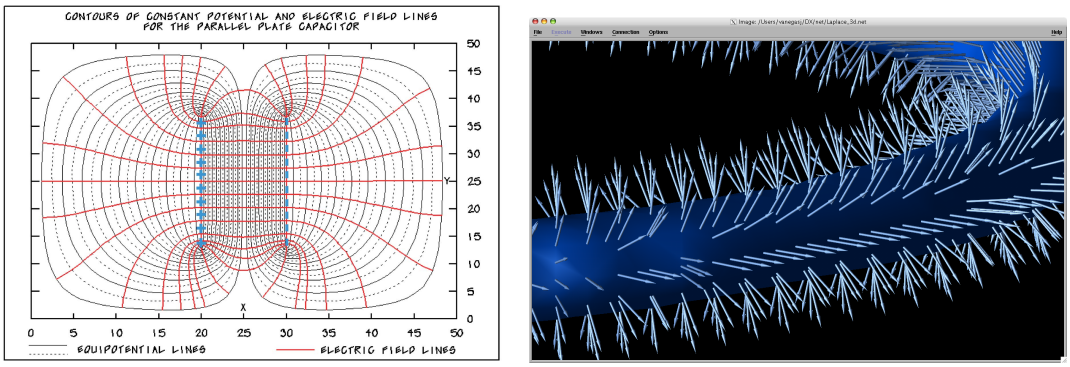


Figure 17.7 *Left:* Computed equipotential surfaces and electric field lines for a realistic capacitor. *Right:* Equipotential surfaces and electric field lines mapped onto the surface for a 3-D capacitor constructed from two tori.



on expansion of the Laplacian in rectangular coordinates, you cannot just convert it to a radial and angle grid.

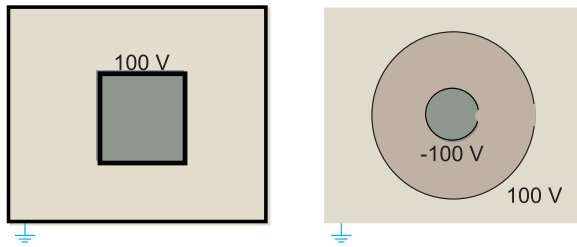
17.7 IMPLEMENTATION AND ASSESSMENT

1. Write or modify the program to find the electric potential for a capacitor within a grounded box. Use the labeling scheme on the left in Figure 17.4.
2. To start, have your program undertake 1000 iterations and then quit. 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 Δ and draw conclusions regarding the stability and accuracy of the solution.
4. Once your program produces reasonable solutions, modify it so that it stops iterating after convergence is reached, or if the number of iterations becomes too large. Rather than trying to discern small changes in highly compressed surface plots, use a numerical measure of precision, for example,

$$\texttt{trace} = \sum_i |\texttt{V[i][i]}|,$$

which samples the solution along the diagonal. Remember, this is a simple algorithm and so may require many iterations for high precision. You should be able to obtain changes in the trace that are less than 1 part in 10^4. (The **`break`** command or a **`while`**

Figure 17.8 *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.



loop is useful for this type of test.)

5. Equation (17.31) expresses the **successive overrelaxation** technique in which convergence is accelerated by using a judicious choice of ω. Determine by trial and error the approximate best value of ω. You will be able to double the speed.
6. Now that the code is accurate, modify it to simulate a more realistic capacitor in which the plate separation is approximately $\frac{1}{10}$ of the plate length. You should find the field more condensed and more uniform between the plates.
7. If you are working with the wire-in-the-box problem, compare your numerical solution to the analytic one (17.18). Do not be surprised if you need to sum thousands of terms before the analytic solution converges!

17.8 ELECTRIC FIELD VISUALIZATION (EXPLORATION)

Plot the equipotential surfaces on a separate 2-D plot. Start with a crude, hand-drawn sketch of the electric field by drawing curves orthogonal to the equipotential lines, beginning and ending on the boundaries (where the charges lie). The regions of high density are regions of high electric field. Physics tells us that the electric field $\mathbf{E}$ is the negative gradient of the potential:

$$\mathbf{E} = -\nabla U(x,y) = -\frac{\partial U(x,y)}{\partial x}\,\hat{\epsilon}_x - \frac{\partial U(x,y)}{\partial y}\,\hat{\epsilon}_y, \tag{17.33}$$

where $\hat{\epsilon}_i$ is a unit vector in the i direction. While at first it may seem that some work is involved in determining these derivatives, once you have a solution for $U(x,y)$ on a grid, it is simple to use the central-difference approximation for the derivative to determine the field, for example:

$$E_x \simeq \frac{U(x+\Delta,y) - U(x-\Delta,y)}{2\Delta} = \frac{U_{i+1,j} - U_{i-1,j}}{2\Delta}. \tag{17.34}$$

Once you have a data file representing such a vector field, it can be visualized by plotting arrows of varying lengths and directions, or with just lines (Figure 17.7). This is possible in Maple and Mathematica [L 05] or with **`vectors`** style in Gnuplot[2], where **`N`** is a normalization factor.

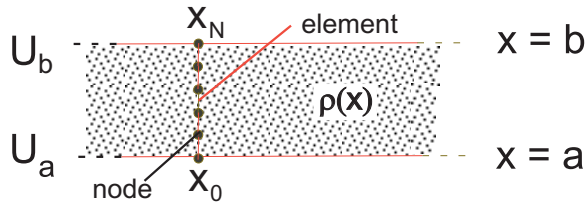
17.9 LAPLACE QUIZ

You are given a simple Laplace-type equation

$$\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = -\rho(x,y),$$

[2]The Gnuplot command **`plot "Laplace_field.dat" using 1:2:3:4 with Vectors`** plots variable-length arrows at (x,y) with components `Dx` $\propto E_x$ and **`Dy`** $\propto E_y$. You determine empirically what scale factor gives you the best visualization (nonoverlapping arrows). Accordingly, you output data lines of the form **`(x, y, Ex/N, Ey/N)`**

Figure 17.9 A finite elements solution to Laplace's equation for two metal plates with a charge density between them. The dots are the nodes x_i, and the lines connecting the nodes are the finite elements.



where x and y are Cartesian spatial coordinates and $\rho(x, y)$ is the charge density in space.

1. Develop a simple algorithm that will permit you to solve for the potential u between two square conductors kept at fixed u, with a charge density ρ between them.
2. Make a simple sketch that shows with arrows how your algorithm works.
3. Make sure to specify how you start and terminate the algorithm.
4. **Thinking outside the box**⊙: Find the electric potential for all points *outside* the charge-free square shown in Figure 17.1. Is your solution unique?

17.10 UNIT II. FINITE-ELEMENT METHOD ⊙

In this unit we solve a simpler problem than the one in Unit I (1-D rather than 2-D), but we do it with a less simple algorithm (finite element). Our usual approach to PDEs in this text uses finite differences to approximate various derivatives in terms of the finite differences of a function evaluated upon a fixed grid. The finite-element method (FEM), in contrast, breaks space up into multiple geometric objects (elements), determines an approximate form for the solution appropriate to each element, and then matches the solutions up at the domain edges.

The theory and practice of FEM as a numerical method for solving partial differential equations have been developed over the last 30 years and still provide an active field of research. One of the theoretical strengths of FEM is that its mathematical foundations allow for elegant proofs of the convergence of solutions to many delicate problems. One of the practical strengths of FEM is that it offers great flexibility for problems on irregular domains or highly varying coefficients or singularities. Although finite differences are simpler to implement than FEM, they are less robust mathematically and less efficient in terms of computer time for big problems. Finite elements in turn are more complicated to implement but more appropriate and precise for complicated equations and complicated geometries. In addition, the same basic finite-element technique can be applied to many problems with only minor modifications and yields solutions that may be evaluated for any value of x, not just those on a grid. In fact, the finite-elements method with various preprogrammed multigrid packages has very much become the standard for large-scale practical applications. Our discussion is based upon [Shaw 92, Li, Otto].

17.11 ELECTRIC FIELD FROM CHARGE DENSITY (PROBLEM)

You are given two conducting plates a distance $b - a$ apart, with the lower one kept at potential U_a, the upper plate at potential U_b, and a uniform charge density $\rho(x)$ placed between them (Figure 17.9). Your **problem** is to compute the electric potential between the plates.

17.12 ANALYTIC SOLUTION

The relation between charge density $\rho(\mathbf{x})$ and potential $U(\mathbf{x})$ is given by Poisson's equation (17.5). For our problem, the potential U changes only in the x direction, and so the PDE becomes the ODE:

$$\frac{d^2U(x)}{dx^2} = -4\pi\rho(x) = -1, \quad 0 < x < 1, \tag{17.35}$$

where we have set $\rho(x) = 1/4\pi$ to simplify the programming. The solution we want is subject to the Dirichlet boundary conditions:

$$U(x = a = 0) = 0, \quad U(x = b = 1) = 1, \tag{17.36}$$

$$\Rightarrow \quad U(x) = -\frac{x}{2}(x - 3). \tag{17.37}$$

Although we know the analytic solution, we shall develop the finite-element method for solving the ODE as if it were a PDE (it would be in 2-D) and as if we did not know the solution. Although we will not demonstrate it, this method works equally well for any charge density $\rho(x)$.

17.13 FINITE-ELEMENT (NOT DIFFERENCE) METHODS

In a finite-element method, the domain in which the PDE is solved is split into finite subdomains, called *elements*, and a trial solution to the PDE in each subdomain is hypothesized. Then the parameters of the trial solution are adjusted to obtain a *best fit* (in the sense of Chapter 8, "Solving Systems of Equations with Matrices; Data Fitting") to the exact solution. The numerically intensive work is in finding the best values for these parameters and in matching the trial solutions for the different subdomains. A FEM solution follows six basic steps [Li]:

1. Derivation of a *weak form* of the PDE. This is equivalent to a least-squares minimization of the integral of the difference between the approximate and exact solutions.
2. Discretization of the computational domains.
3. Generation of interpolating or trial functions.
4. Assembly of the *stiffness matrix* and *load vector*.
5. Implementation of the boundary conditions.
6. Solution of the resulting linear system of equations.

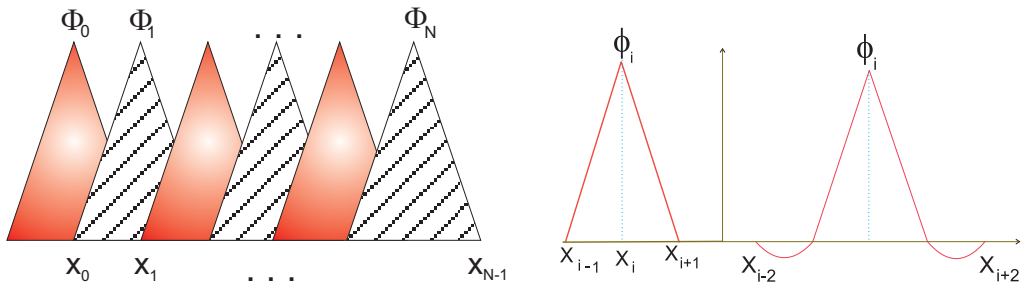
17.13.1 Weak Form of PDE

Finite-difference methods look for an approximate solution of an approximate PDE. Finite-element methods strive to obtain the best possible global agreement of an approximate trial solution with the exact solution. We start with the differential equation

$$-\frac{d^2U(x)}{dx^2} = 4\pi\rho(x). \tag{17.38}$$

A measure of overall agreement must involve the solution $U(x)$ over some region of space, such as the integral of the trial solution. We can obtain such a measure by converting the differential equation (17.38) to its equivalent integral or *weak* form. We assume that we have an approximate or *trial solution* $\Phi(x)$ that vanishes at the endpoints, $\Phi(a) = \Phi(b) = 0$ (we satisfy the boundary conditions later). We next multiply both sides of the differential equation

Figure 17.10 Basis functions used in finite-elements solution of Laplace's equation. *Left:* A set of overlapping basis functions ϕ_i. Each function is a triangle from x_{i-1} to x_{i+1}. *Middle:* A Piecewise-linear function. *Right:* A piecewise-quadratic function.



(17.38) by Φ:

$$-\frac{d^2U(x)}{dx^2}\Phi(x) = 4\pi\rho(x)\Phi(x). \tag{17.39}$$

Next we integrate (17.39) by parts from a to b:

$$-\int_a^b dx\, \frac{d^2U(x)}{dx^2}\Phi(x) = \int_a^b dx\, 4\pi\rho(x)\,\Phi(x) \tag{17.40}$$

$$\begin{aligned} -\frac{dU(x)}{dx}\Phi(x)\,\Big|_a^b + \int_a^b dx\, \frac{dU(x)}{dx}\Phi'(x) &= \int_a^b dx\, 4\pi\rho(x)\,\Phi(x) \\ \Rightarrow \quad \int_a^b dx\, \frac{dU(x)}{dx}\Phi'(x) &= \int_a^b dx\, 4\pi\rho(x)\,\Phi(x) \end{aligned} \tag{17.41}$$

Equation (17.41) is the weak form of the PDE. The unknown exact solution $U(x)$ and the trial function Φ are still to be specified. Because the approximate and exact solutions are related by the integral of their difference over the entire domain, the solution provides a global best fit to the exact solution.

17.13.2 Galerkin Spectral Decomposition

The approximate solution to a weak PDE is found via a stepwise procedure. We split the full domain of the PDE into subdomains called *elements*, find approximate solutions within each element, and then match the elemental solutions onto each other. For our 1-D problem the subdomain elements are straight lines of equal length, while for a 2-D problem, the elements can be parallelograms or triangles (Figure 17.9). Although life is simpler if all the finite elements are the same size, this is not necessary. Indeed, higher precision and faster run times may be obtained by picking small domains in regions where the solution is known to vary rapidly, and picking large domains in regions of slow variation.

The critical step in the finite-element method is expansion of the trial solution U in terms of a set of basis functions ϕ_i:

$$U(x) \simeq \sum_{j=0}^{N-1} \alpha_j \phi_j(x). \tag{17.42}$$

We chose ϕ_i's that are convenient to compute with and then determine the unknown expansion coefficients α_j. Even if the basis functions are not sines or cosines, this expansion is still called a *spectral* decomposition. In order to satisfy the boundary conditions, we will later add another term to the expansion. Considerable study has gone into the effectiveness of different basis functions. If the sizes of the finite elements are made sufficiently small, then good accuracy is obtained with simple piecewise-continuous basis functions, such as the triangles in Figure

17.10. Specifically, we use basis functions ϕ_i that form a triangle or "hat" between x_{i-1} and x_{i+1} and equal 1 at x_i:

$$\phi_i(x) = \begin{cases} 0, & \text{for } x < x_{i-1}, \text{ or } x > x_{i+1}, \\ \frac{x - x_{i-1}}{h_{i-1}}, & \text{for } x_{i-1} \le x \le x_i, \\ \frac{x_{i+1} - x}{h_i}, & \text{for } x_i \le x \le x_{i+1}, \end{cases} \qquad (h_i = x_{i+1} - x_i). \tag{17.43}$$

Because we have chosen $\phi_i(x_i) = 1$, the values of the expansion coefficients α_i equal the values of the (still-to-be-determined) solution at the nodes:

$$U(x_i) \simeq \sum_{i=0}^{N-1} \alpha_i \phi_i(x_i) = \alpha_i \phi_i(x_i) = \alpha_i, \tag{17.44}$$

$$\Rightarrow \quad U(x) \simeq \sum_{j=0}^{N-1} U(x_j)\phi_j(x). \tag{17.45}$$

Consequently, you can think of the hat functions as linear interpolations between the solution at the nodes.

Solution via Linear Equations

Because the basis functions ϕ_i in (17.42) are known, solving for $U(x)$ involves determining the coefficients α_j, which are just the unknown values of $U(x)$ on the nodes. We determine those values by substituting the expansions for $U(x)$ and $\Phi(x)$ into the weak form of the PDE (17.41) and thereby convert them to a set of simultaneous linear equations (in the standard matrix form):

$$\mathbf{A}\mathbf{y} = \mathbf{b}. \tag{17.46}$$

We substitute the expansion $U(x) \simeq \sum_{j=0}^{N-1} \alpha_j \phi_j(x)$ into the weak form (17.41):

$$\int_a^b dx \, \frac{d}{dx} \left(\sum_{j=0}^{N-1} \alpha_j \phi_j(x) \right) \frac{d\Phi}{dx} = \int_a^b dx 4\pi\rho(x)\Phi(x).$$

By successively selecting $\Phi(x) = \phi_0, \phi_1, \ldots, \phi_{N-1}$, we obtain N simultaneous linear equations for the unknown α_j's:

$$\int_a^b dx \, \frac{d}{dx} \left(\sum_{j=0}^{N-1} \alpha_j \phi_j(x) \right) \frac{d\phi_i}{dx} = \int_a^b dx \, 4\pi\rho(x)\phi_i(x), \quad i = 0, N-1. \tag{17.47}$$

We factor out the unknown α_j's and write out the equations explicitly:

$$\begin{aligned}
\alpha_0 \int_a^b \phi_0'\phi_0' \, dx + \alpha_1 \int_a^b \phi_0'\phi_1' \, dx + \cdots + \alpha_{N-1} \int_a^b \phi_0'\phi_{N-1}' \, dx &= \int_a^b 4\pi\rho\phi_0 \, dx, \\
\alpha_0 \int_a^b \phi_1'\phi_0' \, dx + \alpha_1 \int_a^b \phi_1'\phi_1' \, dx + \cdots + \alpha_{N-1} \int_a^b \phi_1'\phi_{N-1}' \, dx &= \int_a^b 4\pi\rho\phi_1 \, dx, \\
&\ddots \\
\alpha_0 \int_a^b \phi_{N-1}'\phi_0' \, dx + \alpha_1 \int \cdots + \alpha_{N-1} \int_a^b \phi_{N-1}'\phi_{N-1}' \, dx &= \int_a^b 4\pi\rho\phi_{N-1} \, dx.
\end{aligned}$$

Because we have chosen the ϕ_i's to be the simple hat functions, the derivatives are easy to evaluate analytically (otherwise they can be done numerically):

$$\frac{d\phi_{i,i+1}}{dx} = \begin{cases} 0, & x < x_{i-1}, \text{or } x_{i+1} < x, \\ \frac{1}{h_{i-1}}, & x_{i-1} \le x \le x_i, \\ \frac{-1}{h_i}, & x_i \le x \le x_{i+1}, \\ 0, & x < x_i, \text{ or } x_{i+2} < x \\ \frac{1}{h_i}, & x_i \le x \le x_{i+1}, \\ \frac{-1}{h_{i+1}}, & x_{i+1} \le x \le x_{i+2}. \end{cases} \tag{17.48}$$

The integrals to evaluate are

$$\int_{x_{i-1}}^{x_{i+1}} dx(\phi_i^{'})^2 = \int_{x_{i-1}}^{x_i} dx \frac{1}{(h_{i-1})^2} + \int_{x_i}^{x_{i+1}} dx \frac{1}{h_i^2} = \frac{1}{h_{i-1}} + \frac{1}{h_i},$$

$$\int_{x_{i-1}}^{x_{i+1}} dx\, \phi_i^{'}\phi_{i+1}^{'} = \int_{x_{i-1}}^{x_{i+1}} dx\, \phi_{i+1}^{'}\phi_i^{'} = \int_{x_i}^{x_{i+1}} dx \frac{-1}{h_i^2} = -\frac{1}{h_i},$$

$$\int_{x_{i-1}}^{x_{i+1}} dx\, (\phi_{i+1}^{'})^2 = \int_{x_i}^{x_{i+1}} dx\, (\phi_{i+1}^{'})^2 = \int_{x_i}^{x_{i+1}} dx\, \frac{+1}{h_i^2} = +\frac{1}{h_i}.$$

We rewrite these equations in the standard matrix form (17.46) with $\mathbf{y}$ constructed from the unknown α_j's, and the tridiagonal stiffness matrix $\mathbf{A}$ constructed from the integrals over the derivatives:

$$\mathbf{y} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \ddots \\ \alpha_{N-1} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} \int_{x_0}^{x_1} dx\, 4\pi\rho(x)\phi_0(x) \\ \int_{x_1}^{x_2} dx\, 4\pi\rho(x)\phi_1(x) \\ \ddots \\ \int_{x_{N-1}}^{x_N} dx 4\pi\rho(x)\phi_{N-1}(x) \end{pmatrix}, \tag{17.49}$$

$$\mathbf{A} = \begin{pmatrix} \frac{1}{h_0} + \frac{1}{h_1} & -\frac{1}{h_1} & -\frac{1}{h_0} & 0 & \cdots \\ -\frac{1}{h_1} & \frac{1}{h_1} + \frac{1}{h_2} & -\frac{1}{h_2} & 0 & \cdots \\ 0 & -\frac{1}{h_2} & \frac{1}{h_2} + \frac{1}{h_3} & -\frac{1}{h_3} & \cdots \\ \ddots & \ddots & -\frac{1}{h_{N-1}} & -\frac{1}{h_{N-2}} & \frac{1}{h_{N-2}} + \frac{1}{h_{N-1}} \end{pmatrix}. \tag{17.50}$$

The elements in $\mathbf{A}$ are just combinations of inverse step sizes and so do not change for different charge densities $\rho(x)$. The elements in $\mathbf{b}$ do change for different ρ's, but the required integrals can be performed analytically or with Gaussian quadrature (Chapter 6, "Integration"). Once $\mathbf{A}$ and $\mathbf{b}$ are computed, the matrix equations are solved for the expansion coefficients α_j contained in $\mathbf{y}$.

Dirichlet Boundary Conditions

Because the basis functions vanish at the endpoints, a solution expanded in them also vanishes there. This will not do in general, and so we add the particular solution $U_a\phi_0(x)$, which satisfies

the boundary conditions [Li]:

$$U(x) = \sum_{j=0}^{N-1} \alpha_j \phi_j(x) + U_a \phi_N(x) \qquad \text{(satisfies boundary conditions)}, \tag{17.51}$$

where $U_a = U(x_a)$. We substitute $U(x) - U_a\phi_0(x)$ into the weak form to obtain $(N+1)$ simultaneous equations, still of the form $\mathbf{Ay} = \mathbf{b}$ but now with

$$\mathbf{A} = \begin{pmatrix} A_{0,0} & \cdots & A_{0,N-1} & 0 \\ & \ddots & & \\ A_{N-1,0} & \cdots & A_{N-1,N-1} & 0 \\ 0 & 0 & \cdots & 1 \end{pmatrix}, \quad \mathbf{b}' = \begin{pmatrix} b_0 - A_{0,0}U_a \\ \ddots \\ b_{N-1} - A_{N-1,0}U_a \\ U_a \end{pmatrix}. \tag{17.52}$$

This is equivalent to adding a new element and changing the load vector:

$$b'_i = b_i - A_{i,0}U_a, \quad i = 1, \ldots, N-1, \quad b'_N = U_a. \tag{17.53}$$

To impose the boundary condition at $x = b$, we again add a term and substitute into the weak form to obtain

$$b'_i = b_i - A_{i,N-1}U_b, \quad i = 1, \ldots, N-1 \quad b'_N = U_b. \tag{17.54}$$

We now solve the linear equations $\mathbf{Ay} = \mathbf{b}'$. For 1-D problems, 100–1000 equations are common, while for 3-D problems there may be millions. Because the number of calculations varies approximately as N^2, it is important to employ an efficient and accurate algorithm because round-off error can easily accumulate after thousands of steps. We recommend one from a scientific subroutine library (see Chapter 8, "Solving Systems of Equations with Matrices; Data Fitting").

17.14 FEM IMPLEMENTATION AND EXERCISES

In Listing 17.2 we give our program **`LaplaceFEM.py`** that determines the FEM solution, and in Figure 17.11 we show that solution. We see on the left that three elements do not provide good agreement with the analytic result, whereas $N = 11$ elements does.

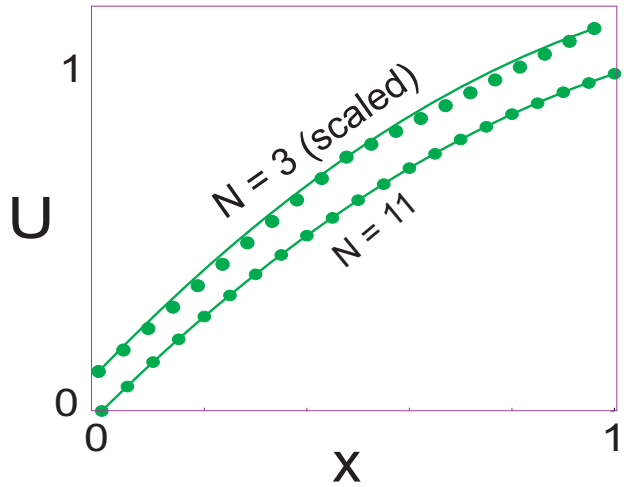
1. Examine the FEM solution for the choice of parameters
$$a = 0, \quad b = 1, \quad U_a = 0, \quad U_b = 1.$$
2. Generate your own triangulation by assigning explicit x values at the nodes over the interval $[0, 1]$.
3. Start with $N = 3$ and solve the equations for N values up to 1000.
4. Examine the stiffness matrix $\mathbf{A}$ and ensure that it is triangular.
5. Verify that the integrations used to compute the load vector $\mathbf{b}$ are accurate.
6. Verify that the solution of the linear equation $\mathbf{Ay} = \mathbf{b}$ is correct.
7. Plot the numerical solution for $U(x)$ for $N = 10$, 100, and 1000 and compare with the analytic solution.
8. The log of the relative global error (number of significant figures) is
$$\mathcal{E} = \log_{10} \left| \frac{1}{b-a} \int_a^b dx \frac{U_{\text{FEM}}(x) - U_{\text{exact}}(x)}{U_{\text{exact}}(x)} \right|.$$
Plot the global error *versus* x for $N = 10$, 100, and 1000.

Listing 17.2 **LaplaceFEM.py** provides a finite-element method solution of the 1-D Laplace equation via a Galerkin spectral decomposition. The resulting matrix equations are solved with Matplotlib. Although the

algorithm is more involved than the solution via relaxation (Listing 17.1), it is a direct solution with no iteration required.

```
# LaplaceFEM.py:  Solutn of Laplace Eq via finite elements method

from numpy import*
from numpy.linalg import solve
from visual.graph import *

N = 11;  h = 1./(N - 1)
u = zeros((N),float);   A = zeros((N,N),float);  b = zeros((N,N),float)
x2 = zeros((21),float);  u_fem = zeros((21),float); u_exact = zeros((21),float)
error = zeros( (21), float); x = zeros( (N),  float)

graph1 = gdisplay(width=500, height=500, title = 'Exact: blue, FEM: red',
 xtitle = 'x', ytitle = 'U',xmax=1,ymax=1,xmin=0,ymin=0)
funct1 = gcurve(color = color.blue); funct2 = gdots(color  = color.red)
funct3 = gcurve(color = color.cyan)

for i in range(0, N): x[i] = i*h
for i in range(0, N):                                          # Initialize
    b[i, 0] = 0.
    for j in range(0, N): A[i][j] = 0.

def lin1(x, x1, x2):  return  (x - x1)/(x2 - x1)                # Hat func

def lin2(x, x1, x2):  return (x2 - x)/(x2 - x1)

def f(x): return 1.                                       # RHS of equation

def int1 (min, max):  # Simpson
    no = 1000
    sum = 0.
    interval = ( (max  - min) /(no - 1) )
    for n in range(2, no, 2):                             # Loop odd points
        x = interval * (n - 1)
        sum   += 4 * f(x)*lin1(x, min, max)
    for n in range(3, no, 2):                            # Loop even points
        x = interval * (n - 1)
        sum   += 2 * f(x)*lin1(x, min, max)
    sum += f(min)*lin1(min, min, max) + f(max)*lin1(max, min, max)
    sum *= interval/6.
    return (sum)

def int2 (min, max):                                          # Simpson
    no = 1000
    sum = 0.
    interval = ( (max  - min) /(no - 1) )
    for n in range(2, no, 2):                             # Loop odd points
        x = interval * (n - 1)
        sum   += 4 * f(x)*lin2(x, min, max)
    for n in range(3, no, 2):                            # Loop even points
        x = interval * (n - 1)
        sum   += 2 * f(x)*lin2(x, min, max)
    sum   += f(min)*lin2(min, min, max) + f(max)*lin2(max, min, max)
    sum *= interval/6.
    return (sum)

def numerical(x, u, xp):
    N = 11                                  # interpolate numerical solution
    y = 0.
    for i in range(0, N - 1):
        if ( xp >= x[i] and xp <= x[i + 1] ):
            y = lin2(xp,x[i],x[i+1])*u[i] + lin1(xp, x[i], x[i+1])*u[i+1]
    return y

def exact(x):                                            # Analytic solution
    u = - x*(x - 3.)/2.
    return u

for i in range(1, N):
    A[i - 1, i - 1] = A[i - 1, i - 1] + 1./h
    A[i - 1, i]  = A[i - 1, i]  -  1./h
    A[i, i - 1]  = A[i - 1, i]
    A[i, i]     = A[i, i] + 1./h
    b[i - 1, 0]  = b[i - 1, 0] + int2(x[i - 1], x[i])
    b[i, 0]     = b[i, 0] + int1(x[i - 1], x[i])
for i in range(1, N):                              # Dirichlet BC @ left end
    b[i, 0] = b[i, 0] - 0.*A[i, 0]
    A[i, 0] = 0.
    A[0, i] = 0.
A[0, 0] = 1.
b[0, 0] = 0.
```

Figure 17.11 Exact (line) *versus* FEM solution (points) for the two-plate problem for $N = 3$ and $N = 11$ finite elements. At this scale the $N = 3$ solution appears identical to the exact one.




```
for i in range(1, N):                          # Dirichlet bc @ right end
    b[i, 0] = b[i, 0] - 1.*A[i, N - 1]
    A[i, N - 1] = 0.
    A[N - 1, i] = 0.
A[N - 1, N - 1] = 1.
b[N - 1, 0] = 1.
sol = solve(A, b)

for i in range(0, N):      u[i] = sol[i, 0]
for i in range(0, 21 ):    x2[i] = 0.05*i
for i in range(0, 21):
    rate(6)
    u_fem[i] = numerical(x, u, x2[i])
    u_exact[i] = exact(x2[i])
    funct1.plot(pos = (0.05*i, u_exact[i]) )
    rate(6)
    funct2.plot(pos = (0.05*i, u_fem[i]) )
    error[i] = u_fem[i] - u_exact[i]                    # Global error
```

17.15 EXPLORATION

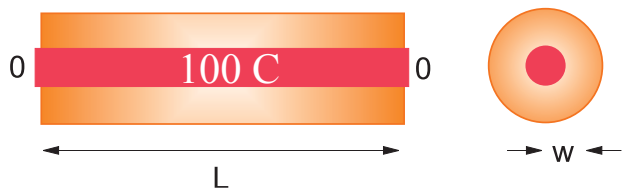
1. Modify your program to use piecewise-quadratic functions for interpolation and compare to the linear function results.
2. Explore the resulting electric field if the charge distribution between the plates has the explicit x dependences

$$\rho(x) = \frac{1}{4\pi}\begin{cases} \frac{1}{2} - x, \\ \sin x, \\ 1 \text{ at } x = 0, \quad -1 \text{ at } x = 1 \ \text{ (a capacitor).} \end{cases}$$

17.16 UNIT III. HEAT FLOW VIA TIME-STEPS (LEAPFROGS)

Problem: You are given an aluminum bar of length $L = 1\,\text{m}$ and width w aligned along the x axis (Figure 17.12). It is insulated along its length but not at its ends. Initially the bar is at a uniform temperature of $100\,\text{C}$, and then both ends are placed in contact with ice water at $0\,\text{C}$. Heat flows out of the noninsulated ends only. Your **problem** is to determine how the temperature will vary as we move along the length of the bar at later times.

Figure 17.12 A metallic bar insulated along its length with its ends in contact with ice. The bar is colored solid red and the insulation is of lighter color.



17.17 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 **H** through a material is proportional to the gradient of the temperature T across the material:

$$\mathbf{H} = -K\nabla T(\mathbf{x}, t), \tag{17.55}$$

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 material's volume:

$$Q(t) = \int d\mathbf{x}\, C\rho(\mathbf{x})\, T(\mathbf{x}, t), \tag{17.56}$$

where C is the specific heat of the material and ρ is its density. Because energy is conserved, the rate of decrease in Q with time must equal the amount of heat flowing out of the material. After this energy balance is struck and the divergence theorem applied, there results the *heat equation*:

$$\frac{\partial T(\mathbf{x}, t)}{\partial t} = \frac{K}{C\rho}\nabla^2 T(\mathbf{x}, t). \tag{17.57}$$

The heat equation (17.57) 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 the Laplacian:

$$\frac{\partial T(x, t)}{\partial t} = \frac{K}{C\rho}\frac{\partial^2 T(x, t)}{\partial x^2}. \tag{17.58}$$

As given, the initial temperature of the bar and the boundary conditions are

$$T(x, t = 0) = 100\,\mathrm{C}, \quad T(x = 0, t) = T(x = L, t) = 0\,\mathrm{C}. \tag{17.59}$$

17.17.1 Solution: Analytic Expansion

Analogous to Laplace's equation, the analytic solution starts with the assumption that the solution separates into the product of functions of space and time:

$$T(x, t) = X(x)\mathcal{T}(t). \tag{17.60}$$

When (17.60) is substituted into the heat equation (17.58) and the resulting equation is divided by $X(x)\mathcal{T}(t)$, two noncoupled ODEs result:

$$\frac{d^2X(x)}{dx^2} + k^2X(x) = 0, \quad \frac{d\mathcal{T}(t)}{dt} + k^2\frac{C}{C\rho}\mathcal{T}(t) = 0, \tag{17.61}$$

where k is a constant still to be determined. The boundary condition that the temperature equals zero at $x = 0$ requires a sine function for X:

$$X(x) = A\sin kx. \tag{17.62}$$

The boundary condition that the temperature equals zero at $x = L$ requires the sine function to vanish there:

$$\sin kL = 0 \;\Rightarrow\; k = k_n = n\pi/L, \quad n = 1, 2, \ldots. \tag{17.63}$$

The time function is a decaying exponential with k in the exponent:

$$\mathcal{T}(t) = e^{-k_n^2 t/C\rho}, \;\Rightarrow\; T(x,t) = A_n \sin k_n x e^{-k_n^2 t/C\rho}, \tag{17.64}$$

where n can be any integer and A_n is an arbitrary constant. Since (17.58) is a linear equation, the most general solution is a linear superposition of all values of n:

$$T(x,t) = \sum_{n=1}^{\infty} A_n \sin k_n x\, e^{-k_n^2 t/C\rho}. \tag{17.65}$$

The coefficients A_n are determined by the initial condition that at time $t = 0$ the entire bar has temperature $T = 100$ K:

$$T(x, t = 0) = 100 \;\Rightarrow\; \sum_{n=1}^{\infty} A_n \sin k_n x = 100. \tag{17.66}$$

Projecting the sine functions determines $A_n = 4T_0/n\pi$ for n odd, and so

$$T(x,t) = \sum_{n=1,3,\ldots}^{\infty} \frac{4T_0}{n\pi} \sin k_n x e^{-k_n^2 Kt/(C\rho)}. \tag{17.67}$$

17.17.2 Solution: Time-Stepping

As we did with Laplace's equation, the numerical solution is based on converting the differential equation to a finite-difference ("difference") equation. We discretize space and time on a lattice (Figure 17.13) and solve for solutions on the lattice sites. 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*.

Applet As is often the case with PDEs, 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:

$$\frac{\partial T(x,t)}{\partial t} \simeq \frac{T(x, t+\Delta t) - T(x,t)}{\Delta t}. \tag{17.68}$$

Because we know the spatial variation of the temperature along the entire top row and the left and right sides, we are less constrained with the space derivative as with the time derivative.

Figure 17.13 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.)

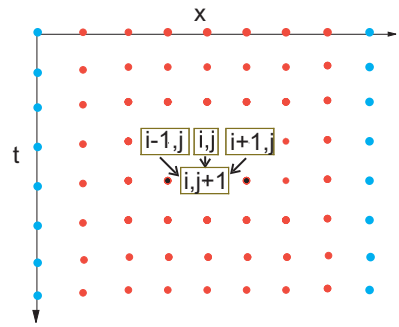
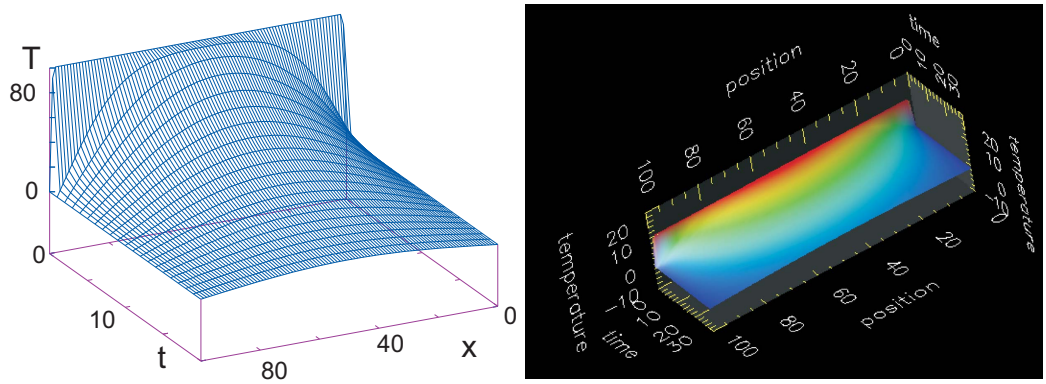


Figure 17.14 A numerical calculation of the temperature *versus* position and *versus* time, with isotherm contours projected onto the horizontal plane on the left and with a red-blue color scale used to indicate temperature on the right.



Consequently, as we did with the Laplace equation, we use the more accurate central-difference approximation for the space derivative:

$$\frac{\partial^2 T(x,t)}{\partial x^2} \simeq \frac{T(x+\Delta x,t) + T(x-\Delta x,t) - 2T(x,t)}{(\Delta x)^2}. \tag{17.69}$$

Substitution of these approximations into (17.58) yields the heat difference equation

$$\frac{T(x,t+\Delta t) - T(x,t)}{\Delta t} = \frac{K}{C\rho}\frac{T(x+\Delta x,t) + T(x-\Delta x,t) - 2T(x,t)}{\Delta x^2}. \tag{17.70}$$

We reorder (17.70) into a form in which T can be stepped forward in t:

$$T_{i,j+1} = T_{i,j} + \eta\left[T_{i+1,j} + T_{i-1,j} - 2T_{i,j}\right], \quad \eta = \frac{K\Delta t}{C\rho\Delta x^2}, \tag{17.71}$$

where $x = i\Delta x$ and $t = j\Delta t$. This algorithm is *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 in Figure. 17.13 simultaneously, then we would have an *implicit* algorithm that requires us to solve equations involving unknown values of the temperature. 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 and keeping the temperature along the ends fixed at 0 K (Figure 17.14).

17.17.3 Von Neumann Stability Assessment

When we solve a PDE by converting it to a difference equation, we hope that the solution of the latter is a good approximation to the solution of the former. If the difference-equation solution diverges, then we know we have a bad approximation, but if it converges, then we may feel confident that we have 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

$$T_{m,j} = \xi(k)^j \, e^{ikm\Delta x}, \tag{17.72}$$

where $x = m\Delta x$ and $t = j\Delta t$, but $i = \sqrt{-1}$ is the imaginary number. The constant k in (17.72) is an unknown wave vector $(2\pi/\lambda)$, and $\xi(k)$ is an unknown complex function. View (17.72) as a basis function that oscillates in space (the exponential) with an amplitude or *amplification factor* $\xi(k)^j$ that increases by a power of ξ 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 ξ cannot grow in time j, which means $|\xi(k)| < 1$ for all values of the parameter k [Pres 94, Anc 02].

Application of a stability analysis is more straightforward than it might appear. We just substitute the expression (17.72) into the difference equation (17.71):

$$\xi^{j+1} e^{ikm\Delta x} = \xi^{j+} e^{ikm\Delta x} \tag{17.73}$$

$$+\eta \left[\xi^j e^{ik(m+1)\Delta x} + \xi^{j+} e^{ik(m-1)\Delta x} - 2\xi^{j+} e^{ikm\Delta x} \right] \quad . \tag{17.74}$$

After canceling some common factors, it is easy to solve for ξ:

$$\xi(k) = 1 + 2\eta[\cos(k\Delta x) - 1]. \tag{17.75}$$

In order for $|\xi(k)| < 1$ for all possible k values, we must have

$$\eta = \frac{K\,\Delta t}{C\rho\,\Delta x^2} < \frac{1}{2}. \tag{17.76}$$

This equation tells us that if we make the time step Δt smaller, we will always improve the stability, as we would expect. But if we decrease the space step Δx without a simultaneous quadratic *increase* in the time step, we will worsen the stability. The lack of space-time symmetry arises from our use of stepping in time but not in space.

In general, you should perform a stability analysis for every PDE you have to solve, although it can get complicated [Pres 94]. Yet even if you do not, the lesson here is that you may have to try different *combinations* of Δx and Δt variations until a stable, reasonable solution is obtained. You may expect, nonetheless, that there are choices for Δx and Δt for which the numerical solution fails and that simply decreasing an individual Δx or Δt, in the hope that this will increase precision, may not improve the solution.

Listing 17.3 **EqHeat.py** solves the heat equation for a 1-D 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.py: solves heat equation via finite differences, 3-D plot

from numpy import *
import matplotlib.pylab as p
from mpl_toolkits.mplot3d import Axes3D

Nx = 101;          Nt = 3000;      Dx = 0.03;      Dt = 0.9
KAPPA = 210.; SPH = 900.; RHO = 2700. # conductivity, specf heat, density
T = zeros( (Nx, 2), float);  Tpl = zeros( (Nx, 31), float)
```

```
print("Working, wait for figure after count to 10")

for ix in range (1, Nx - 1):  T[ix, 0] = 100.0;     # initial temperature
T[0,0] = 0.0 ;   T[0,1] = 0.                  # first and last points at 0
T[Nx-1,0] = 0. ; T[Nx-1,1] = 0.0
cons = KAPPA/(SPH*RHO)*Dt/(Dx*Dx);                                # constant
m = 1                          # counter for rows, one every 300 time steps

for t in range (1, Nt):                                     # time iteration
   for ix in range (1, Nx - 1):                           # Finite differences
      T[ix, 1] = T[ix, 0] +  cons*(T[ix+1, 0] + T[ix-1, 0] - 2.*T[ix,0])
   if t%300 == 0 or t == 1:            # for t = 1 and every 300 time steps
        for ix in range (1, Nx - 1, 2): Tpl[ix, m] = T[ix, 1]
        print(m)
        m = m + 1                       # increase m every 300 time steps
   for ix in range (1, Nx - 1):  T[ix, 0] = T[ix, 1]# 100 positons at t=m
x = list(range(1, Nx - 1, 2))                   # plot every other x point
y = list(range(1, 30))                        # every 10 points in y (time)
X, Y = p.meshgrid(x, y)                        # grid for position and time

def functz(Tpl):                             # Function returns temperature
    z = Tpl[X, Y]
    return z

Z = functz(Tpl)
fig = p.figure()                                            # create figure
ax = Axes3D(fig)                                               # plots axis
ax.plot_wireframe(X, Y, Z, color = 'r')                     # red wireframe
ax.set_xlabel('Position')                                      # label axes
ax.set_ylabel('time')
ax.set_zlabel('Temperature')
p.show()                              # shows figure, close Python shell
print("finished")
```

EqHeatAnimate.py

17.17.4 Heat Equation Implementation

Recollect that we want to solve for the temperature distribution within an aluminum bar of length $L = 1$ m subject to the boundary and initial conditions

$$T(x = 0, t) = T(x = L, t) = 0\,\text{K}, \qquad T(x, t = 0) = 100\,\text{K}. \tag{17.77}$$

The thermal conductivity, specific heat, and density for Al are

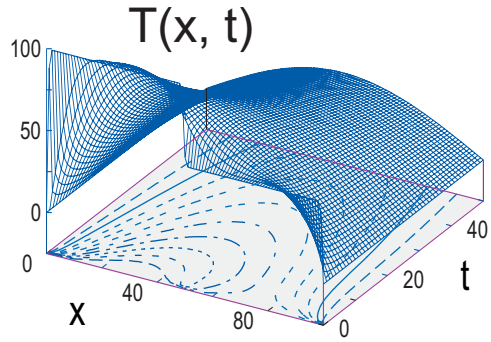
$$K = 237\,\text{W/(mK)}, \quad C = 900\,\text{J/(kg K)}, \quad \rho = 2700\,\text{kg/m}^3. \tag{17.78}$$

1. Write or modify **`EqHeat.py`** in Listing 17.3 to solve the heat equation.
2. Define a 2-D 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 is for present and past times (because you may have to make thousands of time steps, you 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 ends are at 100 K. Set the temperatures of the ends to 0 K.
4. Apply (17.68) 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.`

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 approximately every 500 time steps, print the time and temperature along the bar.

Figure 17.15 Temperature *versus* position and time when two bars at differing temperatures are placed in contact at $t = 0$. The projected contours show the isotherms.



17.18 ASSESSMENT AND VISUALIZATION

1. Check that your program gives a temperature distribution that varies smoothly along the bar and agrees with the boundary conditions, as in Figure 17.14.
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 that does not appear smooth and continuous.
4. Make surface plots of temperature *versus* position for several times.
5. Better yet, make a surface plot of temperature *versus* position *versus* time.
6. Plot the *isotherms* (contours of constant temperature).
7. **Stability test:** Check (17.76) that the temperature diverges in t if $\eta > \frac{1}{4}$.
8. **Material dependence:** Repeat the calculation for iron. 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 the analytic solution,
$$T(x,t) = \sin(\pi x/L)e^{-\pi^2 Kt/(L^2 C\rho)}.$$
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 K. One is kept in a heat bath at 100 K, and the other at 50 K. Determine how the temperature varies with time and location (Figure 17.15).
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
$$\frac{\partial T}{\partial t} = -h(T - T_e), \tag{17.79}$$
where h is a positive constant. This leads to the modified heat equation
$$\frac{\partial T(x,t)}{\partial t} = \frac{K}{C\rho}\frac{\partial^2 T}{\partial^2 x} - hT(x,t). \tag{17.80}$$
Modify the algorithm to include Newton's cooling and compare the cooling of this bar with that of the insulated bar.

17.19 IMPROVED HEAT FLOW: CRANK–NICOLSON METHOD

The Crank–Nicolson method [C&N 47] provides a higher degree of precision for the heat equation (17.57). This method calculates the time derivative with a central-difference approximation, in contrast to the forward-difference approximation used previously. In order to avoid introducing error for the initial time step, where only a single time value is known, the method uses a *split time step*,[3] so that time is advanced from time t to $t + \Delta t/2$:

$$\frac{\partial T}{\partial t}\left(x, t + \frac{\Delta t}{2}\right) \simeq \frac{T(x, t + \Delta t) - T(x, t)}{\Delta t} + O(\Delta t^2). \tag{17.81}$$

Yes, we know that this looks just like the forward-difference approximation for the derivative at time $t + \Delta t$, for which it would be a bad approximation; regardless, it is a better approximation for the derivative at time $t + \Delta t/2$, though it makes the computation more complicated. Likewise, in (17.68) we gave the central-difference approximation for the second space derivative for time t. For $t = t + \Delta t/2$, that becomes

$$\begin{aligned} &2(\Delta x)^2 \frac{\partial^2 T}{\partial x^2}\left(x, t + \frac{\Delta t}{2}\right) \qquad (17.82)\\ &\simeq [T(x - \Delta x,\, t + \Delta t) - 2T(x,\, t + \Delta t) + T(x + \Delta x,\, t + \Delta t)]\\ &+ [T(x - \Delta x,\, t) - 2T(x,\, t) + T(x + \Delta x,\, t)] + O(\Delta x^2). \end{aligned}$$

In terms of these expressions, the heat difference equation is

$$\begin{aligned} &T_{i,j+1} - T_{i,j} = \frac{\eta}{2}\left[T_{i-1,j+1} - 2T_{i,j+1} + T_{i+1,j+1} + T_{i-1,j} - 2T_{i,j} + T_{i+1,j}\right],\\ &x = i\Delta x,\ \ t = j\Delta t,\ \ \eta = \frac{K\Delta t}{C\rho\,\Delta x^2}. \end{aligned} \tag{17.83}$$

We group together terms involving the same temperature to obtain an equation with future times on the LHS and present times on the RHS:

$$-T_{i-1,\,j+1} + \left(\frac{2}{\eta} + 2\right) T_{i,\,j+1} - T_{i+1,\,j+1} = T_{i-1,\,j} + \left(\frac{2}{\eta} - 2\right) T_{i,\,j} + T_{i+1,\,j}. \tag{17.84}$$

This equation represents an *implicit* scheme for the temperature $T_{i,j}$, where the word "implicit" means that we must solve simultaneous equations to obtain the full solution for all space. In contrast, an *explicit* scheme requires iteration to arrive at the solution. It is possible to solve (17.84) simultaneously for all unknown temperatures ($1 \le i \le N$) at times j and $j + 1$. We start with the initial temperature distribution throughout all of space, the boundary conditions at the ends of the bar for all times, and the approximate values from the first derivative:

$$\begin{aligned} &T_{i,\,0},\ \text{known}, && T_{0,\,j},\ \text{known}, && T_{N,\,j}, \text{known},\\ &T_{0,\,j+1} = T_{0,\,j} = 0, && T_{N,\,j+1} = 0, && T_{N,\,j} = 0. \end{aligned}$$

We rearrange (17.84) so that we can use these known values of T to step the $j = 0$ solution forward in time by expressing (17.84) as a set of simultaneous linear equations (in matrix

[3] In §18.6.1 we develop another split-time algorithm for solution of the Schrödinger equation, where the real and imaginary parts of the wave function are computed at times that differ by $\Delta t/2$.

form):

$$\begin{pmatrix} (\frac{2}{\eta}+2) & -1 & & & & & \\ -1 & (\frac{2}{\eta}+2) & -1 & & & & \\ & -1 & (\frac{2}{\eta}+2) & -1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -1 & (\frac{2}{\eta}+2) & -1 & \\ & & & & -1 & (\frac{2}{\eta}+2) & \end{pmatrix} \begin{pmatrix} T_{1,j+1} \\ T_{2,j+1} \\ T_{3,j+1)} \\ \vdots \\ T_{n-2,j+1} \\ T_{n-1,j+1} \end{pmatrix}$$

$$= \begin{pmatrix} T_{0,j+1} + T_{0,j} + (\frac{2}{\eta}-2)T_{1,j} + T_{2,j} \\ T_{1,j} + (\frac{2}{\eta}-2)T_{2,j} + T_{3,j} \\ T_{2,j} + (\frac{2}{\eta}-2)T_{3,j} + T_{4,j} \\ \vdots \\ T_{n-3,j} + (\frac{2}{\eta}-2)T_{n-2,j} + T_{n-1,j} \\ T_{n-2,j} + (\frac{2}{\eta}-2)T_{n-1,j} + T_{n,j} + T_{n,j+1} \end{pmatrix}. \tag{17.85}$$

Observe that the T's on the RHS are all at the present time j for various positions, and at future time $j+1$ for the two ends (whose Ts are known for all times via the boundary conditions). We start the algorithm with the $T_{i,j=0}$ values of the initial conditions, then solve a matrix equation to obtain $T_{i,j=1}$. With that we know all the terms on the RHS of the equations ($j=1$ throughout the bar and $j=2$ at the ends) and so can repeat the solution of the matrix equations to obtain the temperature throughout the bar for $j=2$. So again we time-step forward, only now we solve matrix equations at each step. That gives us the spatial solution directly.

Not only is the Crank–Nicolson method more precise than the low-order time-stepping method of Unit III, but it also is stable for all values of Δt and Δx. To prove that, we apply the von Neumann stability analysis discussed in §17.17.3 to the Crank–Nicolson algorithm by substituting (17.71) into (17.84). This determines an amplification factor

$$\xi(k) = \frac{1 - 2\eta \sin^2(k\Delta x/2)}{1 + 2\eta \sin^2(k\Delta x/2)}. \tag{17.86}$$

Because $\sin^2()$ is positive-definite, this proves that $|\xi| \le 1$ for all Δt, Δx, and k.

17.19.1 Solution of Tridiagonal Matrix Equations ⊙

The Crank–Nicolson equations (17.85) are in the standard $[A]\mathbf{x} = \mathbf{b}$ form for linear equations, and so we can use our previous methods to solve them. Nonetheless, because the coefficient matrix $[A]$ is tridiagonal (zero elements except for the main diagonal and two diagonals on

either side of it),

$$\begin{pmatrix} d_1 & c_1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ a_2 & d_2 & c_2 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & a_3 & d_3 & c_3 & \cdots & \cdots & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & a_{N-1} & d_{N-1} & c_{N-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & a_N & d_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \ddots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \ddots \\ b_{N-1} \\ b_N \end{pmatrix},$$

a more robust and faster solution exists that makes this implicit method as fast as an explicit one. Because tridiagonal systems occur frequently, we now outline the specialized technique for solving them [Pres 94]. If we store the matrix elements $a_{i,j}$ using both subscripts, then we will need N^2 locations for elements and N^2 operations to access them. However, for a tridiagonal matrix, we need to store only the vectors $\{d_i\}_{i=1,N}$, $\{c_i\}_{i=1,N}$, and $\{a_i\}_{i=1,N}$, along, above, and below the diagonals. The single subscripts on a_i, d_i, and c_i reduce the processing from N^2 to $(3N-2)$ elements.

We solve the matrix equation by manipulating the individual equations until the coefficient matrix is *upper triangular* with all the elements of the main diagonal equal to 1. We start by dividing the first equation by d_1, then subtract a_2 times the first equation,

$$\begin{pmatrix} 1 & \frac{c_1}{d_1} & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & d_2 - \frac{a_2 c_1}{d_1} & c_2 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & a_3 & d_3 & c_3 & \cdots & \cdots & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & a_{N-1} & d_{N-1} & c_{N-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & a_N & d_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \ddots \\ \cdot \\ x_N \end{pmatrix} = \begin{pmatrix} \frac{b_1}{d_1} \\ b_2 - \frac{a_2 b_1}{d_1} \\ b_3 \\ \ddots \\ \cdot \\ b_N \end{pmatrix},$$

and then dividing the second equation by the second diagonal element,

$$\begin{pmatrix} 1 & \frac{c_1}{d_1} & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & \frac{c_2}{d_2 - a_2 \frac{c_1}{a_1}} & 0 & \cdots & & \cdots & 0 \\ 0 & a_3 & d_3 & c_3 & \cdots & \cdots & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & & a_{N-1} & d_{N-1} & c_{N-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & a_N & d_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \ddots \\ \cdot \\ x_N \end{pmatrix} = \begin{pmatrix} \frac{b_1}{d_1} \\ \frac{b_2 - a_2 \frac{b_1}{d_1}}{d_2 - a_2 \frac{c_1}{d_1}} \\ b_3 \\ \ddots \\ \cdot \\ b_N \end{pmatrix}.$$

Assuming that we can repeat these steps without ever dividing by zero, the system of equations will be reduced to upper triangular form,

$$\begin{pmatrix} 1 & h_1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & h_2 & 0 & \cdots & 0 \\ 0 & 0 & 1 & h_3 & \cdots & 0 \\ 0 & \cdots & \cdots & \ddots & \ddots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \ddots \\ \cdot \\ x_N \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ \ddots \\ \cdot \\ p_N \end{pmatrix},$$

where $h_1 = c_1/d_1$ and $p_1 = b_1/d_1$. We then recur for the others elements:

$$h_i = \frac{c_i}{d_i - a_i h_{i-1}}, \quad p_i = \frac{b_i - a_i p_{i-1}}{d_i - a_i h_{i-1}}. \tag{17.87}$$

Finally, back substitution leads to the explicit solution for the unknowns:

$$x_i = p_i - h_i x_{i-1}; \quad i = n-1, n-2, \ldots, 1, \quad x_N = p_N. \tag{17.88}$$

In Listing 17.4 we give the program **`HeatCNTridiag.py`** that solves the heat equation using the Crank–Nicolson algorithm via a triadiagonal reduction.

Listing 17.4 **HeatCNTridiag.py** is the complete program for solution of the heat equation in one space dimension and time via the Crank–Nicolson method. The resulting matrix equations are solved via a technique specialized to tridiagonal matrices.

```
# HeatCNTridiag.py:  solution of heat eqtn via CN method

import matplotlib.pylab as p;
from mpl_toolkits.mplot3d import Axes3D ;
from visual import *;

Max = 51; n   = 50;   m = 50
Ta  = zeros((Max),float); Tb =zeros((Max),float); Tc = zeros((Max),float)
Td  = zeros((Max),float); a = zeros((Max),float); b = zeros((Max),float)
c   = zeros((Max),float); d = zeros((Max),float); x = zeros((Max),float)
t   = zeros( (Max, Max),float)

def Tridiag(a, d, c, b, Ta, Td, Tc, Tb, x, n):    # Define Tridiag method
    Max = 51
    h = zeros( (Max), float )
    p = zeros( (Max), float )
    for i in range(1,n+1):
        a[i] = Ta[i]
        b[i] = Tb[i]
        c[i] = Tc[i]
        d[i] = Td[i]
    h[1] = c[1]/d[1]
    p[1] = b[1]/d[1]
    for i in range(2,n+1):
         h[i] = c[i] / (d[i]-a[i]*h[i-1])
         p[i] = (b[i] - a[i]*p[i-1]) / (d[i]-a[i]*h[i-1])
    x[n] = p[n]
    for i in range( n - 1, 1,-1 ): x[i] = p[i] - h[i]*x[i+1]

width = 1.0; height = 0.1; ct = 1.0                    # Rectangle W & H
for i in range(0, n):   t[i,0]  = 0.0                      # Initialize
for i in range( 1, m):  t[0][i] = 0.0
h  = width  / ( n - 1 )                # Compute step sizes and constants
k  = height / ( m - 1 )
r  = ct * ct * k / ( h * h )

for j in range(1,m+1):
     t[1,j] = 0.0
     t[n,j] = 0.0                                                  # BCs
for i in range( 2, n):   t[i][1] = sin( pi * h *i)                 # ICs
for i in range(1, n+1):  Td[i] = 2. + 2./r
Td[1] = 1.; Td[n] = 1.
for i in range(1,n ): Ta[i] = -1.0;      Tc[i] = -1.0;     # Off diagonal
Ta[n-1] = 0.0;   Tc[1] = 0.0; Tb[1] = 0.0; Tb[n] = 0.0
print("I'm working hard, wait for fig while I count to 50")

for j in range(2,m+1):
      print(j)
      for i in range(2,n): Tb[i] = t[i-1][j-1] + t[i+1][j-1] + (2/r-2) * t[i][j-1]
      Tridiag(a, d, c, b, Ta, Td, Tc, Tb, x, n)            # Solve system
      for i in range(1, n+1):  t[i][j] = x[i]
print("Finished")
x = list(range(1, m+1))                        # Plot every other x point
y = list(range(1, n+1))                            # every other y point
X, Y = p.meshgrid(x,y)

def functz(t):                              # Function returns potential
    z = t[X, Y]
    return z
Z = functz(t)
fig = p.figure()                                          # Create figure
ax = Axes3D(fig)                                             # plots axes
ax.plot_wireframe(X, Y, Z, color= 'r')                    # red wireframe
ax.set_xlabel('t')                                           # label axes
ax.set_ylabel('x')
ax.set_zlabel('T')
p.show()                              # display figure, close shell to quit
```

17.19.2 Crank–Nicolson Implementation, Assessment

Use the Crank–Nicolson method to solve for the heat flow in the metal bar in §17.16.

1. Write a program using the Crank–Nicolson method to solve the heat equation for at least 100 time steps.
2. Solve the linear system of equations (17.85) using either Matplotlib or the special tridiagonal algorithm.
3. Check the stability of your solution by choosing different values for the time and space steps.
4. Construct a contoured surface plot of temperature *versus* position and versus time.
5. Compare the implicit and explicit algorithms used in this chapter for relative precision and speed. You may assume that a stable answer that uses very small time steps is accurate.

Chapter Eighteen

PDE Waves: String, Quantum Packet, and E&M

In this chapter we explore the numerical solution of a number of PDEs known as wave equations. *We have two purposes in mind. First, especially if you have skipped the discussion of the heat equation in Chapter 17,* "PDES for Electrostatics & Heat Flow," *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. Unit I deals with a number of aspects of waves on a string. Unit II deals with quantum wave packets, which have their real and imaginary parts solved for at different (split) times. Unit III extends the treatment to electromagnetic waves that have the extra complication of being vector waves with interconnected E and H fields. Shallow-water waves, dispersion, and shock waves are studied in Chapter 19,* "Solitons and Computational Fluid Dynamics."

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Realistic String Waves	pdf	18.1	Catenary Waves	pdf	18.3
Quantum WavePackets	pdf	18.5	EM Waves (FDTD)	pdf	18.9

Applets & Animations Applet

Name	*Sections*	*Name*	*Sections*
Waves on a String	18.1–18.2	String normal mode	18.1–18.2
Two peaks on a String	18.1–18.2	Catenary Wave Movie	18.4
Wavepacket-Wavepacket Scattering	18.5-18.7	Wavepacket Slit Movie	18.5-18.7
Square Well	18.6	Harmonic Oscillator	18.7
Two Slit Interference	18.8		

18.1 UNIT I. VIBRATING STRING

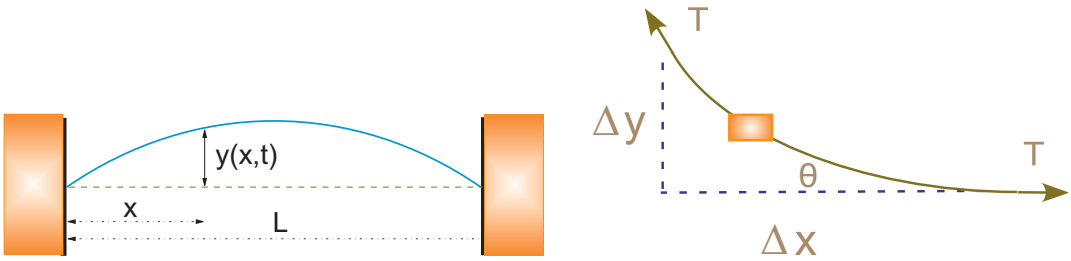
Problem: Recall the demonstration from elementary physics in which a string tied down at both ends is plucked "gently" at one location and a pulse is observed to travel along the string. Likewise, if the string has one end free and you shake it just right, a standing-wave pattern is set up in which the nodes remain in place and the antinodes move just up and down. Your **problem** is to develop an accurate model for wave propagation on a string and to see if you can set up traveling- and standing-wave patterns.[1]

18.2 THE HYPERBOLIC WAVE EQUATION (THEORY)

Consider a string of length L tied down at both ends (Figure 18.1 left). The string has a constant density ρ per unit length, a constant tension T, no frictional forces acting on it, and a tension

[1] Some similar but independent studies can also be found in [Raw 96].

Figure 18.1 *Left:* A stretched string of length L tied down at both ends and under high enough tension to ignore gravity. The vertical disturbance of the string from its equilibrium position is $y(x,t)$. *Right:* A differential element of the string showing how the string's displacement leads to the restoring force.



that is so high that we may ignore sagging due to gravity. We assume that 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 Chapter 19, "Solitons & Computational Fluid Dynamics"), 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 Δx of the string (Figure 18.1 right) and see that the difference in the vertical components of the tension at 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:

$$\sum F_y = \rho\,\Delta x\,\frac{\partial^2 y}{\partial t^2}, \tag{18.1}$$

$$\begin{aligned} &= T\sin\theta(x+\Delta x) - T\sin\theta(x) \\ &= T\left.\frac{\partial y}{\partial x}\right|_{x+\Delta x} - T\left.\frac{\partial y}{\partial x}\right|_{x} \simeq T\frac{\partial^2 y}{\partial x^2}, \end{aligned} \tag{18.2}$$

$$\Rightarrow \quad \frac{\partial^2 y(x,t)}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 y(x,t)}{\partial t^2}, \quad c = \sqrt{\frac{T}{\rho}}, \tag{18.3}$$

where we have assumed that θ 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 that this signal velocity c is *not* the same as the velocity of a string element $\partial y/\partial t$.

The initial condition for our problem is that the string is plucked gently and released. We assume that the "pluck" places the string in a triangular shape with the center of triangle $\frac{8}{10}$ of the way down the string and with a height of 1:

$$y(x,t=0) = \begin{cases} 1.25x/L, & x \le 0.8L, \\ (5-5x/L), & x > 0.8L, \end{cases} \quad \text{(initial condition 1).} \tag{18.4}$$

Because (18.3) 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:

$$\frac{\partial y}{\partial t}(x,t=0) = 0, \quad \text{(initial condition 2).} \tag{18.5}$$

The boundary conditions have both ends of the string tied down for all times:

$$y(0,t) \equiv 0, \quad y(L,t) \equiv 0, \quad \text{(boundary conditions)}. \tag{18.6}$$

18.2.1 Solution via Normal-Mode Expansion

Applet

The analytic solution to (18.3) is obtained via the familiar separation-of-variables technique. We assume that the solution is the product of a function of space and a function of time:

$$y(x,t) = X(x)T(t). \tag{18.7}$$

We substitute (18.7) into (18.3), divide by $y(x,t)$, and are left with an equation that has a solution only if there are solutions to the two ODEs:

$$\frac{d^2T(t)}{dt^2} + \omega^2 T(t) = 0, \quad \frac{d^2X(x)}{dx^2} + k^2 X(x) = 0, \quad k \stackrel{\text{def}}{=} \frac{\omega}{c}. \tag{18.8}$$

The angular frequency ω 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

$$X(x=0,t) = X(x=l,t) = 0 \tag{18.9}$$

$$\Rightarrow \quad X_n(x) = A_n \sin k_n x, \quad k_n = \frac{\pi(n+1)}{L}, \quad n = 0, 1, \ldots. \tag{18.10}$$

The time solution is

$$T_n(t) = C_n \sin \omega_n t + D_n \cos \omega_n t, \quad \omega_n = nck_0 = n\frac{2\pi c}{L}, \tag{18.11}$$

where the frequency of this nth *normal mode* is also fixed. In fact, it is the single frequency of oscillation that defines a normal mode. The *initial condition* (18.4) of zero velocity, $\partial y/\partial t(t = 0) = 0$, requires the C_n values in (18.11) to be zero. Putting the pieces together, the normal-mode solutions are

$$y_n(x,t) = \sin k_n x \cos \omega_n t, \quad n = 0, 1, \ldots. \tag{18.12}$$

Since the wave equation (18.3) 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:

$$y(x,t) = \sum_{n=0}^{\infty} B_n \sin k_n x \cos \omega_n t. \tag{18.13}$$

(Yet we will lose linear superposition once we include nonlinear terms in the wave equation.) The Fourier coefficient B_n is determined by the second initial condition (18.4), which describes how the wave is plucked:

$$y(x, t=0) = \sum_{n}^{\infty} B_n \sin nk_0 x. \tag{18.14}$$

Multiply both sides by $\sin mk_0 x$, substitute the value of $y(x,0)$ from (18.4), and integrate from 0 to l to obtain

$$B_m = 6.25 \, \frac{\sin(0.8m\pi)}{m^2\pi^2}. \tag{18.15}$$

You will be asked to compare the Fourier series (18.13) 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 be done only approximately.

18.2.2 Algorithm: Time-Stepping

Applet

As with Laplace's equation and the heat equation, we look for a solution $y(x,t)$ only for discrete values of the independent variables x and t on a grid (Figure 18.2):

$$x = i\Delta x, \quad i = 1, \ldots, N_x, \quad t = j\Delta t, \quad j = 1, \ldots, N_t, \tag{18.16}$$

$$y(x,t) = y(i\Delta x,\ i\Delta t) \stackrel{\text{def}}{=} y_{i,j}. \tag{18.17}$$

In contrast to Laplace's equation where the grid was in two space dimensions, the grid in Figure 18.2 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 time, while moving down a column corresponds to increasing time steps for a fixed position. Even though the grid in Figure 18.2 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:

$$\frac{\partial^2 y}{\partial t^2} \simeq \frac{y_{i,j+1} + y_{i,j-1} - 2y_{i,j}}{(\Delta t)^2}, \quad \frac{\partial^2 y}{\partial x^2} \simeq \frac{y_{i+1,j} + y_{i-1,j} - 2y_{i,j}}{(\Delta x)^2}. \tag{18.18}$$

Substituting (18.18) in the wave equation (18.3) yields the difference equation

$$\frac{y_{i,j+1} + y_{i,j-1} - 2y_{i,j}}{c^2(\Delta t)^2} = \frac{y_{i+1,j} + y_{i-1,j} - 2y_{i,j}}{(\Delta x)^2}. \tag{18.19}$$

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:

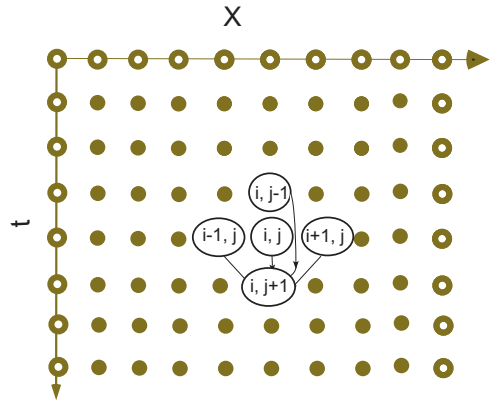
$$y_{i,j+1} = 2y_{i,j} - y_{i,j-1} + \frac{c^2}{c'^2}\left[y_{i+1,j} + y_{i-1,j} - 2y_{i,j}\right], \quad c' \stackrel{\text{def}}{=} \frac{\Delta x}{\Delta t}. \tag{18.20}$$

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 (18.20) 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 (Figure 18.2).

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 one step at a time. If we write the solution for present times to a file, then we need to store only three time values on the computer, which saves memory. In fact, because the time steps must be quite small to obtain high precision, you may want to store the solution only for every fifth or tenth time.

Initializing the recurrence relation is a bit tricky because it requires displacements from two earlier times, whereas the initial conditions are for only one time. Nonetheless, the rest condition (18.4) when combined with the *central-difference* approximation lets us extrapolate

Figure 18.2 The solutions of the wave equation for four earlier space-time points are used to obtain the solution at the present time. The boundary and initial conditions are indicated by the white-centered dots.



to negative time:

$$\frac{\partial y}{\partial t}(x,0) \simeq \frac{y(x,\Delta t) - y(x,-\Delta t)}{2\Delta t} = 0, \quad \Rightarrow \quad y_{i,0} = y_{i,2}. \tag{18.21}$$

Here we take the initial time as $j = 1$, and so $j = 0$ corresponds to $t = -\Delta t$. Substituting this relation into (18.20) yields for the initial step

$$y_{i,2} = y_{i,1} + \frac{c^2}{2c'^2}\left[y_{i+1,1} + y_{i-1,1} - 2y_{i,1}\right] \quad \text{(for } j = 2 \text{ only)}. \tag{18.22}$$

Equation (18.22) uses the solution throughout all space at the initial time $t = 0$ to propagate (leapfrog) it forward to a time Δt. Subsequent time steps use (18.20) 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 §17.17.3, a complicated equation results. Nonetheless, [Pres 94] shows that the difference-equation solution will be stable for the general class of transport equations if

$$c \le c' = \Delta x/\Delta t \quad \text{(Courant condition)}. \tag{18.23}$$

Equation (18.23) 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 (18.3) 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.

Exercise: Try to figure out a procedure for 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?

18.2.3 Wave Equation Implementation

The program **EqStringAnimate.py** in Listing 18.1 solves the wave equation for a string of length $L = 1$ m with its ends fixed and with the gently plucked initial conditions. Note that our use of $L = 1$ violates our 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 and $T = 40$ N, with the space grid set at 101 points, corresponding to $\Delta = 0.01$ cm.

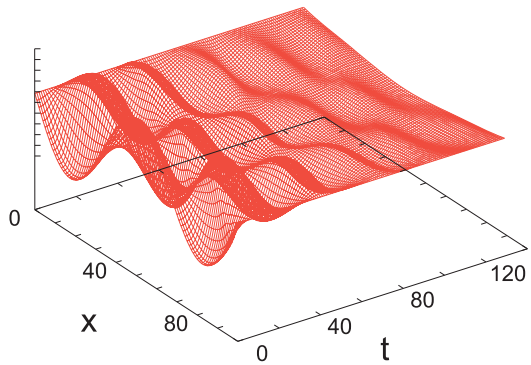
Listing 18.1 **EqStringAnimate.py** solves the wave equation via time stepping for a string of length $L = 1$ m with its ends fixed and with the gently plucked initial conditions. You will need to modify this code to include new physics.

```
# EqString.py:  Animated leapfrog solution of wave equation

from visual import *

# Set up curve
g = display(width = 600, height = 300, title = 'Vibrating string')
vibst = curve(x = list(range(0, 100)), color = color.yellow)
ball1 = sphere(pos = (100, 0), color = color.red, radius = 2)
ball2 = sphere(pos = ( - 100, 0), color = color.red, radius = 2)
ball1.pos
ball2.pos
vibst.radius = 1.0

# Parameters
rho   = 0.01                                          # string density
ten   = 40.                                           # string tension
c     = sqrt(ten/rho)                               # Propagation speed
cl    = c                                               # CFL criterium
ratio =  c*c/(cl*cl)

# Initialization
xi = zeros((101,3), float)                          # 101 x's & 3 t's
for i in range(0, 81):      xi[i, 0] = 0.00125*i;                   # IC
for i in range (81, 101):  xi[i, 0] = 0.1 - 0.005*(i - 80)          # IC
for i in range(0, 100):                                         # 1st t step
    vibst.x[i] = 2.0*i - 100.0                              # assign,scale x
    vibst.y[i] = 300.*xi[i, 0]                             # assign, scale y
vibst.pos                                                       # draw string

# Later time steps
for i in range(1, 100): xi[i,1] = xi[i,0] + 0.5*ratio*(xi[i+1,0]+xi[i-1,0]-2*xi[i,0])
while 1:                                # continue plotting till user quits
    rate(50)                            # delays plotting, (bigger = slower)
    for i in range(1, 100):
        xi[i,2] = 2.*xi[i,1] - xi[i,0] + ratio * (xi[i+1,1]+xi[i-1,1]-2*xi[i, 1])
    for i in range(1, 100):
         vibst.x[i] = 2.*i - 100.0                                  # scaled x
         vibst.y[i] = 300.*xi[i, 2]                                 # scaled y
    vibst.pos                                                   # plot string
    for i in range(0, 101):
        xi[i, 0] = xi[i, 1]                                     # recycle array
        xi[i, 1] = xi[i, 2]

print("Done!")
```

18.2.4 Assessment, Exploration

1. Applet Solve the wave equation and make a surface plot of displacement *versus* time and position.
2. Explore a number of space and time step combinations. In particular, try steps that satisfy and that do not satisfy the Courant condition (18.23). Does your exploration conform with the stability condition?
3. Compare the analytic and numeric solutions, summing at least 200 terms in the analytic solution.

Figure 18.3 The vertical displacement as a function of position x and time t for a string initially plucked near its right end. Notice how a pulse forms and divides into waves traveling to the right and to the left. (Courtesy of J. Wiren.)



4. Use the plotted time dependence to estimate the peak's propagation velocity c. Compare the deduced c to (18.3).
5. Our solution of the wave equation for a plucked string leads to the formation of a wave packet that corresponds to the sum of multiple normal modes of the string. On the right in Figure 18.3 we show the motion resulting from the string initially placed in a single normal mode (standing wave),

$$y(x, t = 0) = 0.001 \sin 2\pi x, \qquad \frac{\partial y}{\partial t}(x, t = 0) = 0.$$

   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. ⊙ Figure 18.4 shows the wave packets that result as a function of time for initial conditions corresponding to the double pluck indicated on the left in the figure. Verify that initial conditions of the form

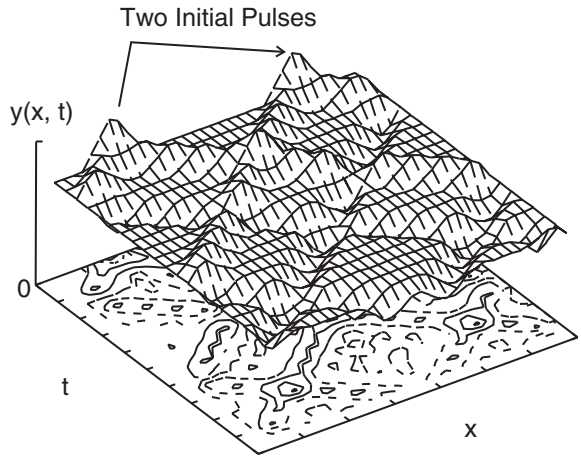
$$\frac{y(x, t = 0)}{0.005} = \begin{cases} 0, & 0.0 \le x \le 0.1, \\ 10x - 1, & 0.1 \le x \le 0.2, \\ -10x + 3, & 0.2 \le x \le 0.3, \\ 0, & 0.3 \le x \le 0.7, \\ 10x - 7, & 0.7 \le x \le 0.8, \\ -10x + 9, & 0.8 \le x \le 0.9, \\ 0, & 0.9 \le x \le 1.0 \end{cases}$$

   lead to this type of a repeating pattern. In particular, observe whether the pulses move or just oscillate up and down.

18.3 WAVES WITH FRICTION (EXTENSION)

The string problem we have investigated so far can be handled by either a numerical or an analytic technique. We now wish to extend the theory to include some more realistic physics.

Figure 18.4 The vertical displacement as a function of position and time of a string initially plucked simultaneously at two points, as shown by arrows. 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 ends. As a consequence of these inversions, the $t \simeq 12$ wave is an inverted $t = 0$ wave.



These extensions have only numerical solutions.

Real plucked strings do not vibrate forever because the real world contains friction. Consider again the element of a string between x and $x + dx$ (Figure 18.1 right) but now imagine that this element is moving in a viscous fluid such as air. An approximate model has the frictional force pointing in a direction opposite the (vertical) velocity of the string and proportional to that velocity, as well as proportional to the length of the string element:

$$F_f \simeq -2\kappa\, \Delta x\, \frac{\partial y}{\partial t}, \tag{18.24}$$

where κ 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

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2} - \frac{2\kappa}{\rho} \frac{\partial y}{\partial t}. \tag{18.25}$$

In Figure 18.3 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 to the left that are reflected and inverted by the fixed ends. Because those parts of the wave with the higher velocity experience greater friction, the peak tends to be smoothed out the most as time progresses.

Exercise: Generalize the algorithm used to solve the wave equation to now include friction and check if the wave's behavior seems physical (damps in time). Start with $T = 40\,\text{N}$ and $\rho = 10\,\text{kg/m}$ and pick a value of κ large enough to cause a noticeable effect but not so large as to stop the oscillations. As a check, reverse the sign of κ and see if the wave grows in time (which would eventually violate our assumption of small oscillations). ▌

18.4 WAVES FOR 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, for instance, 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 will need to be extended. In addition, if the density increases, then so will the tension because it takes greater

tension to accelerate a greater mass. If gravity acts, then we will also expect the tension at the ends of the string to be higher than in the middle because the ends must support the entire weight of the string.

To derive the equation for wave motion with variable density and tension, consider again the element of a string (Figure 18.1 right) used in our derivation of the wave equation. If we do not assume the tension T is constant, then Newton's second law gives

$$F = ma \tag{18.26}$$

$$\Rightarrow \quad \frac{\partial}{\partial x}\left[T(x)\,\frac{\partial y(x,t)}{\partial x}\right]\Delta x = \rho(x)\Delta x\,\frac{\partial^2 u(x,t)}{\partial t^2} \tag{18.27}$$

$$\Rightarrow \quad \frac{\partial T(x)}{\partial x}\,\frac{\partial y(x,t)}{\partial x} + T(x)\,\frac{\partial^2 y(x,t)}{\partial x^2} = \rho(x)\,\frac{\partial^2 y(x,t)}{\partial t^2}. \tag{18.28}$$

If $\rho(x)$ and $T(x)$ are known functions, then these equations can be solved with just a small modification of our algorithm.

In §18.4.1 we will solve for the tension in a string due to gravity. Readers interested in an **alternate easier problem** that still shows the new physics may assume that the density and tension are proportional:

$$\rho(x) = \rho_0 e^{\alpha x}, \quad T(x) = T_0 e^{\alpha x}. \tag{18.29}$$

While we would expect the tension to be greater in regions of higher density (more mass to move and support), being proportional is clearly just an approximation. Substitution of these relations into (18.28) yields the new wave equation:

$$\frac{\partial^2 y(x,t)}{\partial x^2} + \alpha\,\frac{\partial y(x,t)}{\partial x} = \frac{1}{c^2}\,\frac{\partial^2 y(x,t)}{\partial t^2}, \quad c^2 = \frac{T_0}{\rho_0}. \tag{18.30}$$

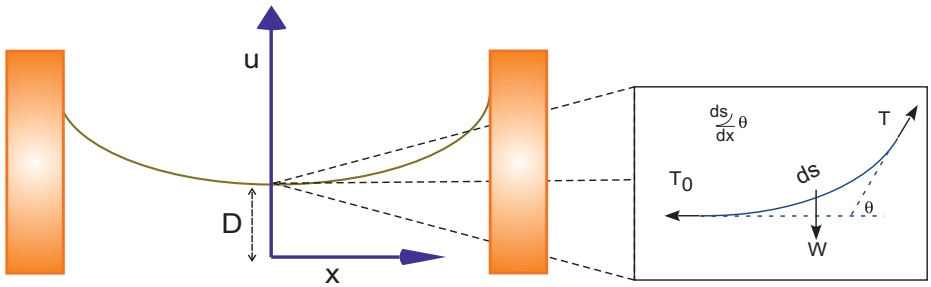
Here c is a constant that would be the wave velocity if $\alpha = 0$. This equation is similar to the wave equation with friction, only now the first derivative is with respect to x and not t. The corresponding difference equation follows from using central-difference approximations for the derivatives:

$$\begin{aligned} y_{i,j+1} &= 2y_{i,j} - y_{i,j-1} + \frac{\alpha c^2(\Delta t)^2}{2\Delta x}[y_{i+1,j} - y_{i,j}] + \frac{c^2}{c'^2}[y_{i+1,j} + y_{i-1,j} - 2y_{i,j}], \\ y_{i,2} &= y_{i,1} + \frac{c^2}{c'^2}[y_{i+1,1} + y_{i-1,1} - 2y_{i,1}] + \frac{\alpha c^2(\Delta t)^2}{2\Delta x}[y_{i+1,1} - y_{i,1}]. \end{aligned} \tag{18.31}$$

18.4.1 Waves on Catenary

Up until this point we have been ignoring the effect of gravity upon our string's shape and tension. This is a good approximation if there is very little sag in the string, as might happen if the tension is very high and the string is light. Even if there is some sag, our solution for $y(x,t)$ could be used as the disturbance about the equilibrium shape. However, if the string is massive, say, like a chain or heavy cable, then the sag in the middle caused by gravity could be quite large (Figure 18.5), and the resulting variation in shape and tension needs to be incorporated into the wave equation. Because the tension is no longer uniform, waves travel faster near the ends of the string, which are under greater tension since they must support the entire weight of the string.

Figure 18.5 *Left:* A uniform string suspended from its ends in a gravitational field assumes a catenary shape. *Right:* A force diagram of a section of the catenary at its lowest point. Because the ends of the string must support the entire weight of the string, the tension now varies along the string.



18.4.2 Derivation of Catenary Shape

Consider a string of uniform density ρ acted upon by gravity. To avoid confusion with our use of $y(x)$ to describe a disturbance on a string, we call $u(x)$ the equilibrium shape of the string (Figure 18.5). The statics problem we need to solve is to determine the shape $u(x)$ and the tension $T(x)$. The inset in Figure 18.5 is a free-body diagram of the midpoint of the string and shows that the weight W of this section of arc length s is balanced by the vertical component of the tension T. The horizonal tension T_0 is balanced by the horizontal component of T:

$$T(x)\sin\theta = W = \rho g s, \quad T(x)\cos\theta = T_0, \tag{18.32}$$

$$\Rightarrow \quad \tan\theta = \rho g s/T_0. \tag{18.33}$$

The trick is to convert (18.33) to a differential equation that we can solve. We do that by replacing the slope $\tan\theta$ by the derivative du/dx and taking the derivative with respect to x:

$$\frac{du}{dx} = \frac{\rho g}{T_0}s, \quad \Rightarrow \quad \frac{d^2u}{dx^2} = \frac{\rho g}{T_0}\frac{ds}{dx}. \tag{18.34}$$

Yet since $ds = \sqrt{dx^2 + du^2}$, we have our differential equation

$$\frac{d^2u}{dx^2} = \frac{1}{D}\frac{\sqrt{dx^2+du^2}}{dx} = \frac{1}{D}\sqrt{1+\left(\frac{du}{dx}\right)^2}, \tag{18.35}$$

$$D = T_0/\rho g, \tag{18.36}$$

where D is a combination of constants with the dimension of length. Equation (18.35) is the equation for the *catenary* and has the solution [Becker 54]

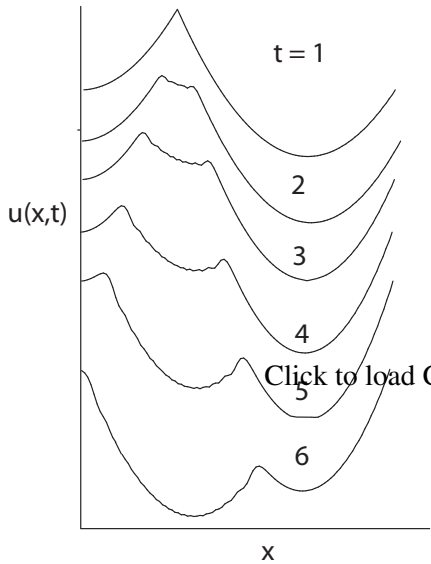
$$u(x) = D\cosh\frac{x}{D}. \tag{18.37}$$

Here we have chosen the x axis to lie a distance D below the bottom of the catenary (Figure 18.5) so that $x = 0$ is at the center of the string where $y = D$ and $T = T_0$. Equation (18.34) tells us the arc length $s = Ddu/dx$, so we can solve for $s(x)$ and, via (18.32), for the tension $T(x)$:

$$s(x) = D\sinh\frac{x}{D}, \quad \Rightarrow \quad T(x) = T_0\frac{ds}{dx} = \rho g u(x) = T_0\cosh\frac{x}{D}. \tag{18.38}$$

It is this variation in tension that causes the wave velocity to change for different positions on the string.

Figure 18.6 The wave motion of a plucked catenary with friction. (Courtesy of Juan Vanegas.)



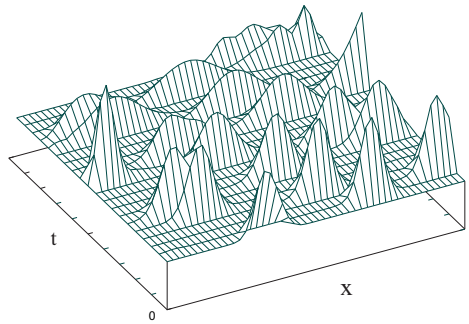
Click to load CatFrictionAnimate; permission may be pending?.

18.4.3 Catenary and Frictional Wave Exercises

We have given you the program **`EqStringAnimate.py`** (Listing 18.1) that solves the wave equation. Modify it to produce waves on a catenary including friction or for the assumed density and tension given by (18.29) with $\alpha = 0.5$, $T_0 = 40$ N, and $\rho_0 = 0.01$ kg/m. (The instructor's manual contains the programs **`CatFriction.py`** and **`CatString.py`** that do this.)

1. Look for some interesting 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 Figure 18.6.
3. **Normal modes:** Search for normal-mode solutions of the variable-tension wave equation, that is, solutions that vary as
$$u(x,t) = A\cos(\omega t)\sin(\gamma x).$$
Try using this form to start your program and see if you can find standing waves. Use large values for ω.
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 program; that is, build into your code the condition that for all times
$$y(x=0,t) = A\sin\omega t.$$
Try to vary A and ω until a normal mode (standing wave) is obtained.
5. (For the exponential density case.) If you were able to find standing waves, then verify that this string acts like a high-frequency filter, that is, that there is a frequency below which no waves occur.
6. For the catenary problem, plot 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 in the tension, we should be able

Figure 18.7 The position as a function of time of a localized electron confined to a square well (computed with the code `SqWell.py` available in the instructor's manual). The electron is initially on the right with a Gaussian wave packet. In time, the wave packet spreads out and collides with the walls.



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 in k.

18.5 UNIT II. QUANTUM WAVE PACKETS

Problem: An experiment places an electron with a definite momentum and position in a 1-D region of space the size of an atom. It is confined to that region by some kind of attractive potential. Your **problem** is to determine the resultant electron behavior in time and space.

18.6 TIME-DEPENDENT SCHRÖDINGER EQUATION (THEORY)

Because the region of confinement is the size of an atom, we must solve this problem quantum mechanically. Nevertheless, it is different from the problem of a particle confined to a box considered in Chapter 9, "Differential Equation Applications", because now we are starting with a particle of definite momentum and position. In Chapter 9 we had a time-independent situation in which we had to solve the eigenvalue problem. Now the definite momentum and position of the electron imply that the solution is a wave packet, which is not an eigenstate with a uniform time dependence of $\exp(-i\omega t)$. Consequently, we must now solve the time-dependent Schrödinger equation.

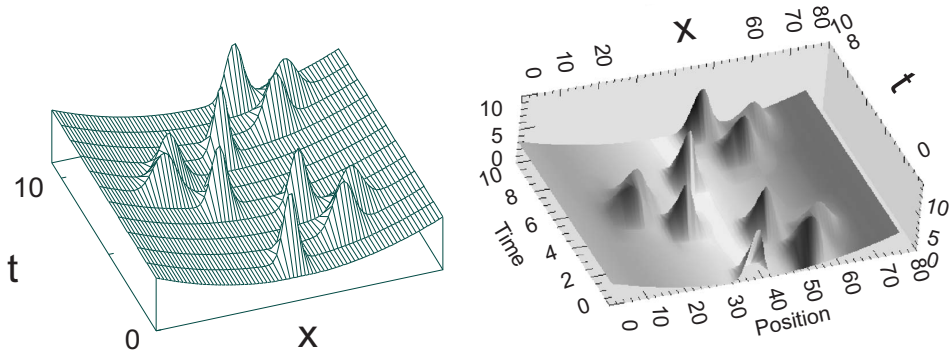
Applet We model an electron initially localized in space at $x = 5$ with momentum k_0 ($\hbar = 1$ in our units) by a wave function that is a wave packet consisting of a Gaussian multiplying a plane wave:

$$\psi(x, t = 0) = \exp\left[-\frac{1}{2}\left(\frac{x-5}{\sigma_0}\right)^2\right] e^{ik_0 x}. \tag{18.39}$$

To solve the **problem** we must determine the wave function for all later times. If (18.39) were an eigenstate of the Hamiltonian, its $\exp(-i\omega t)$ time dependence can be factored out of the Schrödinger equation (as is usually done in textbooks). However, $\tilde{H}\psi \neq E\psi$ for this ψ, and so we must solve the full time-dependent Schrödinger equation. To show you where we are going, the resulting wave packet behavior is shown in Figures 18.7 and 18.8.

The time and space evolution of a quantum particle is described by the 1-D time-

Figure 18.8 The probability density as a function of time for an electron confined to a 1-D harmonic oscillator potential well. On the left is a conventional surface plot from Gnuplot, while on the right is a color visualization from OpenDX of the same output.



dependent Schrödinger equation,

$$i\,\frac{\partial\psi(x,t)}{\partial t} = \tilde{H}\psi(x,t) \tag{18.40}$$

$$i\,\frac{\partial\psi(x,t)}{\partial t} = -\frac{1}{2m}\frac{\partial^2\psi(x,t)}{\partial x^2} + V(x)\psi(x,t), \tag{18.41}$$

where we have set $2m = 1$ to keep the equations simple. Because the initial wave function is complex (in order to have a definite 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:

$$\psi(x,t) = R(x,t) + i\,I(x,t), \tag{18.42}$$

$$\psi(x,t) = R(x,t) + i\,I(x,t), \tag{18.43}$$

$$\Rightarrow \quad \frac{\partial R(x,t)}{\partial t} = -\frac{1}{2m}\frac{\partial^2 I(x,t)}{\partial x^2} + V(x)I(x,t), \tag{18.44}$$

$$\frac{\partial I(x,t)}{\partial t} = +\frac{1}{2m}\frac{\partial^2 R(x,t)}{\partial x^2} - V(x)R(x,t), \tag{18.45}$$

where $V(x)$ is the potential acting on the particle.

18.6.1 Finite-Difference Algorithm

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) to a high level of precision for all time. For our project we use an *explicit* method that improves the numerical conservation of probability by solving for the real and imaginary parts of the wave function at slightly different or "staggered" times [Ask 77, Viss 91, MLP 00]. Explicitly, the real part R is determined at times 0, $\Delta t, \ldots$, and the imaginary part I at $\frac{1}{2}\Delta t$, $\frac{3}{2}\Delta t, \ldots$. The algorithm is based on (what else?) the Taylor expansions of R and I:

$$R\left(x, t + \frac{1}{2}\Delta t\right) = R\left(x, t - \frac{1}{2}\Delta t\right) + [4\alpha + V(x)\,\Delta t]I(x,t)$$

$$-2\alpha[I(x+\Delta x,t) + I(x-\Delta x,t)], \tag{18.46}$$

where $\alpha = \Delta t/2(\Delta x)^2$. In discrete form with $R^{t=n\Delta t}_{x=i\Delta x}$, we have

$$R_i^{n+1} = R_i^n - 2\left\{\alpha\left[I_{i+1}^n + I_{i-1}^n\right] - 2\left[\alpha + V_i\,\Delta t\right] I_i^n\right\}, \tag{18.47}$$

$$I_i^{n+1} = I_i^n + 2\left\{\alpha\left[R_{i+1}^n + R_{i-1}^n\right] - 2\left[\alpha + V_i\,\Delta t\right] R_i^n\right\}, \tag{18.48}$$

where the superscript n indicates the time and the subscript i the position.

The probability density ρ is defined in terms of the wave function evaluated at three different times:

$$\rho(t) = \begin{cases} R^2(t) + I\big(t + \frac{\Delta t}{2}\big) I\big(t - \frac{\Delta t}{2}\big), & \text{for integer } t, \\ I^2(t) + R\big(t + \frac{\Delta t}{2}\big) R\big(t - \frac{\Delta t}{2}\big), & \text{for half-integer } t. \end{cases} \tag{18.49}$$

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 ρ may seem strange, it reduces to the usual one for $\Delta t \to 0$ 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 [Koon 86, Viss 91] for details on the stability of the algorithm.

18.6.2 Wave Packet Implementation, Animation

In Listing 18.2 you will find the program **`HarmosAnimate.py`** that solves for the motion of the wave packet (18.39) inside a harmonic oscillator potential. The program **`Slit.py`** in the instructor's manual solves for the motion of a Gaussian wave packet as it passes through a slit (Figure 18.10). You should solve for a wave packet confined to the square well:

$$V(x) = \begin{cases} \infty, & x < 0, \text{ or } x > 15, \\ 0, & 0 \le x \le 15. \end{cases}$$

1. Define arrays **`psr[751][2]`** and **`psi[751][2]`** for the real and imaginary parts of ψ, and **`Rho[751]`** for the probability. 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 equation (18.39) for the initial wave packet to define **`psr[j][1]`** for all j at $t = 0$ and to define **`psi[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 (18.47) to compute **`psr[j][2]`** in terms of **`psr[j][1]`**, and (18.48) to compute **`psi[j][2]`** in terms of **`psi[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.

Listing 18.2 **HarmosAnimate.py** solves the time-dependent Schrödinger equation for a particle described by a Gaussian wave packet moving within a harmonic oscillator potential.

```
# HarmonsAnimate: Soltn of t-dependent Sch Eqt fro HO with animation
```

```
from visual import *

# initialize wave function, probability, potential
dx = 0.04;    dx2 = dx*dx;  k0 = 5.5*pi;  dt = dx2/20.0;  xmax = 6.0
xs = arange(-xmax,xmax+dx/2,dx)                    # array of x positions

g = display(width=500, height=250, title='Wave packet in HO Well')
PlotObj = curve(x=xs, color=color.yellow, radius=0.1)
g.center = (0,2,0)                                          # center of scene
                                              # initial condition; wave packet
psr = exp(-0.5*(xs/0.5)**2) * cos(k0*xs)           # Re wave function Psi
psi = exp(-0.5*(xs/0.5)**2) * sin(k0*xs)           # Im wave function Psi
v   = 15.0*xs**2

while True:
   rate(500)
   psr[1:-1] = psr[1:-1]-(dt/dx2)*(psi[2:]+psi[:-2]-2*psi[1:-1])+dt*v[1:-1]*psi[1:-1]
   psi[1:-1] = psi[1:-1]+(dt/dx2)*(psr[2:]+psr[:-2]-2*psr[1:-1])-dt*v[1:-1]*psr[1:-1]
   PlotObj.y = 4*(psr**2 + psi**2)
```

1. **Animation:** 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 Figure 18.7 or 18.8.
3. Make an animation showing the wave function as a function of time.
4. Check how well the 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.)

18.6.3 Wave Packets in Other Wells (Exploration)

Applet

1-D Well: Now confine the electron to lie within the harmonic oscillator potential:

$$V(x) = \frac{1}{2}x^2 \quad (-\infty \le x \le \infty).$$

Take the momentum $k_0 = 3\pi$, the space step $\Delta x = 0.02$, and the time step $\Delta t = \frac{1}{4}\Delta x^2$. Note that the wave packet broadens yet returns to its initial shape!

2-D Well ⊙**:** Now confine the electron to lie within a 2-D parabolic tube (Figure 18.9):

$$V(x,y) = 0.9x^2, \quad -9.0 \le x \le 9.0, \quad 0 \le y \le 18.0.$$

The extra degree of freedom means that we must solve the 2-D PDE:

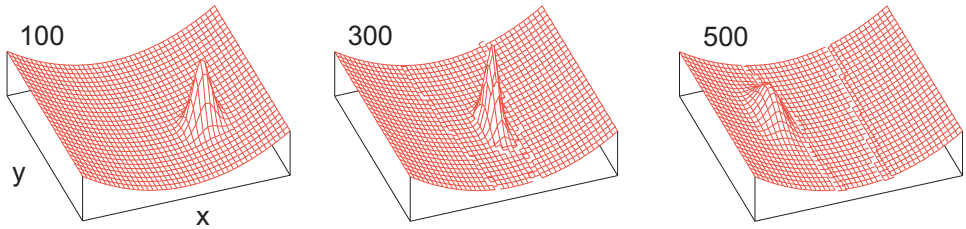
$$i\,\frac{\partial\psi(x,y,t)}{\partial t} = -\left(\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2}\right) + V(x,y)\psi. \tag{18.50}$$

Assume that the electron's initial localization is described by the 2-D Gaussian wave packet:

$$\psi(x,y,t=0) = e^{ik_{0x}x}\, e^{ik_{0y}y}\, \exp\left[-\frac{(x-x_0)^2}{2\sigma_0^2}\right] \exp\left[-\frac{(y-y_0)^2}{2\sigma_0^2}\right]. \tag{18.51}$$

Note that you can solve the 2-D equation by extending the method we just used in 1-D or you can look at the next section where we develop a special algorithm.

Figure 18.9 The probability density as a function of x and y of an electron confined to a 2-D parabolic tube (infinite in y direction). The electron is initially placed in a Gaussian wave packet in both the x and y directions, and it is to be noted how there is spreading of the wave packet in the y direction, but not in the x direction.



18.7 ALGORITHM FOR THE 2-D SCHRÖDINGER EQUATION

One way to develop an algorithm for solving the time-dependent Schrödinger equation in 2-D is to extend the 1-D algorithm to another dimension. Rather than do that, we apply quantum theory directly to obtain a more powerful algorithm [MLP 00]. First we note that equation (18.50) can be integrated in a formal sense [L&L,M 76] to obtain the operator solution:

$$\psi(x,y,t) = U(t)\psi(x,y,t=0) \tag{18.52}$$

$$U(t) = e^{-i\tilde{H}t}, \quad \tilde{H} = -\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) + V(x,y),$$

where $U(t)$ is an operator that translates a wave function by an amount of time t and $\tilde{H}$ is the Hamiltonian operator. From this formal solution we deduce that a wave packet can be translated ahead by time Δt via

$$\psi_{i,j}^{n+1} = U(\Delta t)\psi_{i,j}^{n}, \tag{18.53}$$

where the superscripts denote time $t = n\,\Delta t$ and the subscripts denote the two spatial variables $x = i\Delta x$ and $y = j\Delta y$. Likewise, the inverse of the time evolution operator moves the solution back one time step:

$$\psi^{n-1} = U^{-1}(\Delta t)\psi^{n} = e^{+i\tilde{H}\Delta t}\psi^{n}. \tag{18.54}$$

While it would be nice to have an algorithm based on a direct application of (18.54), the references show that the resulting algorithm is not stable. That being so, we base our algorithm on an indirect application [Ask 77], namely, the relation between the difference in ψ^{n+1} and ψ^{n-1}:

$$\psi^{n+1} = \psi^{n-1} + [e^{-i\tilde{H}\Delta t} - e^{+i\tilde{H}\Delta t}]\psi^{n}, \tag{18.55}$$

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,

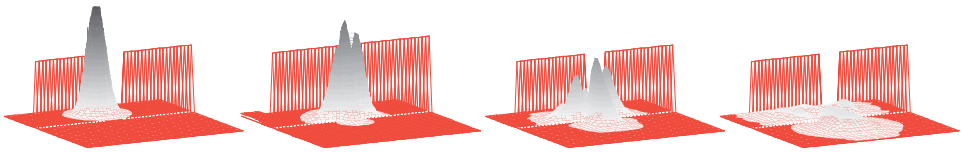
$$\frac{\partial^2\psi}{\partial x^2} \simeq -\frac{1}{2}\left[\psi_{i+1,j}^{n} + \psi_{i-1,j}^{n} - 2\psi_{i,j}^{n}\right], \tag{18.56}$$

with the corresponding-order expansion of the evolution equation (18.55). Substituting the resulting expression for the second derivative into the 2-D time-dependent Schrödinger equation results in[2]

$$\psi_{i,j}^{n+1} = \psi_{i,j}^{n-1} - 2i\left[\left(4\alpha + \tfrac{1}{2}\Delta t V_{i,j}\right)\psi_{i,j}^{n} - \alpha\left(\psi_{i+1,j}^{n} + \psi_{i-1,j}^{n} + \psi_{i,j+1}^{n} + \psi_{i,j-1}^{n}\right)\right],$$

[2]For reference sake, note that the constants in the equation change as the dimension of the equation changes; that is, there will be different constants for the 3-D equation, and therefore our constants are different from the references!

Figure 18.10 The probability density as a function of position and time for an electron incident upon and passing through a slit. Significant reflection is seen to occur.



where $\alpha = \Delta t/2(\Delta x)^2$. We convert this complex equations to coupled real equations by substituting in the wave function $\psi = R + iI$,

$$R_{i,j}^{n+1} = R_{i,j}^{n-1} + 2\left[\left(4\alpha + \tfrac{1}{2}\Delta t V_{i,j}\right) I_{i,j}^n - \alpha\left(I_{i+1,j}^n + I_{i-1,j}^n + I_{i,j+1}^n + I_{i,j-1}^n\right)\right],$$

$$I_{i,j}^{n+1} = I_{i,j}^{n-1} - 2\left[\left(4\alpha + \tfrac{1}{2}\Delta t V_{i,j}\right) R_{i,j}^n + \alpha\left(R_{i+1,j}^n + R_{i-1,j}^n + R_{i,j+1}^n + R_{i,j-1}^n\right)\right].$$

This is the algorithm we use to integrate the 2-D Schrödinger equation. To determine the probability, we use the same expression (18.49) used in 1-D.

18.7.1 Exploration: Bound & Diffracted 2-D Packet

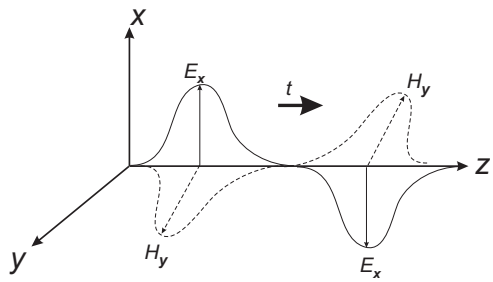
1. Determine the motion of a 2-D Gaussian wave packet within a 2-D harmonic oscillator potential:

$$V(x, y) = 0.3(x^2 + y^2), \quad -9.0 \le x \le 9.0, \quad -9.0 \le y \le 9.0. \tag{18.57}$$

2. Center the initial wave packet at $(x, y) = (3.0, -3)$ and give it momentum $(k_{0x}, k_{0y}) = (3.0, 1.5)$.
3. Young's single-slit experiment has a wave passing through a small slit with the transmitted wave showing interference effects. 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. Pass a Gaussian wave packet of width 3 through a slit of width 5 (Figure 18.10) and look for the resultant quantum interference.

Loading, pending permission, 2slits.mp4

Figure 18.11 A single electromagnetic pulse traveling along the z axis. The coupled E and H pulses are indicated by solid and dashed curves, respectively, and the pulses at different z values correspond to different times.



18.8 UNIT III. E&M WAVES VIA FINITE-DIFFERENCE TIME DOMAIN ⊙

Problem: You are given a region in space in which the E and H fields are known to have a sinusoidal spatial variation

$$E_x(z, t = 0) \ = 0.1 \sin \frac{2\pi z}{100}, \tag{18.58}$$

$$H_y(z, t = 0) \ = 0.1 \sin \frac{2\pi z}{100}, \qquad 0 \le z \le 200. \tag{18.59}$$

Determine the fields for all z values for all subsequent times.

Simulations of electromagnetic waves are of tremendous practical importance. Indeed, the fields of nanotechnology and spintronics rely heavily upon such simulations. The basic techniques used to solve for electromagnetic waves are essentially the same as those we used in Units I and II for string and quantum waves: Set up a grid in space and time and then step the initial solution forward in time one step at a time. For E&M *simulations, this technique is known as the* finite difference time domain *(FDTD) method. What is new for E&M waves is that they are vector fields, with the variations of one generating the other, so that the components of* **E** *and* **B** *are coupled to each other. Our treatment of FDTD does not do justice to the wealth of physics that can occur, and we recommend [Sull 00] for a more complete treatment and [Ward 04] (and their Web site) for modern applications.*

18.9 MAXWELL'S EQUATIONS

The description of electromagnetic (EM) waves via Maxwell's equations is given in many textbooks. For propagation in just one dimension (z) and for free space with no sinks or sources, four coupled PDEs result:

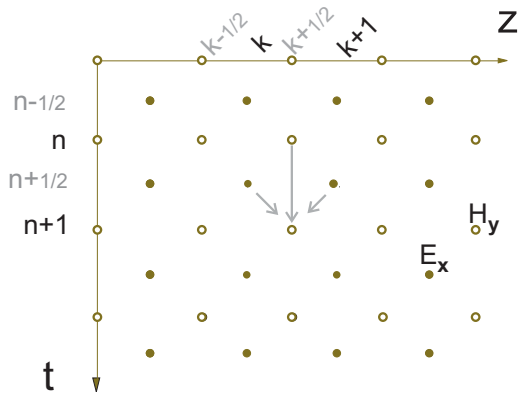
$$\vec{\nabla} \cdot \mathbf{E} = 0 \ \Rightarrow \ \frac{\partial E_x(z,t)}{\partial x} = 0 \tag{18.60}$$

$$\vec{\nabla} \cdot \mathbf{H} = 0 \ \Rightarrow \ \frac{\partial H_y(z,t)}{\partial y} = 0, \tag{18.61}$$

$$\frac{\partial \mathbf{E}}{\partial t} = +\frac{1}{\epsilon_0}\vec{\nabla} \times \mathbf{H} \ \Rightarrow \ \frac{\partial E_x}{\partial t} = -\frac{1}{\epsilon_0}\frac{\partial H_y(z,t)}{\partial z}, \tag{18.62}$$

$$\frac{\partial \mathbf{H}}{\partial t} = -\frac{1}{\mu_0}\vec{\nabla} \times \mathbf{E} \ \Rightarrow \ \frac{\partial H_y}{\partial t} = -\frac{1}{\mu_0}\frac{\partial E_x(z,t)}{\partial z}. \tag{18.63}$$

Figure 18.12 The algorithm for using the known values of E_x and H_y at three earlier times and three different space positions to obtain the solution at the present time. Note that the values of E_x are determined on the lattice of filled circles, corresponding to integer space indices and half-integer time indices. In contrast, the values of H_y are determined on the lattice of open circles, corresponding to half-integer space indices and integer time indices.



As indicated in Figure 18.11, we have chosen the electric field $\mathbf{E}(z,t)$ to oscillate (be polarized) in the x direction and the magnetic field $\mathbf{H}(z,t)$ to be polarized in the y direction. As indicated by the bold arrow in Figure 18.11, the direction of power flow for the assumed transverse electromagnetic (TEM) wave is given by the right-hand rule for $\mathbf{E}\times\mathbf{H}$. Note that although we have set the initial conditions such that the EM wave is traveling in only one dimension (z), its electric field oscillates in a perpendicular direction (x) and its magnetic field oscillates in yet a third direction (y); so while some may call this a 1-D wave, the vector nature of the fields means that the wave occupies all three dimensions.

18.10 FDTD ALGORITHM

We need to solve the two coupled PDEs (18.62) and (18.63) appropriate for our problem. As is usual for PDEs, we approximate the derivatives via the central-difference approximation, here in both time and space. For example,

$$\frac{\partial E(z,t)}{\partial t} \simeq \frac{E(z,t+\frac{\Delta t}{2}) - E(z,t-\frac{\Delta t}{2})}{\Delta t}, \tag{18.64}$$

$$\frac{\partial E(z,t)}{\partial z} \simeq \frac{E(z+\frac{\Delta z}{2},t) - E(z-\frac{\Delta z}{2},t)}{\Delta z}. \tag{18.65}$$

We next substitute the approximations into Maxwell's equations and rearrange the equations into the form of an algorithm that advances the solution through time. Because only first derivatives occur in Maxwell's equations, the equations are simple, although the electric and magnetic fields are intermixed.

As introduced by Yee [Yee 66], we set up a space-time lattice (Figure 18.12) in which there are half-integer time steps as well as half-integer space steps. The magnetic field will be determined at integer time sites and half-integer space sites (open circles), and the electric field will be determined at half-integer time sites and integer space sites (filled circles). While this is an extra level of complication, the transposed lattices do lead to an accurate and robust algorithm. Because the fields already have subscripts indicating their vector nature, we indicate the lattice position as superscripts, for example,

$$E_x(z,t) \to E_x(k\Delta z, n\Delta t) \to E_x^{k,n}. \tag{18.66}$$

Maxwell's equations (18.62) and (18.63) now become the discrete equations

$$\frac{E_x^{k,n+1/2} - E_x^{k,n-1/2}}{\Delta t} = -\frac{H_y^{k+1/2,n} - H_y^{k-1/2,n}}{\epsilon_0 \Delta z},$$

$$\frac{H_y^{k+1/2,n+1} - H_y^{k+1/2,n}}{\Delta t} = -\frac{E_x^{k+1,n+1/2} - E_x^{k,n+1/2}}{\mu_0 \Delta z}.$$

To repeat, this formulation solves for the electric field at integer space steps (k) but half-integer time steps (n), while the magnetic field is solved for at half-integer space steps but integer time steps.

We convert these equations into two simultaneous algorithms by solving for E_x at time $n + \frac{1}{2}$, and H_y at time n:

$$E_x^{k,n+1/2} = E_x^{k,n-1/2} - \frac{\Delta t}{\epsilon_0 \Delta z}\left(H_y^{k+1/2,n} - H_y^{k-1/2,n}\right), \tag{18.67}$$

$$H_y^{k+1/2,n+1} = H_y^{k+1/2,n} - \frac{\Delta t}{\mu_0 \Delta z}\left(E_x^{k+1,n+1/2} - E_x^{k,n+1/2}\right). \tag{18.68}$$

The algorithms must be applied simultaneously because the space variation of H_y determines the time derivative of E_x, while the space variation of E_x determines the time derivative of H_y (Figure 18.12). This algorithm is more involved than our usual time-stepping ones in that the electric fields (filled circles in Figure 18.12) at future times $t = n + \frac{1}{2}$ are determined from the electric fields at one time step earlier $t = n - \frac{1}{2}$, and the magnetic fields at half a time step earlier $t = n$. Likewise, the magnetic fields (open circles in Figure 18.12) at future times $t = n + 1$ are determined from the magnetic fields at one time step earlier $t = n$, and the electric field at half a time step earlier $t = n + \frac{1}{2}$. In other words, it is as if we have two interleaved lattices, with the electric fields determined for half-integer times on lattice 1 and the magnetic fields at integer times on lattice 2.

Although these half-integer times appear to be the norm for FDTD methods [Taf 89, Sull 00], it may be easier for some readers to understand the algorithm by doubling the index values and referring to even and odd times:

$$E_x^{k,n} = E_x^{k,n-2} - \frac{\Delta t}{\epsilon_0 \Delta z}\left(H_y^{k+1,n-1} - H_y^{k-1,n-1}\right), \quad k \text{ even, odd}, \tag{18.69}$$

$$H_y^{k,n} = H_y^{k,n-2} - \frac{\Delta t}{\mu_0 \Delta z}\left(E_x^{k+1,n-1} - E_x^{k-1,n-1}\right), \quad k \text{ odd, even}. \tag{18.70}$$

This makes it clear that E is determined for even space indices and odd times, while H is determined for odd space indices and even times.

We simplify the algorithm and make its stability analysis simpler by normalizing the electric fields to have the same dimensions as the magnetic fields,

$$\tilde{E} = \sqrt{\frac{\epsilon_0}{\mu_0}} E. \tag{18.71}$$

The algorithm (18.67) and (18.68) now becomes

$$\tilde{E}_x^{k,n+1/2} = \tilde{E}_x^{k,n-1/2} + \beta\left(H_y^{k-1/2,n} - H_y^{k+1/2,n}\right), \tag{18.72}$$

$$H_y^{k+1/2,n+1} = H_y^{k+1/2,n} + \beta\left(\tilde{E}_x^{k,n+1/2} - \tilde{E}_x^{k+1,n+1/2}\right), \tag{18.73}$$

$$\beta = \frac{c}{\Delta z/\Delta t}, \quad c = \frac{1}{\sqrt{\epsilon_0 \mu_0}}. \tag{18.74}$$

Here c is the speed of light in a vacuum and β is the ratio of the speed of light to grid velocity $\Delta z/\Delta t$.

The space step Δz and the time step Δt must be chosen so that the algorithm is stable. The scales of the space and time dimensions are set by the wavelength and frequency, respectively, of the propagating wave. As a minimum, we want at least 10 grid points to fall within a wavelength:

$$\Delta z \leq \frac{\lambda}{10}. \tag{18.75}$$

The time step is then determined by the Courant stability condition [Taf 89, Sull 00] to be

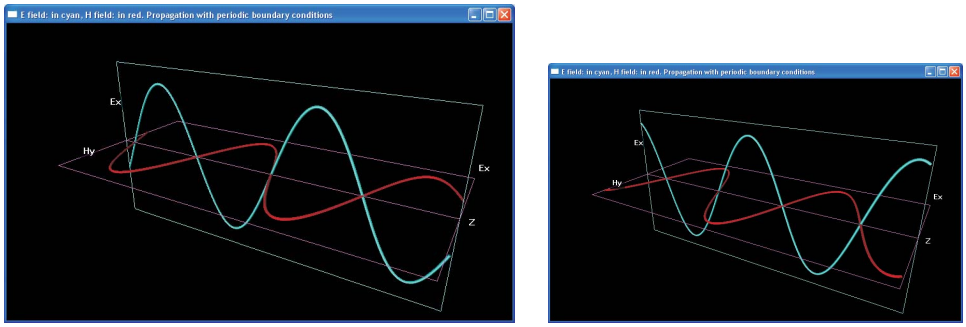
$$\beta = \frac{c}{\Delta z/\Delta t} \leq \frac{1}{2}. \tag{18.76}$$

As we have seen before, (18.76) implies that making the time step smaller improves precision and maintains stability, but making the space step smaller must be accompanied by a simultaneous decrease in the time step in order to maintain stability (you should check this).

Listing 18.3 **FDTD.py** solves Maxwell's equations via FDTD time stepping (finite-difference time domain) for linearly polarized wave propagation in the z direction in free space.

```
# FDTD.py  FDTD solution of Maxwell's equations in 1-D

from visual import *
xmax=201
ymax=100
zmax=100
scene = display(x=0,y=0,width= 800, height= 500, \
               title= 'E: cyan, H: red. Periodic BC',forward=(-0.6,-0.5,-1))
Efield = curve(x=list(range(0,xmax)),color=color.cyan,radius=1.5,display=scene)
Hfield = curve(x=list(range(0,xmax)),color=color.red, radius=1.5,display=scene)
vplane= curve(pos=[(-xmax,ymax),(xmax,ymax),(xmax,-ymax),(-xmax,-ymax),
                   (-xmax,ymax)],color=color.cyan)
zaxis=curve(pos=[(-xmax,0),(xmax,0)],color=color.magenta)
hplane=curve(pos=[(-xmax,0,zmax),(xmax,0,zmax),(xmax,0,-zmax),(-xmax,0,-zmax),
                   (-xmax,0,zmax)],color=color.magenta)
ball1 = sphere(pos = (xmax+30, 0,0), color = color.black, radius = 2)
ts = 2                          # for old and new time
beta = 0.01
Ex = zeros((xmax,ts),float)      # init E, 201 points, ts=0 old, ts=1 new
Hy = zeros((xmax,ts),float)      # init H, 201 points, ts=0 old, ts=1 new
Exlabel1 = label( text = 'Ex', pos = (-xmax-10, 50), box = 0 )
Exlabel2 = label( text = 'Ex', pos = (xmax+10, 50), box = 0 )
Hylabel  = label( text = 'Hy', pos = (-xmax-10, 0,50), box = 0 )
zlabel   = label( text = 'Z',  pos = (xmax+10, 0), box = 0 )  # shift fig
ti=0                                # t=0: initial position, t=1 next time

def inifields():
    k = arange(xmax)
    Ex[:xmax,0] = 0.1*sin(2*pi*k/100.0)
    Hy[:xmax,0] = 0.1*sin(2*pi*k/100.0)

def plotfields(ti):                                  # screen coordinates
    k = arange(xmax)
    Efield.x = 2*k-xmax                          # world to screen coords
    Efield.y = 800*Ex[k,ti]
    Hfield.x = 2*k-xmax                          # world to screen coords
    Hfield.z = 800*Hy[k,ti]

inifields()                                                # initial time
plotfields(ti)
while True:
    rate(600)
    Ex[1:xmax-1,1] = Ex[1:xmax-1,0] + beta*(Hy[0:xmax-2,0]-Hy[2:xmax,0])
    Hy[1:xmax-1,1] = Hy[1:xmax-1,0] + beta*(Ex[0:xmax-2,0]-Ex[2:xmax,0])
    Ex[0,1]        = Ex[0,0]        + beta*(Hy[xmax-2,0]  -Hy[1,0])  # BC
    Ex[xmax-1,1]   = Ex[xmax-1,0]   + beta*(Hy[xmax-2,0]  -Hy[1,0])
    Hy[0,1]        = Hy[0,0]        + beta*(Ex[xmax-2,0]  -Ex[1,0])  # BC
    Hy[xmax-1,1]   = Hy[xmax-1,0]   + beta*(Ex[xmax-2,0] - Ex[1,0])
    plotfields(ti)                                     # plot new fields
    Ex[:xmax,0] = Ex[:xmax,1]                           # next iteration
    Hy[:xmax,0] = Hy[:xmax,1]                                # old = new
```

Figure 18.13 The E field (cyan) and the H field (red) at the intial time (left) and at a later time (right). Periodic boundary conditions are used at the ends of the spatial region, which means that the large z wave continues into the $z = 0$ wave.



18.10.1 Implementation

In Listing 18.3 we provide a simple implementation of the FDTD algorithm for a z lattice of 200 sites. The initial conditions correspond to a sinusoidal variation of the E and H fields for all z values in for $0 \leq z \leq 200$:

$$E_x(z, t = 0) \;=\; 0.1 \sin \frac{2\pi z}{100}, \qquad H_y(z, t = 0) \;=\; 0.1 \sin \frac{2\pi z}{100}, \tag{18.77}$$

The algorithm then steps out in time for as long as the user desires. The discrete form of Maxwell equations used are:

```
Ex[k, 1] = Ex[k, 0] + beta* (Hy[k-1, 0] - Hy[k+1, 0])      (18.78)
Hy[k, 1] = Hy[k, 0] + beta* (Ex[k-1, 0] - Ex[k+1, 0])      (18.79)
```

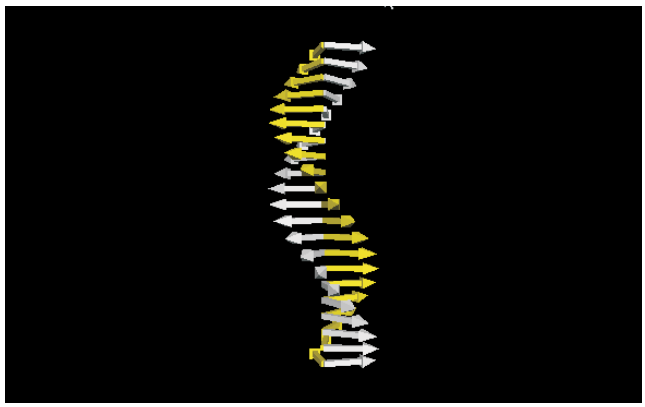
where $1 \leq k \leq 200$, and beta is a constant. The second index takes the values 0 and 1, with 0 being the old time and 1 the new. At the end of each iteration the new field throughout all of space becomes the old one, and a new, new one is computed. With this algorithm, the spatial endpoints `k=0` and `k=xmax-1` remain undefined. We define them by assuming periodic boundary conditions:

```
     Ex[0, 1] = Ex[0, 0] + beta* (Hy[xmax-2, 0] - Hy[1,0])      (18.80)
Ex[xmax-1, 1] = Ex[xmax-1, 0] + beta* (Hy[xmax-2, 0] - Hy[1,0])      (18.81)
     Hy[0, 1] = Hy[0, 0] + beta* (Ex[xmax-2, 0] - Ex[1,0])      (18.82)
Hy[xmax-1, 1] = Hy[xmax-1, 0] + beta* (Ex[xmax-2, 0] - Ex[1,0])      (18.83)
```

18.10.2 Assessment

1. Impose boundary conditions such that all fields vanish on the boundaries. Compare the solutions so obtained to those without explicit conditions for times less than and greater than those at which the pulses hit the walls.
2. Examine the stability of the solution for different values of Δz and Δt and thereby test the Courant condition (18.76).
3. Extend the algorithm to include the effect of entering, propagating through, and exiting a dielectric material placed within the z integration region.
   a. Ensure that you see both transmission and reflection at the boundaries.
   b. Investigate the effect of varying the dielectric's index of refraction.

Figure 18.14 E and H fields at $t = 100$ for a circularly polarized wave in free space.



4. The direction of propagation of the pulse is given $\mathbf{E} \times \mathbf{H}$, which depends on the relative phase between the E and H fields. (With no initial $\mathbf{H}$ field, we obtain pulses both to the right and the left.)
   a. Modify the program so that that there is an initial H pulse as well as an initial E pulse, both with a Gaussian times a sinusoidal shape.
   b. Verify that the direction of propagation changes if the E and H fields have relative phases of 0 or π.
5. Investigate the resonator modes of a wave guide by picking the initial conditions corresponding to plane waves with nodes at the boundaries.
6. Investigate standing waves with wavelengths longer than the size of the integration region.
7. Simulate unbounded propagation by building in periodic boundary conditions into the algorithm.
8. $\odot$ Place a medium with periodic permittivity in the integration volume. This should act as a frequency-dependent filter, which does not propagate certain frequencies at all.

18.10.3 Extension: Circularly Polarized Waves

We now extend our treatment to EM waves in which the $\mathbf{E}$ and $\mathbf{H}$ fields, while still transverse and propagating in the z direction, are not restricted to linear polarizations along just one axis. Accordingly, we add to (18.62) and (18.63):

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu_0}\frac{\partial E_y}{\partial z}, \tag{18.84}$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\epsilon_0}\frac{\partial H_x}{\partial z}. \tag{18.85}$$

When discretized in the same way as (18.67) and (18.68), we obtain

$$H_x^{k+1/2,n+1} = H_x^{k+1/2,n} + \frac{\Delta t}{\mu_0\,\Delta z}\big(E_y^{k+1,n+1/2} - E_y^{k,n+1/2}\big), \tag{18.86}$$

$$E_y^{k,n+1/2} = E_y^{k,n-1/2} + \frac{\Delta t}{\epsilon_0\,\Delta z}\big(H_y^{k+1/2,n} - H_y^{k-1/2,n}\big). \tag{18.87}$$

To produce a circularly polarized traveling wave, we set the initial conditions:

$$E_x = \cos\left(t - \frac{z}{c} + \phi_y\right), \quad H_x = \sqrt{\frac{\epsilon_0}{\mu_0}} \cos\left(t - \frac{z}{c} + \phi_y\right), \tag{18.88}$$

$$E_y = \cos\left(t - \frac{z}{c} + \phi_x\right), \quad H_y = \sqrt{\frac{\epsilon_0}{\mu_0}} \cos\left(t - \frac{z}{c} + \phi_x + \pi\right). \tag{18.89}$$

We take the phases to be $\phi_x = \pi/2$ and $\phi_y = 0$, so that their difference $\phi_x - \phi_y = \pi/2$, which leads to circular polarization. We include the initial conditions in the same manner as we did the Gaussian pulse, only now with these cosine functions.

Listing 18.4 gives our implementation **`EMcirc.py`** for waves with transverse two-component **E** and **H** fields. Some results of the simulation are shown in Figure 18.14, where you will note the difference in phase between **E** and **H**.

Listing 18.4 **CircPolartzn.py** solves Maxwell's equations via FDTD time-stepping for circularly polarized wave propagation in the z direction in free space.

```
# CircPolarztn.py: solves Maxwell eqs. using FDTD
# initial E and H components = 200 cosines
# The same space region is observed as time progresses

from visual import *                          # graphics and math classes

scene = display(x = 0,y = 0,width = 600,height = 400,range = 200,
              title='Circular polarization, E field in white, H field in yellow')
global phy,pyx
max = 201
                                                                  # c= (c0/dz)*dt
c = 0.01                # Courant stability condition, unstable for c>0.1

Ex = zeros((max+2,2),float)                          # Ex and Hy components
Hy = zeros((max+2,2),float)
Ey = zeros((max+2,2),float)                          # Ey and Hx components
Hx = zeros((max+2,2),float)

arrowcol= color.white                                    # E in white color
Earrows = []
Harrows = []
for i in range(0,max,10):       # xyz reference system different for plots
    Earrows.append(arrow(pos=(0,i-100,0), axis=(0,0,0), color=arrowcol))
    Harrows.append(arrow(pos=(0,i-100,0), axis=(0,0,0), color=color.yellow))

def plotfields(Ex,Ey,Hx,Hy):
    for n, arr in enumerate(Earrows):
        arr.axis = (35*Ey[10*n,1],0,35*Ex[10*n,1])
    for n, arr in enumerate(Harrows):
        arr.axis = (35*Hy[10*n,1],0,35*Hx[10*n,1])

def inifields():                      # Initial values for E and H fields
    phx = 0.5*pi
    phy = 0.0
    k = arange(0,max)
    Ex[:-2,0] = cos(-2*pi*k/200 + phx)
    Ey[:-2,0] = cos(-2*pi*k/200 + phy)
    Hx[:-2,0] = cos(-2*pi*k/200 + phy + pi)
    Hy[:-2,0] = cos(-2*pi*k/200 + phx)

def newfields():
    while True:                                                # Time steps
       rate(1000)
       Ex[1:max-1,1] =  Ex[1:max-1,0]+c*(Hy[:max-2,0]-Hy[2:max,0]) # New
       Ey[1:max-1,1] =  Ey[1:max-1,0] + c*(Hx[2:max,0]-Hx[:max-2,0])
       Hx[1:max-1,1] =  Hx[1:max-1,0] + c*(Ey[2:max,0]-Ey[:max-2,0])# New
       Hy[1:max-1,1] =  Hy[1:max-1,0] + c*(Ex[:max-2,0]-Ex[2:max,0])
       Ex[0,1]    = Ex[0,0]    + c*(Hy[200-1,0]-Hy[1,0])       # periodic BC
       Ex[200,1] = Ex[200,0] + c*(Hy[200-1,0]-Hy[1,0])    # BC first point
       Ey[0,1]    = Ey[0,0]    + c*(Hx[1,0]-  Hx[200-1,0])   # BC last points
       Ey[200,1] = Ey[200,0] + c*(Hx[1,0]-  Hx[200-1,0])
       Hx[0,1]    = Hx[0,0]    + c*(Ey[1,0]-  Ey[200-1,0])
       Hx[200,1] = Hx[200,0] + c*(Ey[1,0]-  Ey[200-1,0])
       Hy[0,1]    = Hy[0,0]    + c*(Ex[200-1,0]-Ex[1,0])
       Hy[200,1] = Hy[200,0] + c*(Ex[200-1,0]-Ex[1,0])
```

```
        plotfields(Ex,Ey,Hx,Hy)

        Ex[:max,0] = Ex[:max,1]                    # update fields old=new
        Ey[:max,0] = Ey[:max,1]
        Hx[:max,0] = Hx[:max,1]
        Hy[:max,0] = Hy[:max,1]

inifields()                              # Initial field components at t=0
newfields()                               # subsequent evolution of fields
```

Chapter Nineteen

Solitons & Computational Fluid Dynamics

In Unit I of this chapter we discuss shallow-water soliton waves. This extends the discussion of waves in Chapter 18, "PDE Waves: String, Quantum Packet, and E&M," *by progressively including nonlinearities, dispersion, and hydrodynamic effects. In Unit II we confront the more general equations of computational fluid dynamics (CFD) and their solutions.*[1] *The mathematical description of the motion of fluids, though not a new subject, remains a challenging one. The equations are complicated and nonlinear, there are many degrees of freedom, the nonlinearities may lead to instabilities, analytic solutions are rare, and the boundary conditions for realistic geometries (like airplanes) are not intuitive. These difficulties may explain why fluid dynamics is often absent from undergraduate and even graduate physics curricula. Nonetheless, as an essential element of the real world that also has tremendous practical importance, we encourage its study. We recommend [F&W 80, L&L,F 87] for those interested in the derivations, and [Shaw 92] for more details about the computations.*

VIDEO LECTURES, APPLETS AND ANIMATIONS FOR THIS CHAPTER

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Shocks & Solitons	pdf	19.3	Computational Fluid Dynamics	pdf	19.6

Applets and Animations Applet

Name	*Sections*	*Name*	*Sections*
Solitons	19.5	2-D Solitons	19.5
Smoothed Particle Hydrodynamics	19.6–19.9	Shock Waves	19.3

19.1 UNIT I. ADVECTION, SHOCKS, RUSSELL'S SOLITON

In 1834, J. Scott Russell observed on the Edinburgh-Glasgow canal [Russ 44]:

> 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

[1] We acknowledge some helpful reading of Unit I by Satoru S. Kano.

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 Figure 19.2 we see a single steplike wave breaking up into approximately eight of these solitary waves (also called *solitons*). These eight solitons occur so frequently that some consider them the normal modes for this nonlinear system. 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

$$c^2 = g(h + A), \tag{19.1}$$

where g is the acceleration due to gravity. Equation (19.1) implies an effect not found for linear systems, namely, that waves with greater amplitudes A travel faster than those with smaller amplitudes. Observe that this is similar to the formation of shock waves but different from *dispersion* in which waves of different wavelengths have different velocities. The dependence of c on amplitude A is illustrated in Figure 19.3, where we see a taller soliton catching up with and passing through a shorter one.

Problem: Explain Russell's observations and see if they relate to the formation of *tsunamis*. The latter are ocean waves that form from sudden changes in the level of the ocean floor and then travel over long distances without dispersion or attenuation until they wreak havoc on a distant shore.

19.2 THEORY: CONTINUITY AND ADVECTION EQUATIONS

The motion of a fluid is described by the continuity equation and the Navier–Stokes equation [L&L,M 76]. We will discuss the former here and the latter in §19.7 of Unit II. The continuity equation describes conservation of mass:

$$\frac{\partial \rho(\mathbf{x}, t)}{\partial t} + \vec{\nabla} \cdot \mathbf{j} = 0, \quad \mathbf{j} \stackrel{\text{def}}{=} \rho \mathbf{v}(\mathbf{x}, t). \tag{19.2}$$

Here $\rho(\mathbf{x}, t)$ is the mass density, $\mathbf{v}(\mathbf{x}, t)$ is the velocity of the fluid, and the product $\mathbf{j} = \rho\mathbf{v}$ is the mass current. As its name implies, the divergence $\vec{\nabla} \cdot \mathbf{j}$ describes the spreading of the current in a region of space, as might occur if there were a current source. Physically, the continuity equation (19.2) states that changes in the density of the fluid within some region of space arise from the flow of current in and out of that region.

For 1-D flow in the x direction and for a fluid that is moving with a constant velocity $v = c$, the continuity equation (19.2) takes the simple form

$$\frac{\partial \rho}{\partial t} + \frac{\partial (c\rho)}{\partial x} = 0, \tag{19.3}$$

$$\frac{\partial \rho}{\partial t} + c\frac{\partial \rho}{\partial x} = 0. \tag{19.4}$$

This equation is known as the *advection equation*, where the term "advection" is used to describe the horizontal transport of a conserved quantity from one region of space to another due to a velocity field. For instance, advection describes dissolved salt transported in water.

The advection equation looks like a first-derivative form of the wave equation, and indeed, the two are related. A simple substitution proves that any function with the form of a

traveling wave,

$$u(x,t) = f(x - ct), \tag{19.5}$$

will be a solution of the advection equation. If we consider a surfer riding along the crest of a traveling wave, that is, remaining at the same position relative to the wave's shape as time changes, then the surfer does not see the shape of the wave change in time, which implies that

$$x - ct = \text{constant} \quad \Rightarrow \quad x = ct + \text{constant}. \tag{19.6}$$

The speed of the surfer is therefore $dx/dt = c$, which is a constant. Any function $f(x - ct)$ is clearly a traveling wave solution in which an arbitrary pulse is carried along by the fluid at velocity c without changing shape.

19.2.1 Advection Implementation

Although the advection equation is simple, trying to solve it by a simple differencing scheme (the leapfrog method) may lead to unstable numerical solutions. As we shall see when we look at the nonlinear version of this equation, there are better ways to solve it. Listing 19.1 presents our code for solving the advection equation using the Lax–Wendroff method (a better method).

19.3 THEORY: SHOCK WAVES VIA BURGERS' EQUATION

In a later section we will examine use of the KdeV equation to describe Russell's solitary waves. In order to understand the physics contained in that equation, we study some terms in it one at a time. To start, consider Burgers' equation [Burg 74]:

$$\frac{\partial u}{\partial t} + \epsilon u \frac{\partial u}{\partial x} = 0, \tag{19.7}$$

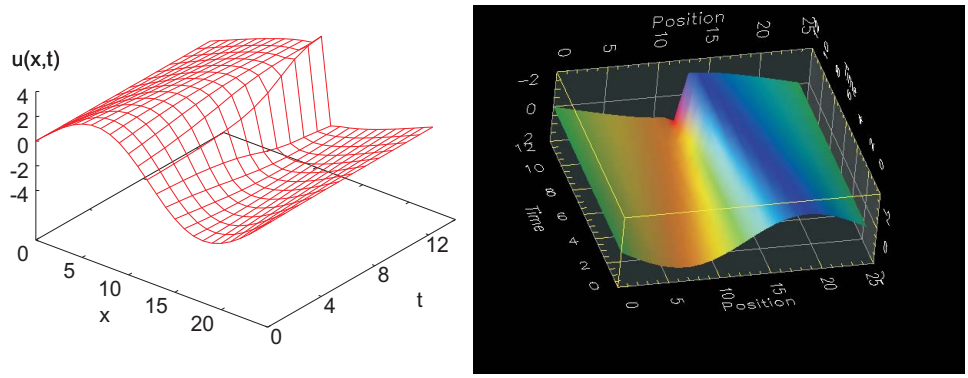
$$\frac{\partial u}{\partial t} + \epsilon \frac{\partial (u^2/2)}{\partial x} = 0, \tag{19.8}$$

where the second equation is the *conservative form*.This equation can be viewed as a variation on the advection equation (19.4) in which the wave speed $c = \epsilon u$ is proportional to the amplitude of the wave, as Russell found for his waves. The second nonlinear term in Burgers' equation leads to some unusual behaviors. Indeed, John von Neumann studied this equation as a simple model for turbulence [F&S].

Listing 19.1 **AdvecLax.py** solves the advection equation via the Lax–Wendroff scheme.

```
# AdvecLax.py: Solve advection eqnt via Lax - Wendroff scheme

from visual.graph import *

m = 100;  c = 1.;          dx = 1./m;      beta = 0.8       # beta = c*dt/dx
u = zeros((m+1),float); u0 = zeros((m+1), float); uf = zeros((m+1),float)
dt = beta*dx/c;            T_final = 0.5;              n = int(T_final/dt)


graph1 = gdisplay(width=600, height=500,
   title='Advec Eqn: Initial (red), Exact (blue), Lax-Wendroff (yellow)',
   xtitle = 'x', ytitle = 'u(x), Blue=exact, Yellow=Sim',
   xmin=0,xmax=1,ymin=0,ymax=1)
initfn = gcurve(color = color.red)                    # Initial function
exactfn = gcurve(color = color.blue)                    # Exact function
numfn = gcurve(color = color.yellow)                 # Numerical function

for i in range(0, m):                        # Initial and exact functions
    x = i*dx
    u0[i] = exp( - 300.* (x - 0.12)**2)            # Gaussian initial data
    initfn.plot(pos = (0.01*i, u0[i]) )
    uf[i] = exp( - 300.*(x - 0.12 - c*T_final)**2 ) # Exact in blue color
    exactfn.plot(pos = (0.01*i, uf[i]) )
    rate(20)
```

Figure 19.1 Wave height *versus* position for increasing times showing the formation of a shock wave (sharp edge) from an initial sine wave. The visualization on the left uses Gnuplot and that on the right OpenDX.




```
for j in range(0, n+1):
    for i in range(0, m − 1):
        u[i + 1] = (1.−beta∗beta)∗u0[i+1]−(0.5∗beta)∗(1.−beta)∗u0[i+2] \
                   +(0.5∗beta)∗(1. + beta)∗u0[i]  # Lax − Wendroff scheme
        u[0] = 0.
        u[m − 1] = 0.
        u0[i] = u[i]
for j in range(0, m−1 ):
    rate(30)
    numfn.plot(pos = (0.01∗j, u[j]) )                    # Solution
```

In the advection equation (19.4), all points on the wave move at the same speed c, and so the shape of the wave remains unchanged in time. In Burgers' equation (19.7), the points on the wave move ("advect") themselves such that the local speed depends on the local wave's amplitude, with the high parts of the wave moving progressively faster than the low parts. This changes the shape of the wave in time; if we start with a wave packet that has a smooth variation in height, the high parts will speed up and push their way to the front of the packet, thereby forming a sharp leading edge known as a *shock wave* [Tab 89]. A shock wave solution to Burgers' equation with $\epsilon = 1$ is shown in Figure 19.1.

19.3.1 Lax–Wendroff Algorithm for Burgers' Equation

We first solve Burgers' equation (19.4) via the usual approach in which we express the derivatives as central differences. This leads to a leapfrog scheme for the future solution in terms of present and past ones:

$$u(x, t + \Delta t) = u(x, t - \Delta t) - \beta \left[\frac{u^2(x + \Delta x, t) - u^2(x - \Delta x, t)}{2} \right],$$

$$u_{i,j+1} = u_{i,j-1} - \beta \left[\frac{u^2_{i+1,j} - u^2_{i-1,j}}{2} \right], \beta = \frac{\epsilon}{\Delta x / \Delta t}. \tag{19.9}$$

Here u^2 is the square of u and is not its second derivative, and β is a ratio of constants known as the *Courant–Friedrichs–Lewy* (CFL) *number*. As you should prove for yourself, $\beta < 1$ is required for stability.

While we have used a leapfrog method with success in the past, its low-order approximation for the derivative becomes inaccurate when the gradients can get large, as happens with shock waves, and the algorithm may become unstable [Pres 94]. The *Lax–Wendroff method* at-

tains better stability and accuracy by retaining second-order differences for the time derivative:

$$u(x,t+\Delta t) \simeq u(x,t) + \frac{\partial u}{\partial t}\,\Delta t + \frac{1}{2}\,\frac{\partial^2 u}{\partial t^2}\Delta t^2. \tag{19.10}$$

To covert (19.10) to an algorithm, we use Burgers' equation $\partial u/\partial t = -\epsilon\partial(u^2/2)/\partial x$ for the first-order time derivative. Likewise, we use this equation to express the second-order time derivative in terms of space derivatives:

$$\begin{aligned}\frac{\partial^2 u}{\partial t^2} &= \frac{\partial}{\partial t}\left[-\epsilon\frac{\partial}{\partial x}\left(\frac{u^2}{2}\right)\right] = -\epsilon\,\frac{\partial}{\partial x}\,\frac{\partial}{\partial t}\left(\frac{u^2}{2}\right) \\ &= -\epsilon\,\frac{\partial}{\partial x}\left(u\,\frac{\partial u}{\partial t}\right) = \epsilon^2\,\frac{\partial}{\partial x}\left[u\,\frac{\partial}{\partial x}\left(\frac{u^2}{2}\right)\right].\end{aligned} \tag{19.11}$$

We next substitute these derivatives into the Taylor expansion (19.10) to obtain

$$u(x,t+\Delta t) = u(x,t) - \Delta t\epsilon\,\frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) + \frac{(\Delta t)^2}{2}\epsilon^2\,\frac{\partial}{\partial x}\left[u\,\frac{\partial}{\partial x}\left(\frac{u^2}{2}\right)\right].$$

We now replace the outer x derivatives by central differences of spacing $\Delta x/2$:

$$\begin{aligned}u(x,t+\Delta t) = u(x,t) &- \frac{\Delta t\epsilon}{2}\,\frac{u^2(x+\Delta x,t) - u^2(x-\Delta x,t)}{2\Delta x} + \frac{(\Delta t)^2\,\epsilon^2}{2} \\ &\times\frac{1}{2\Delta x}\left[u\left(x+\frac{\Delta x}{2},t\right)\frac{\partial}{\partial x}u^2\left(x+\frac{\Delta x}{2},t\right) - u\left(x-\frac{\Delta x}{2},t\right)\right. \\ &\left.\frac{\partial}{\partial x}u^2\left(x-\frac{\Delta x}{2},t\right)\right].\end{aligned}$$

Next we approximate $u(x\pm\Delta x/2,t)$ by the average of adjacent grid points,

$$u(x\pm\frac{\Delta x}{2},t) \simeq \frac{u(x,t)+u(x\pm\Delta x,t)}{2},$$

and apply a central-difference approximation to the second derivatives:

$$\frac{\partial u^2(x\pm\Delta x/2,t)}{\partial x} = \frac{u^2(x\pm\Delta x,t) - u^2(x,t)}{\pm\Delta x}.$$

Finally, putting all these derivatives together yields the discrete form

$$\begin{aligned}u_{i,j+1} = u_{i,j} &- \frac{\beta}{4}\left(u_{i+1,j}^2 - u_{i-1,j}^2\right) + \frac{\beta^2}{8}\left[\left(u_{i+1,j}+u_{i,j}\right)\left(u_{i+1,j}^2 - u_{i,j}^2\right)\right. \\ &\left.-\left(u_{i,j}+u_{i-1,j}\right)\left(u_{i,j}^2 - u_{i-1,j}^2\right)\right],\end{aligned} \tag{19.12}$$

where we have substituted the CFL number β. This Lax–Wendroff scheme is explicit, centered upon the grid points, and stable for $\beta < 1$ (small nonlinearities).

19.3.2 Implementation and Assessment of Burgers' Shock Equation

Applet

1. Write a program to solve Burgers' equation via the leapfrog method.
2. Define arrays `u0[100]` and `u[100]` for the initial data and the solution.
3. Take the initial wave to be sinusoidal, `u0[i]`$= 3\sin(3.2x)$, with speed $c = 1$.
4. Incorporate the boundary conditions `u[0]=0` and `u[100]=0`.
5. Keep the CFL number $\beta < 1$ for stability.
6. Now modify your program to solve Burgers' shock equation (19.8) using the Lax–Wendroff method (19.12).
7. Save the initial data and the solutions for a number of times in separate files for plotting.
8. Plot the initial wave and the solution for several time values on the same graph in order to see the formation of a shock wave (like Figure 19.1).

9. Run the code for several increasingly large CFL numbers. Is the stability condition $\beta < 1$ correct for this nonlinear problem?
10. Compare the leapfrog and Lax–Wendroff methods. With the leapfrog method you should see shock waves forming but breaking up into ripples as the square edge develops. The ripples are numerical artifacts. The Lax–Wendroff method should give a a better shock wave (square edge), although some ripples may still occur.

Listing 19.1 presents our implementation of the Lax–Wendroff method.

19.4 INCLUDING DISPERSION

We have just seen that Burgers' equation can turn an initially smooth wave into a square-edged shock wave. An inverse wave phenomenon is *dispersion*, in which a waveform disperses or broadens as it travel through a medium. Dispersion does not cause waves to lose energy and attenuate but rather to lose information with time. Physically, dispersion may arise when the propagating medium has structures with a spatial regularity equal to some fraction of a wavelength. Mathematically, dispersion may arise from terms in the wave equation that contain higher-order space derivatives. For example, consider the waveform

$$u(x,t) = e^{\pm i(kx-\omega t)} \tag{19.13}$$

corresponding to a plane wave traveling to the right ("traveling" because the phase $kx - \omega t$ remains unchanged if you increase x with time). When this $u(x,t)$ is substituted into the advection equation (19.4), we obtain

$$\omega = ck. \tag{19.14}$$

This equation is an example of a *dispersion relation*, that is, a relation between frequency ω and wave vector k. Because the *group velocity* of a wave

$$v_g = \frac{\partial \omega}{\partial k}, \tag{19.15}$$

the linear dispersion relation (19.14) leads to all frequencies having the same group velocity c and thus *dispersionless* propagation.

Let us now imagine that a wave is propagating with a small amount of *dispersion*, that is, with a frequency that has somewhat less than a linear increase with the wave number k:

$$\omega \simeq ck - \beta k^3. \tag{19.16}$$

Note that we skip the even powers in (19.16), so that the group velocity,

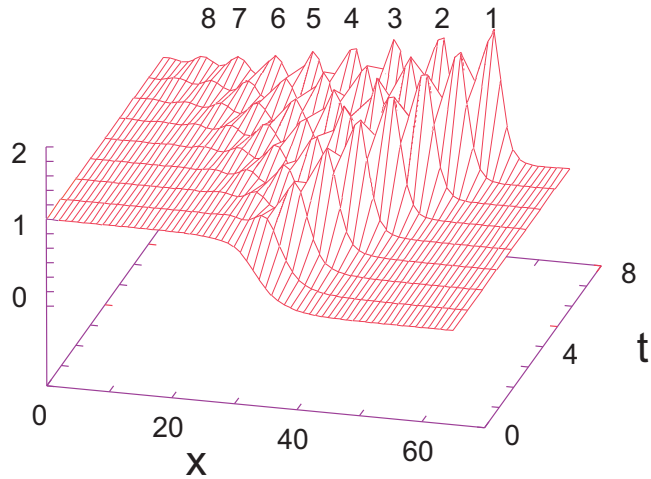
$$v_g = \frac{d\omega}{dk} \simeq c - 3\beta k^2, \tag{19.17}$$

is the same for waves traveling to the left the or the right. If plane-wave solutions like (19.13) were to arise from a wave equation, then (as verified by substitution) the ω term of the dispersion relation (19.16) would arise from a first-order time derivative, the ck term from a first-order space derivative, and the k^3 term from a third-order space derivative:

$$\frac{\partial u(x,t)}{\partial t} + c\,\frac{\partial u(x,t)}{\partial x} + \beta\,\frac{\partial^3 u(x,t)}{\partial x^3} = 0. \tag{19.18}$$

We leave it as an exercise to show that solutions to this equation do indeed have waveforms that disperse in time.

Figure 19.2 The formation of a tsunami. 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. You can generate an animation of this with the program **SolitonAnimate.py**.



19.5 SHALLOW-WATER SOLITONS; THE KDEV EQUATION

In this section we look at shallow-water soliton waves. Though including some complications, this subject is fascinating and is one for which the computer has been absolutely essential for discovery and understanding. In addition, we recommend that you look at some of the soliton animations which we give links to at the beginning of this chapter.

We want to understand the unusual water waves that occur in shallow, narrow channels such as canals [Abar 93, Tab 89]. The analytic description of this "heap of water" was given by Korteweg and deVries (KdeV) [KdeV 95] with the partial differential equation

$$\frac{\partial u(x,t)}{\partial t} + \varepsilon u(x,t)\,\frac{\partial u(x,t)}{\partial x} + \mu\,\frac{\partial^3 u(x,t)}{\partial x^3} = 0. \tag{19.19}$$

As we discussed in §19.1 in our study of Burgers' equation, the nonlinear term $\varepsilon u\,\partial u/\partial t$ leads to a sharpening of the wave and ultimately a *shock* wave. In contrast, as we discussed in our study of dispersion, the $\partial^3 u/\partial x^3$ term produces broadening. These together with the $\partial u/\partial t$ term produce traveling waves. For the proper parameters and initial conditions, the dispersive broadening exactly balances the nonlinear narrowing, and a stable traveling wave is formed.

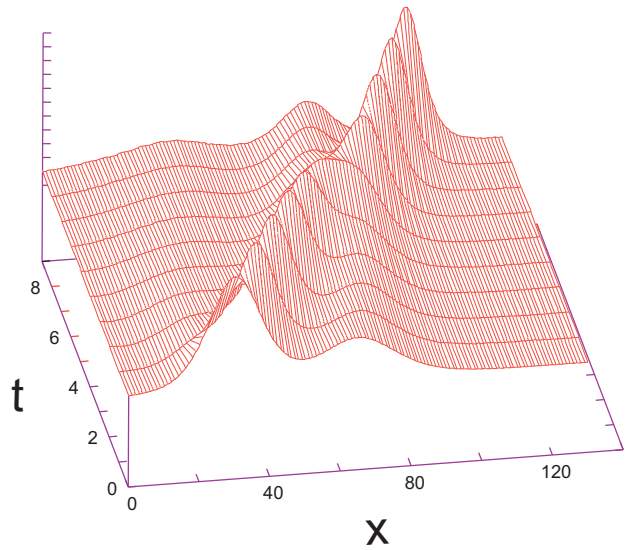
The KdeV equation solved (19.19) analytically and proved that the speed (19.1) given by Russell is in fact correct. Seventy years after its discovery, the KdeV equation was rediscovered by Zabusky and Kruskal [Z&K 65], who solved it numerically and found that a $\cos(x/L)$ initial condition broke up into eight solitary waves (Figure 19.2). They also found that the parts of the wave with larger amplitudes moved faster than those with smaller amplitudes, which is why the higher peaks tend to be on the right in Figure 19.2. As if wonders never cease, Zabusky and Kruskal, who coined the name *soliton* for the solitary wave, also observed that a faster peak actually passed through a slower one unscathed (Figure 19.3).

19.5.1 Analytic Soliton Solution

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,

$$u(x,t) = u(\xi = x - ct). \tag{19.20}$$

Figure 19.3 Two shallow-water solitary waves crossing each other computed with the code `Soliton.py`. The taller soliton on the left catches up with and overtakes the shorter one at $t \simeq 5$. The waves resume their original shapes after the collision.



This form means that if we move with a constant speed c, we will see a constant wave form (but 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 KdeV equation produces a solvable ODE and its solution:

$$-c\,\frac{\partial u}{\partial \xi} + \epsilon\, u\,\frac{\partial u}{\partial \xi} + \mu\,\frac{d^3u}{d\xi^3} = 0, \tag{19.21}$$

$$u(x,t) = \frac{-c}{2}\text{sech}^2\left[\frac{1}{2}\sqrt{c}(x - ct - \xi_0)\right], \tag{19.22}$$

where ξ_0 is the initial phase. We see in (19.22) an amplitude that is proportional to the wave speed c, and a sech^2 function that gives a single lumplike wave. This is a typical analytic form for a soliton.

19.5.2 Algorithm for KdeV Solitons

The KdeV equation is solved numerically using a finite-difference scheme with the time and space derivatives given by central-difference approximations:

$$\frac{\partial u}{\partial t} \simeq \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta t}, \qquad \frac{\partial u}{\partial x} \simeq \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}. \tag{19.23}$$

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 $u(x \pm \Delta x, t)$,

$$u(x \pm \Delta x, t) \simeq u(x,t) \pm (\Delta x)\frac{\partial u}{\partial x} + \frac{(\Delta x)^2}{2!}\,\frac{\partial^2 u}{\partial^2 x} \pm \frac{(\Delta x)^3}{3!}\,\frac{\partial^3 u}{\partial x^3}, \tag{19.24}$$

which we solve for $\partial^3 u(x,t)/\partial x^3$. Finally, the factor $u(x,t)$ in the second term of (19.19) is taken as the average of three x values all with the same t:

$$u(x,t) \simeq \frac{u_{i+1,j} + u_{i,j} + u_{i-1,j}}{3}. \tag{19.25}$$

These substitutions yield the algorithm for the KdeV equation:

$$u_{i,j+1} \simeq u_{i,j-1} - \frac{\epsilon}{3}\frac{\Delta t}{\Delta x}\left[u_{i+1,j} + u_{i,j} + u_{i-1,j}\right]\left[u_{i+1,j} - u_{i-1,j}\right]$$
$$-\mu\frac{\Delta t}{(\Delta x)^3}\left[u_{i+2,j} + 2u_{i-1,j} - 2u_{i+1,j} - u_{i-2,j}\right]. \tag{19.26}$$

To apply this algorithm to predict future times, we need to know $u(x,t)$ at present and past times. The initial-time solution $u_{i,1}$ 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:

$$u_{i,2} \simeq u_{i,1} - \frac{\epsilon\,\Delta t}{6\,\Delta x}\left[u_{i+1,1} + u_{i,1} + u_{i-1,1}\right]\left[u_{i+1,1} - u_{i-1,1}\right]$$
$$-\frac{\mu}{2}\frac{\Delta t}{(\Delta x)^3}\left[u_{i+2,1} + 2u_{i-1,1} - 2u_{i+1,1} - u_{i-2,1}\right]. \tag{19.27}$$

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 (19.27) so that

$$u_{i+2,2} + 2u_{i-1,2} - 2u_{i+1,2} - u_{i-2,2} \to u_{i-1,2} - u_{i+1,2}.$$

The truncation error and stability condition for our algorithm are related:

$$\mathcal{E}(u) = \mathcal{O}[(\Delta t)^3] + \mathcal{O}[\Delta t(\Delta x)^2], \tag{19.28}$$

$$\frac{1}{(\Delta x/\Delta t)}\left[\epsilon|u| + 4\frac{\mu}{(\Delta x)^2}\right] \le 1\,. \tag{19.29}$$

The first equation shows that smaller time and space steps lead to a smaller approximation error, yet because round-off error increases with the number of steps, the total error does not necessarily decrease (Chapter 2, "Errors & Uncertainties in Computations"). Yet we are also limited in how small the steps can be made by the stability condition (19.29), which indicates that making Δx too small always leads to instability. Care and experimentation are required.

19.5.3 Implementation: KdeV Solitons

Applet

Modify or run the program **`Soliton.py`** in Listing 19.2 that solves the KdeV equation (19.19) 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 (19.28) with $|u| = 1$.

1. Define a 2-D 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]`**, $i = 3, 4, \ldots, 129$, corresponding to the next time interval. Use (19.27) to advance the time but note that you cannot start at $i = 1$ or end at $i = 131$

because (19.27) 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 (19.27) so that (19.28) is satisfied.
5. Increment time and compute `u[i][j]` for `j` $=$ `3` and for `i` $=$ `3, 4,..., 129`, using equation (19.26). 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 about 2000 times. Write your solution to a file after approximately every 250 iterations.
8. Use your favorite graphics tool (we used Gnuplot) to plot your results as a 3-D 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.

Listing 19.2 **Soliton.py** solves the KdeV equation for 1-D solitons corresponding to a "bore" initial conditions.

```
# Soliton.py: Solves Korteweg de Vries equation for a soliton.

import matplotlib.pylab as p;
from mpl_toolkits.mplot3d import Axes3D ;
from visual import *;

ds = 0.4;     dt = 0.1;     max = 2000;  mu = 0.1;  eps = 0.2;     mx = 131
u   = zeros( (mx, 3), float); spl = zeros( (mx, 21), float); m = 1

for  i in range(0, 131):                                    # initial wave
    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.     # End points

for i in range (0, 131, 2): spl[i, 0] = u[i, 0]    # initial wave 2x step
fac = mu*dt/(ds**3)
print("Working. Please hold breath and wait while I count to 20")

for  i in range (1, mx-1):                                  # First time step
    a1 = eps*dt*(u[i + 1, 0] + u[i, 0] + u[i - 1, 0])/(ds*6.)
    if i > 1 and  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.

for j in range (1, max+1):                                  # next time steps
    for i in range(1, mx-2):
        a1 = eps*dt*(u[i + 1, 1]  +  u[i, 1]  +  u[i - 1, 1])/(3.*ds)
        if i > 1 and i < mx-2:
          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.
    if j%100 ==  0:                                 # plot every 100 time steps
        for i in range (1, mx - 2): spl[i, m] = u[i, 2]
        print(m)
        m = m + 1
    for k in range(0, mx):                     # recycle array to save memory
        u[k, 0] = u[k, 1]
        u[k, 1] = u[k, 2]

x = list(range(0, mx, 2))                           # plot every other point
y = list(range(0, 21))                     # plot 21 lines every 100 t steps
X, Y = p.meshgrid(x, y)
fig  = p.figure()                                          # create figure
ax = Axes3D(fig)                                               # plot axes
ax.plot_wireframe(X, Y, spl[X, Y], color = 'r')            # red wireframe
ax.set_xlabel('Positon')                                      # label axes
ax.set_ylabel('Time')
ax.set_zlabel('Disturbance')
p.show()                                   # Show figure, close Python shell
print("That's all folks!")
```

The code **SolitonAnimate.py** produces an animation.

19.5.4 Exploration: Solitons in Phase Space, Crossing

1. Explore what happens when a tall soliton collides with a short one.
   a. Start by placing a tall soliton of height 0.8 at $x = 12$ and a smaller soliton in front of it at $x = 26$:
   $$u(x, t = 0) = 0.8\left[1 - \tanh^2\left(\frac{3x}{12} - 3\right)\right] + 0.3\left[1 - \tanh^2\left(\frac{4.5x}{26} - 4.5\right)\right].$$
   b. Do they reflect from each other? Do they go through each other? Do they interfere? Does the tall soliton still move faster than the short one after the collision (Figure 19.3)?
2. Construct phase–space plots [$\dot{u}(t)$ *versus* $u(t)$] of the KdeV 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 KdeV equation. In other words, the stability in time for solitons is analogous to the infinite period for a pendulum balanced straight upward.

19.6 UNIT II. RIVER HYDRODYNAMICS

Problem: In order to give migrating salmon a place to rest during their arduous upstream journey, the Oregon Department of Environment is thinking of placing objects in several deep, wide, fast-flowing streams. One such object is a long beam of rectangular cross section (Figure 19.4 left), and another is a set of plates (Figure 19.4 right). The objects are far enough below the surface so as not to disturb the surface flow, and far enough from the bottom of the stream so as not to disturb the flow there either. Your **problem** is to determine the spatial dependence of the stream's velocity and, specifically, whether the wake of the object will be large enough to provide a resting place for a meter-long salmon.

19.7 HYDRODYNAMICS, THE NAVIER–STOKES EQUATION (THEORY)

We continue with the assumption made in Unit I that water is *incompressible* and thus that its density ρ is constant. We also simplify the theory by looking only at steady-state situations, that is, ones in which the velocity is not a function of time. However, to understand how water flows around objects, like our beam, it is essential to include the complication of frictional forces (*viscosity*).

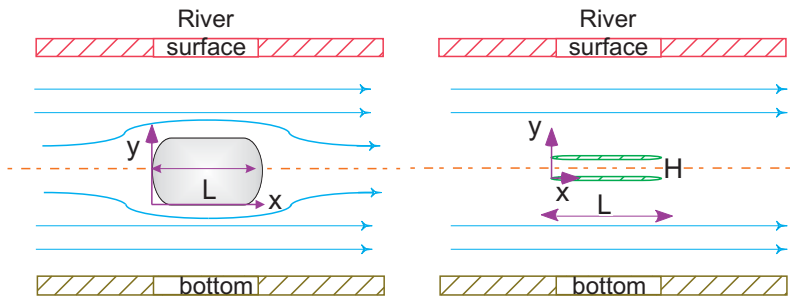
For the sake of completeness, we repeat here the first equation of hydrodynamics, the continuity equation (19.2):

$$\frac{\partial \rho(\mathbf{x}, t)}{\partial t} + \vec{\nabla} \cdot \mathbf{j} = 0, \quad \mathbf{j} \stackrel{\text{def}}{=} \rho \mathbf{v}(\mathbf{x}, t). \tag{19.30}$$

Before proceeding to the second equation, we introduce a special time derivative, the *hydrodynamic* derivative $D\mathbf{v}/Dt$, which is appropriate for a quantity contained in a moving fluid [F&W 80]:

$$\frac{D\mathbf{v}}{Dt} \stackrel{\text{def}}{=} (\mathbf{v} \cdot \vec{\nabla})\mathbf{v} + \frac{\partial \mathbf{v}}{\partial t}. \tag{19.31}$$

Figure 19.4 Cross-sectional view of the flow of a stream around a submerged beam (*left*) and around two parallel plates (*right*). Both beam and plates have length L along the direction of flow. The flow is seen to be symmetric about the centerline and to be unaffected at the bottom and surface by the submerged object.



This derivative gives the rate of change, as viewed from a stationary frame, of the velocity of material in *an element of fluid* and so incorporates changes due to the motion of the fluid (first term) as well as any explicit time dependence of the velocity. Of particular interest is that $D\mathbf{v}/Dt$ is second order in the velocity, and so its occurrence reflects nonlinearities into the theory. You may think of these nonlinearities as related to the fictitious (inertial) forces that would occur if we tried to describe the motion in the fluid's rest frame (an accelerating frame).

The material derivative is the leading term in the *Navier–Stokes equation*,

$$\frac{D\mathbf{v}}{Dt} = \nu\nabla^2\mathbf{v} - \frac{1}{\rho}\vec{\nabla}P(\rho, T, x), \tag{19.32}$$

$$\begin{aligned}
\frac{\partial v_x}{\partial t} + \sum_{j=x}^{z} v_j \frac{\partial v_x}{\partial x_j} &= \nu \sum_{j=x}^{z} \frac{\partial^2 v_x}{\partial x_j^2} - \frac{1}{\rho}\frac{\partial P}{\partial x}, \\
\frac{\partial v_y}{\partial t} + \sum_{j=x}^{z} v_j \frac{\partial v_y}{\partial x_j} &= \nu \sum_{j=x}^{z} \frac{\partial^2 v_y}{\partial x_j^2} - \frac{1}{\rho}\frac{\partial P}{\partial y}, \\
\frac{\partial v_z}{\partial t} + \sum_{j=x}^{z} v_j \frac{\partial v_z}{\partial x_j} &= \nu \sum_{j=x}^{z} \frac{\partial^2 v_z}{\partial x_j^2} - \frac{1}{\rho}\frac{\partial P}{\partial z}.
\end{aligned} \tag{19.33}$$

Here ν is the kinematic viscosity, P is the pressure, and (19.33) shows the derivatives in Cartesian coordinates. This equation describes transfer of the momentum of the fluid within some region of space as a result of forces and flow (think $d\mathbf{p}/dt = \mathbf{F}$), there being a simultaneous equation for each of the three velocity components. The $\mathbf{v}\cdot\nabla\mathbf{v}$ term in $D\mathbf{v}/Dt$ describes transport of momentum in some region of space resulting from the fluid's flow and is often called the *convection* or *advection* term.[2] The $\vec{\nabla}P$ term describes the velocity change as a result of pressure changes, and the $\nu\nabla^2\mathbf{v}$ term describes the velocity change resulting from viscous forces (which tend to dampen the flow).

The explicit functional dependence of the pressure on the fluid's density and temperature $P(\rho, T, x)$ is known as the *equation of state of the fluid* and would have to be known before trying to solve the Navier–Stokes equation. To keep our problem simple we assume that the pressure is independent of density and temperature, which leaves the four simultaneous partial differential equations (19.30) and (19.32) to solve. Because we are interested in *steady-state*

[2] We discuss pure advection in §19.1. In oceanology or meteorology, convection implies the transfer of mass in the vertical direction where it overcomes gravity, whereas advection refers to transfer in the horizontal direction.

flow around an object, we assume that all time derivatives of the velocity vanish. Because we assume that the fluid is incompressible, the time derivative of the density also vanishes, and (19.30) and (19.32) become

$$\vec{\nabla} \cdot \mathbf{v} \equiv \sum_i \frac{\partial v_i}{\partial x_i} = 0, \tag{19.34}$$

$$(\mathbf{v} \cdot \vec{\nabla})\mathbf{v} = \nu \nabla^2 \mathbf{v} - \frac{1}{\rho} \vec{\nabla} P. \tag{19.35}$$

The first equation expresses the equality of inflow and outflow and is known as the *condition of incompressibility*. In as much as the stream in our problem is much wider than the width (z dimension) of the beam and because we are staying away from the banks, we will ignore the z dependence of the velocity. The explicit PDEs we need to solve are then:

$$\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} = 0, \tag{19.36}$$

$$\nu \left(\frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} \right) = v_x \frac{\partial v_x}{\partial x} + v_y \frac{\partial v_x}{\partial y} + \frac{1}{\rho} \frac{\partial P}{\partial x}, \tag{19.37}$$

$$\nu \left(\frac{\partial^2 v_y}{\partial x^2} + \frac{\partial^2 v_y}{\partial y^2} \right) = v_x \frac{\partial v_y}{\partial x} + v_y \frac{\partial v_y}{\partial y} + \frac{1}{\rho} \frac{\partial P}{\partial y}. \tag{19.38}$$

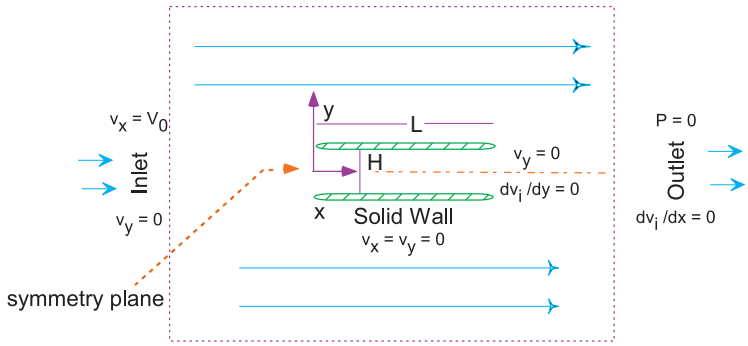
19.7.1 Boundary Conditions for Parallel Plates

The plate problem is relatively easy to solve analytically, and so we will do it! This will give us some experience with the equations as well as a check for our numerical solution. To find a unique solution to the PDEs (19.36)–(19.38), we need to specify boundary conditions. As far as we can tell, picking boundary conditions is somewhat of an acquired skill, and it becomes easier with experience (similar to what happens after solving hundreds of electrostatics problems). Some of the boundary conditions apply to the flow at the surfaces of submerged objects, while others apply to the "box" or tank that surrounds the fluid. As we shall see, sometimes these boundary conditions relate to the velocities directly, while at other times they relate to the derivatives of the velocities.

We assume that the submerged parallel plates are placed in a stream that is flowing with a constant velocity V_0 in the horizontal direction (Figure 19.4 right). If the velocity V_0 is not too high or the kinematic viscosity ν is sufficiently large, then the flow should be smooth and without turbulence. We call such flow *laminar*. Typically, a fluid undergoing laminar flow moves in smooth paths that do not close on themselves, like the smooth flow of water from a faucet. If we imagine attaching a vector to each element of the fluid, then the path swept out by that vector is called a *streamline* or *line of motion* of the fluid. These streamlines can be visualized experimentally by adding a colored dye to the stream. We assume that the plates are so thin that the flow remains laminar as it passes around and through them.

If the plates are thin, then the flow upstream of them is not affected, and we can limit our solution space to the rectangular region in Figure 19.5. We assume that the length L and separation H of the plates are small compared to the size of the stream, so the flow returns to uniform as we get far downstream from the plates. As seen in Figure 19.5, there are boundary conditions at the *inlet* where the fluid enters our solution space, at the *outlet* where it leaves,

Figure 19.5 The boundary conditions for two thin submerged plates. The surrounding box is the integration volume within which we solve the PDEs and upon whose surface we impose the boundary conditions. In practice the box is much larger than L and H.



and at the stationary plates. In addition, since the plates are far from the stream's bottom and surface, we may assume that the dotted-dashed centerline is a plane of symmetry, with identical flow above and below the plane. We thus have four different types of boundary conditions to impose on our solution:

Solid plates: Since there is friction (viscosity) between the fluid and the plate surface, the only way to have laminar flow is to have the fluid's velocity equal to the plate's velocity, which means both are zero:

$$v_x = v_y = 0.$$

Such being the case, we have smooth flow in which the negligibly thin plates lie along streamlines of the fluid (like a "streamlined" vehicle).

Inlet: The fluid enters the integration domain at the inlet with a horizontal velocity V_0. Since the inlet is far upstream from the plates, we assume that the fluid velocity at the inlet is unchanged:

$$v_x = V_0, \quad v_y = 0.$$

Outlet: Fluid leaves the integration domain at the outlet. While it is totally reasonable to assume that the fluid returns to its unperturbed state there, we are not told what that is. So, instead, we assume that there is a physical outlet at the end with the water just shooting out of it. Consequently, we assume that the water pressure equals zero at the outlet (as at the end of a garden hose) and that the velocity does not change in a direction normal to the outlet:

$$P = 0, \quad \frac{\partial v_x}{\partial x} = \frac{\partial v_y}{\partial x} = 0.$$

Symmetry plane: If the flow is symmetric about the $y = 0$ plane, then there cannot be flow through the plane and the spatial derivatives of the velocity components normal to the plane must vanish:

$$v_y = 0, \quad \frac{\partial v_y}{\partial y} = 0.$$

This condition follows from the assumption that the plates are along streamlines and that they are negligibly thin. It means that all the streamlines are parallel to the plates as well as to the water surface, and so it must be that $v_y = 0$ everywhere. The fluid enters in the horizontal direction, the plates do not change the vertical y component of the velocity, and the flow remains symmetric about the centerline. There is a retardation of the flow around the plates due to the viscous nature of the flow and due to the $\mathbf{v} = 0$ boundary layers formed on the plates, but there are no actual v_y components.

19.7.2 Analytic Solution for Parallel Plates

For steady flow around and through the parallel plates, with the boundary conditions imposed and $v_y \equiv 0$, the continuity equation (19.36) reduces to

$$\frac{\partial v_x}{\partial x} = 0. \tag{19.39}$$

This tells us that v_x does not vary with x. With these conditions, the Navier–Stokes equations (19.38) in x and y reduce to the linear PDEs

$$\frac{\partial P}{\partial x} = \rho\nu \frac{\partial^2 v_x}{\partial y^2}, \quad \frac{\partial P}{\partial y} = 0. \tag{19.40}$$

(Observe that if the effect of gravity were also included in the problem, then the pressure would increase with the depth y.) Since the LHS of the first equation describes the x variation, and the RHS the y variation, the only way for the equation to be satisfied in general is if both sides are constant:

$$\frac{\partial P}{\partial x} = C, \quad \rho\nu \frac{\partial^2 v_x}{\partial y^2} = C. \tag{19.41}$$

Double integration of the second equation with respect to y and replacement of the constant by $\partial P/\partial x$ yields

$$\rho\nu \frac{\partial v_x}{\partial y} = \frac{\partial P}{\partial x} y + C_1, \quad \Rightarrow \quad \rho\nu v_x = \frac{\partial P}{\partial x} \frac{y^2}{2} + C_1 y + C_2,$$

where C_1 and C_2 are constants. The values of C_1 and C_2 are determined by requiring the fluid to stop at the plate, $v_x(0) = v_x(H) = 0$, where H is the distance between plates. This yields

$$\rho\nu\, v_x(y) = \frac{1}{2} \frac{\partial P}{\partial x} \left(y^2 - yH\right). \tag{19.42}$$

Because $\partial P/\partial y = 0$, the pressure does not vary with y. The continuity and smoothness of P over the region,

$$\frac{\partial^2 P}{\partial x \partial y} = \frac{\partial^2 P}{\partial y \partial x} = 0, \tag{19.43}$$

are a consequence of laminar flow. Such being the case, we may assume that $\partial P/\partial x$ has no y dependence, and so (19.42) describes a velocity profile varying as y^2.

A check on our numerical CFD simulation ensures that it also gives a parabolic velocity profile for two parallel plates. To be even more precise, we can determine $\partial P/\partial x$ for this problem and thereby produce a purely numerical answer for comparison. To do that we examine a volume of current that at the inlet starts out with $0 \le y \le H$. Because there is no vertical component to the flow, this volume ends up flowing between the plates. If the volume has a unit z width, then the mass flow (mass/unit time) at the inlet is

$$Q(\text{mass/time}) = \rho \times 1 \times H \times \frac{dx}{dt} = \rho H v_x = \rho H V_0. \tag{19.44}$$

When the fluid is between the plates, the velocity has a parabolic variation in height y. Consequently, we integrate over the area of the plates without changing the net mass flow between the plates:

$$Q = \int \rho v_x dA = \rho \int_0^H v_x(y)\, dy = \frac{1}{2\nu} \frac{\partial P}{\partial x} \left(\frac{H^3}{3} - \frac{H^3}{2}\right). \tag{19.45}$$

Yet we know that $Q = \rho H V_0$, and substitution gives us an expression for how the pressure

gradient depends upon the plate separation:

$$\frac{\partial P}{\partial x} = -12\frac{\rho\nu V_0}{H^2}. \tag{19.46}$$

We see that there is a pressure drop as the fluid flows through the plates and that the drop increases as the plates are brought closer together (the Bernoulli effect). To program the equations, we assign the values $V_0 = 1\,\text{m/s}$ ($\simeq$2.24 mi/h), $\rho = 1\,\text{kg/m}^3$ ($\geq$air), $\nu = 1\,\text{m}^2/\text{s}$ (somewhat less viscous than glycerin), and $H = 1\,\text{m}$ (typical boulder size):

$$\frac{\partial P}{\partial x} = -12 \quad \Rightarrow \quad v_x = 6y(1-y). \tag{19.47}$$

19.7.3 Finite-Difference Algorithm and Overrelaxation

Now we develop an algorithm for solution of the Navier–Stokes and continuity PDEs using successive overrelaxation. This is a variation of the method used in Chapter 17, "PDEs for Electrostatics & Heat Flow," to solve Poisson's equation. We divide space into a rectangular grid with the spacing h in both the x and y directions:

$$x = ih, \quad i = 0, \ldots, N_x; \qquad y = jh, \quad j = 0, \ldots, N_y.$$

We next express the derivatives in (19.36)–(19.38) as finite differences of the values of the velocities at the grid points using central-difference approximations. For $\nu = 1\,\text{m}^2/\text{s}$ and $\rho = 1\,\text{kg/m}^3$, this yields

$$\begin{aligned}
&v^x_{i+1,j} - v^x_{i-1,j} + v^y_{i,j+1} - v^y_{i,j-1} = 0,\\
&v^x_{i+1,j} + v^x_{i-1,j} + v^x_{i,j+1} + v^x_{i,j-1} - 4v^x_{i,j}\\
&\quad = \frac{h}{2}v^x_{i,j}\left[v^x_{i+1,j} - v^x_{i-1,j}\right] + \frac{h}{2}v^y_{i,j}\left[v^x_{i,j+1} - v^x_{i,j-1}\right] + \frac{h}{2}\left[P_{i+1,j} - P_{i-1,j}\right],\\
&v^y_{i+1,j} + v^y_{i-1,j} + v^y_{i,j+1} + v^y_{i,j-1} - 4v^y_{i,j}\\
&\quad = \frac{h}{2}v^x_{i,j}\left[v^y_{i+1,j} - v^y_{i-1,j}\right] + \frac{h}{2}v^y_{i,j}\left[v^y_{i,j+1} - v^y_{i,j-1}\right] + \frac{h}{2}\left[P_{i,j+1} - P_{i,j-1}\right].
\end{aligned}$$

Since $v^y \equiv 0$ for this problem, we rearrange terms to obtain for v^x:

$$\begin{aligned}
4v^x_{i,j} = {}& v^x_{i+1,j} + v^x_{i-1,j} + v^x_{i,j+1} + v^x_{i,j-1} - \frac{h}{2}v^x_{i,j}\left[v^x_{i+1,j} - v^x_{i-1,j}\right]\\
&- \frac{h}{2}v^y_{i,j}\left[v^x_{i,j+1} - v^x_{i,j-1}\right] - \frac{h}{2}\left[P_{i+1,j} - P_{i-1,j}\right].
\end{aligned} \tag{19.48}$$

We recognize in (19.48) an algorithm similar to the one we used in solving Laplace's equation by relaxation. Indeed, as we did there, we can accelerate the convergence by writing the algorithm with the new value of v^x given as the old value plus a correction (residual):

$$v^x_{i,j} = v^x_{i,j} + r_{i,j}, \quad r \stackrel{\text{def}}{=} v^{x(\text{new})}_{i,j} - v^{x(\text{old})}_{i,j} \tag{19.49}$$

$$\begin{aligned}
\Rightarrow \quad r = \frac{1}{4}\Bigg\{ & v^x_{i+1,j} + v^x_{i-1,j} + v^x_{i,j+1} + v^x_{i,j-1} - \frac{h}{2}v^x_{i,j}\left[v^x_{i+1,j} - v^x_{i-1,j}\right]\\
&- \frac{h}{2}v^y_{i,j}\left[v^x_{i,j+1} - v^x_{i,j-1}\right] - \frac{h}{2}\left[P_{i+1,j} - P_{i-1,j}\right]\Bigg\} - v^x_{i,j}
\end{aligned} \tag{19.50}$$

As done with the Poisson equation algorithm, successive iterations sweep the interior of the grid, continuously adding in the residual (19.49) until the change becomes smaller than some set level of tolerance, $|r_{i,j}| < \varepsilon$.

A variation of this method, *successive overrelaxation*, increases the speed at which the residuals approach zero via an amplifying factor ω:

$$v^x_{i,j} = v^x_{i,j} + \omega\, r_{i,j} \quad \text{(SOR)}. \tag{19.51}$$

The standard relaxation algorithm (19.49) is obtained with $\omega = 1$, an accelerated convergence (*overrelaxation*) is obtained with $\omega \geq 1$, and *underrelaxation* occurs for $\omega < 1$. Values $\omega > 2$ lead to numerical instabilities and so are not recommended. Although a detailed analysis of the algorithm is necessary to predict the optimal value for ω, we suggest that you test different values for ω to see which one provides the fastest convergence for your problem.

19.7.4 Successive Overrelaxation Implementation

1. Modify the program **`Beam.py`**, or write your own, to solve the Navier–Stokes equation for the velocity of a fluid in 2-D flow. Represent the x and y components of the velocity by the arrays **`vx[Nx][Ny]`** and **`vy[Nx][Ny]`**.
2. Specialize your solution to the rectangular domain and boundary conditions indicated in Figure 19.5.
3. Use of the following parameter values,

$$\nu = 1\,\text{m}^2/\text{s}, \quad \rho = 10^3\,\text{kg/m}^3, \quad \text{(flow parameters)},$$

$$N_x = 400, \quad N_y = 40, \quad h = 1, \quad \text{(grid parameters)},$$

leads to the equations

$$\frac{\partial P}{\partial x} = -12, \quad \frac{\partial P}{\partial y} = 0, \quad v^x = \frac{3j}{20}\left(1 - \frac{j}{40}\right), \quad v^y = 0.$$

4. For the relaxation method, output the iteration number and the computed v^x and then compare the analytic and numeric results.
5. Repeat the calculation and see if SOR speeds up the convergence.

19.8 2-D FLOW OVER A BEAM

Now that the comparison with an analytic solution has shown that our CFD simulation works, we return to determining if the beam in Figure 19.4 might produce a good resting place for salmon. While we have no analytic solution with which to compare, our canoeing and fishing adventures have taught us that *standing waves* with fish in them are often formed behind rocks in streams, and so we expect there will be a standing wave formed behind the beam.

19.9 THEORY: VORTICITY FORM OF NAVIER–STOKES EQUATION

We have seen how to solve numerically the hydrodynamics equations

$$\vec{\nabla} \cdot \mathbf{v} = 0, \quad (\mathbf{v} \cdot \vec{\nabla})\mathbf{v} = -\frac{1}{\rho}\vec{\nabla} P + \nu \nabla^2 \mathbf{v}. \tag{19.52}$$

These equations determine the components of a fluid's velocity, pressure, and density as functions of position. In analogy to electrostatics, where one usually solves for the simpler scalar

potential and then takes its gradient to determine the more complicated vector field, we now recast the hydrodynamic equations into forms that permit us to solve two simpler equations for simpler functions, from which the velocity is obtained via a gradient operation.[3]

We introduce the *stream function* $\mathbf{u}(\mathbf{x})$ from which the velocity is determined by the curl operator:

$$\mathbf{v} \stackrel{\text{def}}{=} \vec{\nabla} \times \mathbf{u}(\mathbf{x}) = \hat{\epsilon}_x \left(\frac{\partial u_z}{\partial y} - \frac{\partial u_y}{\partial z} \right) + \hat{\epsilon}_y \left(\frac{\partial u_x}{\partial z} - \frac{\partial u_z}{\partial x} \right). \tag{19.53}$$

Note the absence of the z component of velocity $\mathbf{v}$ for our problem. Since $\vec{\nabla} \cdot (\vec{\nabla} \times \mathbf{u}) \equiv 0$, we see that any $\mathbf{v}$ that can be written as the curl of $\mathbf{u}$ automatically satisfies the continuity equation $\vec{\nabla} \cdot \mathbf{v} = 0$. Further, since $\mathbf{v}$ for our problem has only x and y components, $\mathbf{u}(\mathbf{x})$ needs to have only a z component:

$$u_z \equiv u \quad \Rightarrow \quad v_x = \frac{\partial u}{\partial y}, \quad v_y = -\frac{\partial u}{\partial x}. \tag{19.54}$$

(Even though the vorticity has just one component, it is a pseudoscalar and not a scalar because it reverses sign upon reflection.) It is worth noting that in 2-D flows, the contour lines $u =$ constant are the *streamlines*.

The second simplifying function is the *vorticity* field $\mathbf{w}(\mathbf{x})$, which is related physically and alphabetically to the angular velocity ω of the fluid. Vorticity is defined as the curl of the velocity (sometimes with a $-$ sign):

$$\mathbf{w} \stackrel{\text{def}}{=} \vec{\nabla} \times \mathbf{v}(\mathbf{x}). \tag{19.55}$$

Because the velocity in our problem does not change in the z direction, we have

$$w_z = \left(\frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y} \right). \tag{19.56}$$

Physically, we see that the vorticity is a measure of how much the fluid's velocity curls or rotates, with the direction of the vorticity determined by the right-hand rule for rotations. In fact, if we could pluck a small element of the fluid into space (so it would not feel the internal strain of the fluid), we would find that it is rotating like a solid with angular velocity $\omega \propto \mathbf{w}$ [Lamb 93]. That being the case, it is useful to think of the vorticity as giving the local value of the fluid's angular velocity vector. If $\mathbf{w} = 0$, we have *irrotational* flow.

The field lines of w are continuous and move as if attached to the particles of the fluid. A uniformly flowing fluid has vanishing curl, while a nonzero vorticity indicates that the current curls back on itself or rotates. From the definition of the stream function (19.53), we see that the vorticity $\mathbf{w}$ is related to it by

$$\mathbf{w} = \vec{\nabla} \times \mathbf{v} = \vec{\nabla} \times (\vec{\nabla} \times \mathbf{u}) = \vec{\nabla}(\vec{\nabla} \cdot \mathbf{u}) - \nabla^2 \mathbf{u}, \tag{19.57}$$

where we have used a vector identity for $\vec{\nabla} \times (\vec{\nabla} \times \mathbf{u})$. Yet the divergence $\vec{\nabla} \cdot \mathbf{u} = 0$ since $\mathbf{u}$ has only a z component that does not vary with z (or because there is no source for $\mathbf{u}$). We have now obtained the basic relation between the stream function $\mathbf{u}$ and the vorticity $\mathbf{w}$:

$$\vec{\nabla}^2 \mathbf{u} = -\mathbf{w}. \tag{19.58}$$

Equation (19.58) is analogous to Poisson's equation of electrostatics, $\nabla^2 \phi = -4\pi\rho$, only now each component of vorticity $\mathbf{w}$ is a source for the corresponding component of the stream function $\mathbf{u}$. If the flow is irrotational, that is, if $\mathbf{w} = 0$, then we need only solve Laplace's

[3] If we had to solve only the simpler problem of *irrotational flow* (no turbulence), then we would be able to use a scalar velocity potential, in close analogy to electrostatics [Lamb 93]. For the more general *rotational flow*, two vector potentials are required.

equation for each component of u. Rotational flow, with its coupled nonlinearities equations, leads to more interesting behavior.

As is to be expected from the definition of $\mathbf{w}$, the vorticity form of the Navier–Stokes equation is obtained by taking the curl of the velocity form, that is, by operating on both sides with $\vec{\nabla}\times$. After significant manipulations we obtain

$$\nu\nabla^2\mathbf{w} = [(\vec{\nabla}\times\mathbf{u})\cdot\vec{\nabla}]\mathbf{w}. \tag{19.59}$$

This and (19.58) are the two simultaneous PDEs that we need to solve. In 2-D, with $\mathbf{u}$ and $\mathbf{w}$ having only z components, they are

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -w, \tag{19.60}$$

$$\nu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2}\right) = \frac{\partial u}{\partial y}\frac{\partial w}{\partial x} - \frac{\partial u}{\partial x}\frac{\partial w}{\partial y}. \tag{19.61}$$

So after all that work, we end up with two simultaneous, nonlinear, elliptic PDEs for the functions $w(x,y)$ and $u(x,y)$ that look like a mixture of Poisson's equation with the frictional and variable-density terms of the wave equation. The equation for u is Poisson's equation with source w and must be solved simultaneously with the second. It is this second equation that contains mixed products of the derivatives of u and w and thus introduces nonlinearity.

19.9.1 Finite Differences and the SOR Algorithm

We solve (19.60) and (19.61) on an $N_x \times N_y$ grid of uniform spacing h with

$$x = i\Delta x = ih, \quad i = 0,\ldots,N_x, \quad y = j\Delta y = jh, \quad j = 0,\ldots,N_y.$$

Since the beam is symmetric about its centerline (Figure 19.4 left), we need the solution only in the upper half-plane. We apply the now familiar central-difference approximation to the Laplacians of u and w to obtain the difference equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \simeq \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2}. \tag{19.62}$$

Likewise, for the first derivatives,

$$\frac{\partial u}{\partial y}\frac{\partial w}{\partial x} \simeq \frac{u_{i,j+1} - u_{i,j-1}}{2h}\,\frac{w_{i+1,j} - w_{i-1,j}}{2h}. \tag{19.63}$$

The difference form of the vorticity Navier–Stokes equation (19.60) becomes

$$u_{i,j} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + h^2 w_{i,j}\right), \tag{19.64}$$

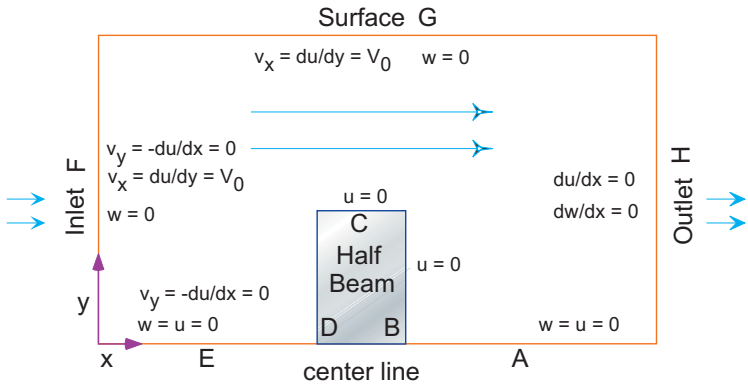
$$w_{i,j} = \frac{1}{4}(w_{i+1,j} + w_{i-1,j} + w_{i,j+1} + w_{i,j-1}) - \frac{R}{16}\{[u_{i,j+1} - u_{i,j-1}] \times [w_{i+1,j} - w_{i-1,j}] - [u_{i+1,j} - u_{i-1,j}]\,[w_{i,j+1} - w_{i,j-1}]\}, \tag{19.65}$$

$$R = \frac{1}{\nu} = \frac{V_0 h}{\nu} \quad \text{(in normal units)}. \tag{19.66}$$

Note that we have placed $u_{i,j}$ and $w_{i,j}$ on the LHS of the equations in order to obtain an algorithm appropriate to solution by relaxation.

The parameter R in (19.66) is related to the *Reynolds number*. When we solve the problem in natural units, we measure distances in units of grid spacing h, velocities in units of initial velocity V_0, stream functions in units of $V_0 h$, and vorticity in units of V_0/h. The second

Figure 19.6 Boundary conditions for flow around the beam in Figure 19.4. The flow is symmetric about the centerline, and the beam has length L in the x direction (along flow).



form is in regular units and is dimensionless. This R is known as the *grid Reynolds number* and differs from the physical R, which has a pipe diameter in place of the grid spacing h.

The grid Reynolds number is a measure of the strength of the coupling of the nonlinear terms in the equation. When the physical R is small, the viscosity acts as a frictional force that damps out fluctuations and keeps the flow smooth. When R is large ($R \simeq 2000$), physical fluids undergo phase transitions from laminar to turbulent flow in which turbulence occurs at a cascading set of smaller and smaller space scales [Rey 83]. However, simulations that produce the onset of turbulence have been a research problem since Reynolds first experiments in 1883 [Rey 83, F&S], possibly because laminar flow is stable against small perturbations and some large-scale "kick" appears necessary to change laminar to turbulent flow. Recent research along these lines have been able to find unstable, traveling-wave solutions to the Navier–Stokes equations, and the hope is that these may lead to a turbulent transition [Fitz 04].

As discussed in §19.7.3, the finite-difference algorithm can have its convergence accelerated by the use of successive overrelaxation (19.64):

$$u_{i,j} = u_{i,j} + \omega\, r^{(1)}_{i,j}, \qquad w_{i,j} = w_{i,j} + \omega\, r^{(2)}_{i,j} \quad \text{(SOR)}. \tag{19.67}$$

Here ω is the overrelaxation parameter and should lie in the range $0 < \omega < 2$ for stability. The residuals are just the changes in a single step, $r^{(1)} = u^{\text{new}} - u^{\text{old}}$ and $r^{(2)} = w^{\text{new}} - w^{\text{old}}$:

$$r^{(1)}_{i,j} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + w_{i,j}\right) - u_{i,j}, \tag{19.68}$$

$$r^{(2)}_{i,j} = \frac{1}{4}\Bigg(w_{i+1,j} + w_{i-1,j} + w_{i,j+1} + w_{i,j-1} - \frac{R}{4}\left\{[u_{i,j+1} - u_{i,j-1}]\right.$$

$$\left.\times\,[w_{i+1,j} - w_{i-1,j}] - [u_{i+1,j} - u_{i-1,j}]\,[w_{i,j+1} - w_{i,j-1}]\right\}\Bigg) - w_{i,j}.$$

19.9.2 Boundary Conditions for a Beam

A well-defined solution of these elliptic PDEs requires a combination of (less than obvious) boundary conditions on u and w. Consider Figure 19.6, based on the analysis of [Koon 86]. We assume that the inlet, outlet, and surface are far from the beam, which may not be evident from the not-to-scale figure.

Freeflow: If there were no beam in the stream, then we would have free flow with the entire fluid possessing the inlet velocity:

$$v_x \equiv V_0, \quad v_y = 0, \quad \Rightarrow \quad u = V_0 y, \quad w = 0. \tag{19.69}$$

(Recollect that we can think of $w = 0$ as indicating no fluid rotation.) The centerline divides the system along a symmetry plane with identical flow above and below it. If the velocity is symmetric about the centerline, then its y component must vanish there:

$$v_y = 0, \quad \Rightarrow \quad \frac{\partial u}{\partial x} = 0 \quad \text{(centerline AE)}. \tag{19.70}$$

Centerline: The centerline is a streamline with $u =$ constant because there is no velocity component perpendicular to it. We set $u = 0$ according to (19.69). Because there cannot be any fluid flowing into or out of the beam, the normal component of velocity must vanish along the beam surfaces. Consequently, the streamline $u = 0$ is the entire lower part of Figure 19.6, that is, the centerline and the beam surfaces. Likewise, the symmetry of the problem permits us to set the vorticity $w = 0$ along the centerline.

Inlet: At the inlet, the fluid flow is horizontal with uniform x component V_0 at all heights and with no rotation:

$$v_y = -\frac{\partial u}{\partial x} = 0, \quad w = 0 \quad \text{(inlet F)}, \quad v_x = \frac{\partial u}{\partial y} = V_0. \tag{19.71}$$

Surface: We are told that the beam is sufficiently submerged so as not to disturb the flow on the surface of the stream. Accordingly, we have free-flow conditions on the surface:

$$v_x = \frac{\partial u}{\partial y} = V_0, \quad w = 0 \quad \text{(surface G)}. \tag{19.72}$$

Outlet: Unless something truly drastic is happening, the conditions on the far downstream outlet have little effect on the far upstream flow. A convenient choice is to require the stream function and vorticity to be constant:

$$\frac{\partial u}{\partial x} = \frac{\partial w}{\partial x} = 0 \quad \text{(outlet H)}. \tag{19.73}$$

Beamsides: We have already noted that the normal component of velocity v_x and stream function u vanish along the beam surfaces. In addition, because the flow is viscous, it is also true that the fluid "sticks" to the beam somewhat and so the tangential velocity also vanishes along the beam's surfaces. While these may all be true conclusions regarding the flow, specifying them as boundary conditions would overrestrict the solution (see Table 17.1 for elliptic equations) to the point where no solution may exist. Accordingly, we simply impose the *no-slip* boundary condition on the vorticity w. Consider a grid point (x, y) on the upper surface of the beam. The stream function u at a point $(x, y + h)$ above it can be related via a Taylor series in y:

$$u(x, y + h) = u(x, y) + \frac{\partial u}{\partial y}(x, y)h + \frac{\partial^2 u}{\partial y^2}(x, y)\frac{h^2}{2} + \cdots . \tag{19.74}$$

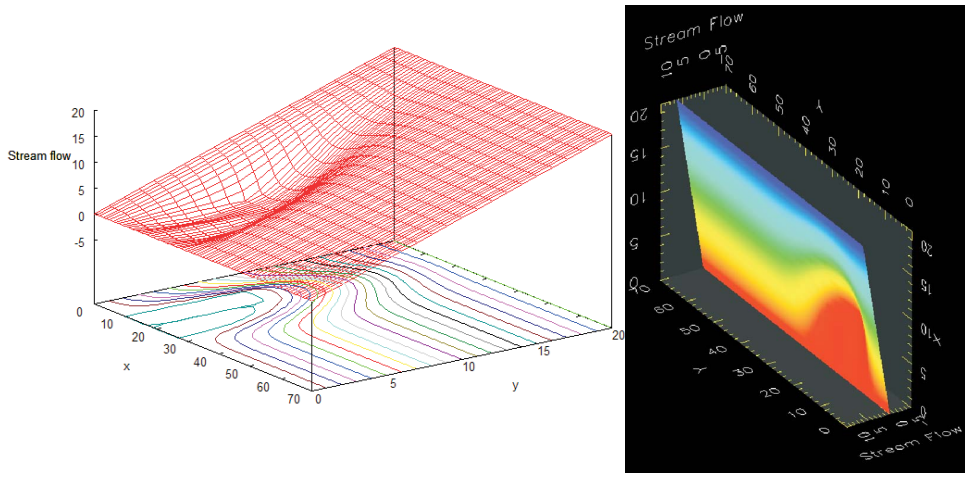
Because $\mathbf{w}$ has only a z component, it has a simple relation to $\nabla \times \mathbf{v}$:

$$w \equiv w_z = \frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y}. \tag{19.75}$$

Because of the fluid's viscosity, the velocity is stationary along the beam top:

$$v_x = \frac{\partial u}{\partial y} = 0 \quad \text{(beam top)}. \tag{19.76}$$

Figure 19.7 The stream function u for Reynold's number $R = 5$. *Left:* Gnuplot surface plot with contours, and *right:* visualized via colors by OpenDX.



Because the current flows smoothly along the top of the beam, v_y must also vanish. In addition, since there is no x variation, we have

$$\frac{\partial v_y}{\partial x} = 0 \;\Rightarrow\; w = -\frac{\partial v_x}{\partial y} = -\frac{\partial^2 u}{\partial y^2}. \tag{19.77}$$

After substituting these relations into the Taylor series (19.74), we can solve for w and obtain the finite-difference version of the top boundary condition:

$$w \simeq -2\frac{u(x, y+h) - u(x, y)}{h^2} \;\Rightarrow\; w_{i,j} = -2\frac{u_{i,j+1} - u_{i,j}}{h^2} \quad \text{(top)}. \tag{19.78}$$

Similar treatments applied to other surfaces yield the following boundary conditions.

$u = 0;$	$w = 0$	Centerline EA
$u = 0,$	$w_{i,j} = -2(u_{i+1,j} - u_{i,j})/h^2$	Beam back B
$u = 0,$	$w_{i,j} = -2(u_{i,j+1} - u_{i,j})/h^2$	Beam top C
$u = 0,$	$w_{i,j} = -2(u_{i-1,j} - u_{i,j})/h^2$	Beam front D
$\partial u/\partial x = 0,$	$w = 0$	Inlet F
$\partial u/\partial y = V_0,$	$w = 0$	Surface G
$\partial u/\partial x = 0,$	$\partial w/\partial x = 0$	Outlet H

19.9.3 SOR on a Grid Implementation

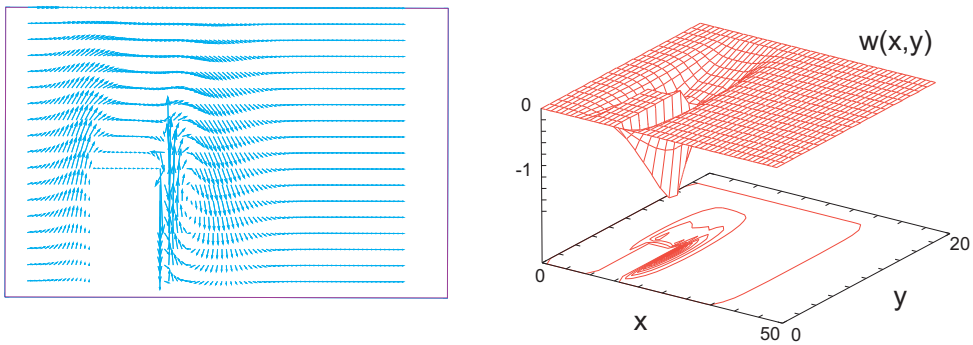
`Beam.py` in Listing 19.3 is our solution of the vorticity form of the Navier–Stokes equation. You will notice that while the relaxation algorithm is rather simple, some care is needed in implementing the many boundary conditions. Relaxation of the stream function and of the vorticity is done by separate methods, and the file output format is that for a Gnuplot surface plot.

Listing 19.3 **Beam.py** solves the Navier–Stokes equation for the flow over a plate.

```
# Beam.py: solves Navier - Stokes equation for flow around beam

import matplotlib.pylab as p;
from mpl_toolkits.mplot3d import Axes3D ;
```

```
from numpy import *;

print( "Working, look for figure window after 100 iterations")

Nxmax = 70;    Nymax = 20;    IL = 10;    H = 8;    T = 8;    h = 1.
u = zeros( (Nxmax + 1, Nymax + 1), float)                    # Stream
w = zeros( (Nxmax + 1, Nymax + 1), float)                 # Vorticity
V0 = 1.0;    omega = 0.1;    nu = 1.;    iter = 0;  R = V0*h/nu  # Renold #

def borders():                                  # Method borders: init & B.C
    for i in range(0, Nxmax + 1):               # Initialize stream function
       for j in range(0, Nymax + 1 ):                      # And vorticity
            w[i, j] = 0.
            u[i, j] = j * V0
    for i in range(0, Nxmax + 1 ):                         # Fluid surface
      u[i, Nymax] = u[i, Nymax - 1]  +  V0 * h
      w[i, Nymax - 1] = 0.
    for j in range(0, Nymax + 1 ):
        u[1, j] = u[0, j]
        w[0, j] = 0.                                                # Inlet
    for  i in range(0, Nxmax + 1 ):                            # Centerline
       if i <=  IL and i>= IL  +  T:
           u[i, 0] = 0.
           w[i, 0] = 0.
    for j in range(1, Nymax ):                                     # Outlet
        w[Nxmax, j] = w[Nxmax - 1, j]
        u[Nxmax, j] = u[Nxmax - 1, j]                 # Boundary conditions

def beam():                                       # Method beam; BC for beam
   for  j in range (0, H + 1):                                # Beam sides
      w[IL, j] =  - 2 * u[IL - 1, j]/(h*h)                    # Front side
      w[IL  +  T, j] =  - 2 * u[IL  +  T  +  1, j]/(h*h)      # Back side
   for  i in range(IL, IL  +  T + 1): w[i, H - 1] =  - 2 * u[i, H]/(h*h);
   for i in range(IL, IL  +  T + 1 ):
     for j in range(0, H + 1):
         u[IL, j] = 0.                                              # Front
         u[IL+T, j] = 0.                                             # Back
         u[i, H] = 0;                                                 # top

def relax():                                     # Method to relax stream
    beam()                                      # Reset conditions at beam
    for  i in range(1, Nxmax):                     # Relax stream function
      for  j in range (1, Nymax):
        r1 = omega*((u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1] + h*h*w[i,j])*0.25-u[i,j])
        u[i, j] +=  r1
    for  i in range(1, Nxmax):                          # Relax vorticity
      for j in range(1, Nymax):
          a1 = w[i+1, j]  +  w[i-1,j]  +  w[i,j+1] + w[i,j-1]
          a2 = (u[i,j+1] - u[i,j-1])*(w[i+1,j] - w[i - 1, j])
          a3 = (u[i+1,j] - u[i-1,j])*(w[i,j+1] - w[i, j - 1])
          r2 = omega *( (a1 - (R/4.)*(a2 - a3) )/4.0  - w[i,j])
          w[i, j]    +=  r2

borders()
while (iter <=  100):
    iter   +=  1
    if iter%10 ==  0: print (iter)
    relax()
for i in range (0, Nxmax + 1):
    for  j in range(0, Nymax + 1 ):  u[i, j] = u[i, j]/(V0*h) # V0h units
x = range(0, Nxmax - 1);        y = range(0, Nymax - 1)
X, Y = p.meshgrid(x, y)

def functz(u):                                # returns stream flow to plot
    z = u[X, Y]                                     # for several iterations
    return z

Z = functz(u)                                 # here the function is called
fig = p.figure()                                        # creates the figure
ax = Axes3D(fig)                            # plots the axis for the figure
ax.plot_wireframe(X, Y, Z, color = 'r')       # surface of wireframe in red
ax.set_xlabel('X')                                     # label the three axes
ax.set_ylabel('Y')
ax.set_zlabel('Stream Function')
p.show()                                  # show figure & close Python shell
```

Figure 19.8 *Left:* The velocity field around the beam as represented by vectors. *Right:* The vorticity as a function of x and y. Rotation is seen to be largest behind the beam.



19.9.4 Assessment

1. Use **`Beam.py`** as a basis for your solution for the stream function u and the vorticity w using the finite-differences algorithm (19.64).
2. A good place to start your simulation is with a beam of size $L = 8h$, $H = h$, Reynolds number $R = 0.1$, and intake velocity $V_0 = 1$. Keep your grid small during debugging, say, $N_x = 24$ and $N_y = 70$.
3. Explore the **convergence** of the algorithm.
   a. Print out the iteration number and u values upstream from, above, and downstream from the beam.
   b. Determine the number of iterations necessary to obtain three-place convergence for successive relaxation ($\omega = 0$).
   c. Determine the number of iterations necessary to obtain three-place convergence for successive overrelaxation ($\omega \simeq 0.3$). Use this number for future calculations.
4. Change the beam's horizontal placement so that you can see the undisturbed current entering on the left and then developing into a standing wave. Note that you may need to increase the size of your simulation volume to see the effect of all the boundary conditions.
5. Make surface plots including contours of the stream function u and the vorticity w. Explain the behavior seen.
6. Is there a region where a big fish can rest behind the beam?
7. The results of the simulation (Figure 19.7) are for the one-component stream function u. Make several visualizations showing the fluid velocity throughout the simulation region. Note that the velocity is a vector with two components (or a magnitude and direction), and both degrees of freedom are interesting to visualize. A plot of vectors would work well here (Gnuplot and OpenDX make *vector* plots for this purpose, Mathematica has *plotfield*, and Maple has *fieldplot*, although the latter two require some work for numerical data).
8. Explore how increasing the Reynolds number R changes the flow pattern. Start at $R = 0$ and gradually increase R while watching for numeric instabilities. To overcome the instabilities, reduce the size of the relaxation parameter ω and continue to larger R values.
9. Verify that the flow around the beam is smooth for small R values, but that it separates from the back edge for large R, at which point a small vortex develops.

19.9.5 Exploration

1. Determine the flow behind a circular rock in the stream.
2. The boundary condition at an outlet far downstream should not have much effect on the simulation. Explore the use of other boundary conditions there.
3. Determine the pressure variation around the beam.

Chapter Twenty

Integral Equations in Quantum Mechanics

The power and accessibility of high-speed computers have changed the view as to what kind of equations are soluble. In Chapter 9, "Differential Equation Applications," and Chapter 12, "Discrete & Continuous Nonlinear Dynamics," we saw how even nonlinear differential equations can be solved easily and can give new insight into the physical world. In this chapter we examine how the integral equations of quantum mechanics can be solved for both bound and scattering states. In Unit I we extend our treatment of the eigenvalue problem, solved as a coordinate-space differential equation in Chapter 9, to the equivalent integral-equation problem in momentum space. In Unit II we treat the singular integral equations for scattering, a more difficult problem. After studying this chapter, we hope that the reader will view both integral and differential equations as soluble.

VIDEO LECTURES, APPLETS AND ANIMATIONS

This Chapter's Lecture & Slide Web Links (All Lectures)

Lecture (Flash)	*Slides*	*Sections*	*Lecture (Flash)*	*Slides*	*Sections*
Integral Equations, QM Bound	pdf	20.1	Integral Equations, QM Scatter	pdf	20.3

20.1 UNIT I. BOUND STATES OF NONLOCAL POTENTIALS

Problem: A particle undergoes a many-body interaction with a medium (Figure 20.1) that results in the particle experiencing an effective potential at **r** that depends on the wave function at the **r′** values of the other particles [L 96]:

$$V(r)\psi(r) \to \int dr'\, V(r,r')\psi(r'). \tag{20.1}$$

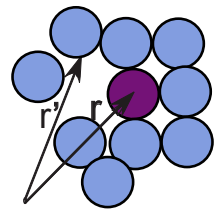
This type of interaction is called *nonlocal* and leads to a Schrödinger equation that is a combined integral and differential ("integrodifferential") equation:

$$-\frac{1}{2\mu}\frac{d^2\psi(r)}{dr^2} + \int dr'\, V(r,r')\psi(r') = E\psi(r). \tag{20.2}$$

Your **problem** is to figure out how to find the bound-state energies E and wave functions ψ for the integral equation in (20.2).[1]

[1] We use natural units in which $\hbar \equiv 1$ and omit the traditional bound-state subscript n on E and ψ in order to keep the notation simpler.

Figure 20.1 A dark particle moving in a dense multiparticle medium. The nonlocality of the potential felt by the dark particle at $\mathbf{r}$ arises from the particle interactions at all $\mathbf{r}'$.



20.2 MOMENTUM-SPACE SCHRÖDINGER EQUATION (THEORY)

One way of dealing with equation (20.2) is by going to momentum space where it becomes the integral equation [L 96]

$$\frac{k^2}{2\mu}\psi(k) + \frac{2}{\pi}\int_0^\infty dp p^2 V(k,p)\psi(p) = E\psi(k), \tag{20.3}$$

where we restrict our solution to $l = 0$ partial waves. In (20.3), $V(k,p)$ is the momentum-space representation (double Fourier transform) of the potential,

$$V(k,p) = \frac{1}{kp}\int_0^\infty dr \sin(kr)V(r)\sin(pr), \tag{20.4}$$

and $\psi(k)$ is the (unnormalized) momentum-space wave function (the probability amplitude for finding the particle with momentum k),

$$\psi(k) = \int_0^\infty drkr\psi(r)\sin(kr). \tag{20.5}$$

Equation (20.3) is an integral equation for $\psi(k)$, in contrast to an integral representation of $\psi(k)$, because the integral in it cannot be evaluated until $\psi(p)$ is known. Although this may seem like an insurmountable barrier, we will transform this equation into a matrix equation that can be solved with the matrix techniques discussed in Chapter 8, "Solving Systems of Equations with Matrices; Data Fitting."

20.2.1 Integral to Linear Equations (Method)

We approximate the integral over the potential as a weighted sum over N integration points (usually Gauss quadrature[2]) for $p = k_j,\ j = 1, N$:

$$\int_0^\infty dp p^2 V(k,p)\psi(p) \simeq \sum_{j=1}^N w_j k_j^2 V(k,k_j)\psi(k_j). \tag{20.6}$$

This converts the integral equation (20.3) to the algebraic equation

$$\frac{k^2}{2\mu}\psi(k) + \frac{2}{\pi}\sum_{j=1}^N w_j k_j^2 V(k,k_j)\psi(k_j) = E. \tag{20.7}$$

Equation (20.7) contains the N unknowns $\psi(k_j)$, the single unknown E, and the unknown function $\psi(k)$. We eliminate the need to know the entire function $\psi(k)$ by restricting the solution to the same values of k_i as used to approximate the integral. This leads to a set of N

[2] See Chapter 6, "Integration," for a discussion of numerical integration.

Figure 20.2 The grid of momentum values on which the integral equation is solved.

coupled linear equations in $(N+1)$ unknowns:

$$\frac{k_i^2}{2\mu}\psi(k_i) + \frac{2}{\pi}\sum_{j=1}^{N} w_j k_j^2\, V(k_i,k_j)\psi(k_j) = E\psi(k_i), \quad i = 1, N. \tag{20.8}$$

As a case in point, if $N = 2$, we would have the two simultaneous linear equations

$$\frac{k_1^2}{2\mu}\psi(k_1) + \frac{2}{\pi}w_1 k_1^2\, V(k_1,k_1)\psi(k_1) + w_2 k_2^2\, V(k_1,k_2) = E\psi(k_1),$$

$$\frac{k_2^2}{2\mu}\psi(k_2) + \frac{2}{\pi}w_1 k_1^2\, V(k_2,k_1)\psi(k_1) + w_2 k_2^2\, V(k_2,k_2)\psi(k_2) = E\psi(k_2).$$

We write our coupled dynamic equations in matrix form as

$$[H][\psi] = E[\psi] \tag{20.9}$$

or as explicit matrices

$$\begin{pmatrix} \frac{k_1^2}{2\mu} + \frac{2}{\pi}V(k_1,k_1)k_1^2 w_1 & \frac{2}{\pi}V(k_1,k_2)k_2^2 w_2 & \cdots & \frac{2}{\pi}V(k_1,k_N)k_N^2 w_N \\ \frac{2}{\pi}V(k_2,k_1)k_1^2 w_1 & \frac{2}{\pi}V(k_2,k_2)k_2^2 w_2 + \frac{k_2^2}{2\mu} & \cdots & \\ \ddots & & & \\ \cdots & \cdots & \cdots & \frac{k_N^2}{2\mu} + \frac{2}{\pi}V(k_N,k_N)k_N^2 w_N \end{pmatrix}$$

$$\times \begin{pmatrix} \psi(k_1) \\ \psi(k_2) \\ \ddots \\ \psi(k_N) \end{pmatrix} = E \begin{pmatrix} \psi(k_1) \\ \psi(k_2) \\ \ddots \\ \psi(k_N) \end{pmatrix}. \tag{20.10}$$

Equation (20.9) is the matrix representation of the Schrödinger equation (20.3). The wave function $\psi(k)$ on the grid is the $N \times 1$ vector

$$[\psi(k_i)] = \begin{pmatrix} \psi(k_1) \\ \psi(k_2) \\ \ddots \\ \psi(k_N) \end{pmatrix}. \tag{20.11}$$

The astute 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 (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). We rewrite (20.9) as

$$[H - EI][\psi] = [0] \tag{20.12}$$

and multiply both sides by the inverse of $[H - EI]$ to obtain the formal solution

$$[\psi] = [H - EI]^{-1}[0]. \tag{20.13}$$

This equation tells us that (1) if the inverse exists, then we have the *trivial* solution $\psi \equiv 0$, which is not very interesting, and (2) for a nontrivial solution to exist, 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:

$$\det[H - EI] = 0 \qquad \text{(bound-state condition)}. \tag{20.14}$$

Equation (20.14) is the additional equation needed to find unique solutions to the eigenvalue problem. Nevertheless, there is no guarantee that solutions of (20.14) can always be found, but if they are found, they are the desired *eigenvalues* of (20.9).

20.2.2 Delta-Shell Potential (Model)

To keep things simple and to have an analytic answer to compare with, we consider the local delta-shell potential:

$$V(r) = \frac{\lambda}{2\mu}\delta(r - b). \tag{20.15}$$

This would be a good model for an interaction that occurs in 3-D when two particles are predominantly a fixed distance b apart. We use (20.4) to determine its momentum-space representation:

$$V(k', k) = \int_0^\infty \frac{\sin(k'r')}{k'k}\frac{\lambda}{2\mu}\delta(r - b)\sin(kr)\,dr = \frac{\lambda}{2\mu}\frac{\sin(k'b)\sin(kb)}{k'k}. \tag{20.16}$$

Beware: We have chosen this potential because it is easy to evaluate the momentum-space matrix element of the potential. However, its singular nature in r space leads to a very slow falloff in k space, and this causes the integrals to converge so slowly that numerics are not as precise as we would like.

If the energy is parameterized in terms of a wave vector κ by $E = -\kappa^2/2\mu$, then for this potential there is, at most, one bound state and it satisfies the transcendental equation [Gott 66]

$$e^{-2\kappa b} - 1 = \frac{2\kappa}{\lambda}. \tag{20.17}$$

Note that bound states occur only for attractive potentials and 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 λ and use them to verify with a hand calculation that (20.17) can be solved for κ. ▌

20.2.3 Binding Energies Implementation

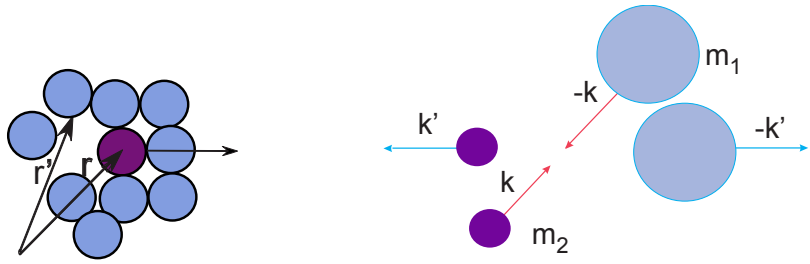
An actual computation may follow two paths. The first calls subroutines to evaluate the determinant of the $[H - EI]$ matrix in (20.14) and then to *search* for those values of energy for which the computed determinant vanishes. This provides E, but not wave functions. The other approach calls an eigenproblem 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. In Listing 20.1 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.

Listing 20.1 **Bound.py** 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 LINALG.

```
# Bound.py: Bound state solutn of Lippmann-Schwinger equation in p space

from numpy import*
from numpy.linalg import*

min1 =0.;     max1 =200.;  u =0.5;    b =10.

def gauss(npts,a,b,x,w):
    pp = 0.;  m = (npts + 1)//2;  eps = 3.E-10        # Accuracy: ADJUST!

    for i in range(1,m+1):
        t = cos(math.pi*(float(i)-0.25)/(float(npts) + 0.5))
        t1 = 1
        while((abs(t-t1)) >= eps):
            p1 = 1. ;  p2 = 0.;
            for j in range(1,npts+1):
                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 i in range(0,npts):
        x[i] = x[i]*(b-a)/2. + (b + a)/2.
        w[i] = w[i]*(b-a)/2.

for M in range(16, 32, 8):
    z=[-1024, -512, -256, -128, -64, -32, -16, -8, -4, -2]
    for lmbda in z:
        A = zeros((M,M), float)                          # Hamiltonian
        WR = zeros((M), float)                  # Eigenvalues, potential
        k = zeros((M), float); w = zeros((M),float);          # Pts & wts
        gauss(M, min1, max1, k, w)                  # Call gauss points
        for i in range(0,M):
            # Set Hamiltonian
            for j in range(0,M):
                VR = lmbda/2/u*sin(k[i]*b)/k[i]*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
        Es, evectors = eig(A)
        realev = Es.real                              # Real eigenvalues
        for j in range(0,M):
            if (realev[j]<0):
                print(" M (size), lmbda, ReE=",M," ",lmbda," ",realev[j])
                break
print("Enter and return any character to quit")
s = input()
```

1. Write your own program, or modify the linked-in one, to solve the integral equation (20.9) for the delta-shell potential (20.16). Either evaluate the determinant of $[H - EI]$ and then find the E for which the determinant vanishes *or* find 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 λ for bound states. A good approach is to start with a large negative value for λ and then make it less negative. You should find that 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. The true bound state will be at negative energy and will be stable as the number of grid points changes. The others are numerical artifacts.
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

Figure 20.3 *Left:* A projectile (dark particle at r) scattering from a dense medium. *Right:* The same process viewed in the CM system where the projectile and target always have equal and opposite momenta.



$[H][\psi] = E[\psi]$.

9. Verify that, regardless of the potential's strength, there is only a single bound state and that it gets deeper as the magnitude of λ increases.

20.2.4 Wave Function (Exploration)

1. Determine the momentum-space wave function $\psi(k)$. Does it fall off at $k \to \infty$? Does it oscillate? Is it well behaved at the origin?
2. Determine the coordinate-space wave function via the Bessel transform

$$\psi(r) = \int_0^\infty dk \psi(k) \frac{\sin(kr)}{kr} k^2. \tag{20.18}$$

Does $\psi_0(r)$ fall off as 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:

$$\psi_0(r) \propto \begin{cases} e^{-\kappa r} - e^{\kappa r}, & \text{for } r < b, \\ e^{-\kappa r}, & \text{for } r > b. \end{cases} \tag{20.19}$$

20.3 UNIT II. NONLOCAL POTENTIAL SCATTERING ⊙

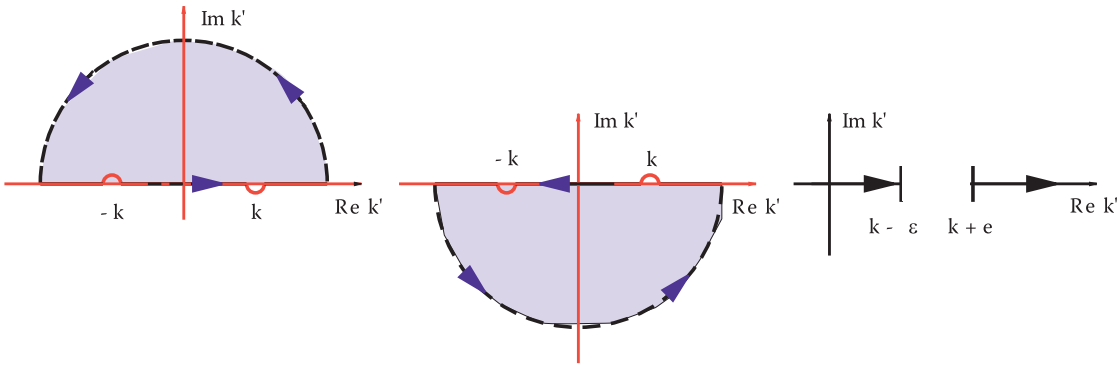
Problem: Again we have a particle interacting with the nonlocal potential discussed in Unit I (Figure 20.3 left), only now the particle has sufficiently high energy that it scatters rather than binds with the medium. Your **problem** is to determine the scattering cross section for scattering from a nonlocal potential.

20.4 LIPPMANN–SCHWINGER EQUATION (THEORY)

Because experiments measure scattering amplitudes and not wave functions, it is more direct to have a theory dealing with amplitudes rather than wave functions. An integral form of the Schrödinger equation dealing with the reaction amplitude or R matrix is the *Lippmann–Schwinger equation*:

$$R(k', k) = V(k', k) + \frac{2}{\pi} \mathcal{P} \int_0^\infty dp \frac{p^2 V(k', p) R(p, k)}{(k_0^2 - p^2)/2\mu}. \tag{20.20}$$

Figure 20.4 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'.



As in Unit I, this equation is for partial wave $l = 0$ and $\hbar = 1$. In (20.20) the momentum k_0 is related to the energy E and the reduced mass μ by

$$E = \frac{k_0^2}{2\mu}, \quad \mu = \frac{m_1 m_2}{m_1 + m_2}, \tag{20.21}$$

and the initial and final COM momenta k and k' are the momentum-space variables. The experimental observable that results from a solution of (20.20) is the diagonal matrix element $R(k_0, k_0)$, which is related to the scattering phase shift δ_0 and thus the cross section:

$$R(k_0, k_0) = -\frac{\tan \delta_l}{\rho}, \quad \rho = 2\mu k_0. \tag{20.22}$$

Note that (20.20) is not just the evaluation of an integral; it is an integral equation in which $R(p, k)$ is integrated over all p, yet since $R(p, k)$ is unknown, the integral cannot be evaluated until after the equation is solved! The symbol $\mathcal{P}$ in (20.20) indicates the Cauchy principal-value prescription for avoiding the singularity arising from the zero of the denominator (we discuss how to do that next).

20.4.1 Singular Integrals (Math)

A *singular* integral

$$\mathcal{G} = \int_a^b g(k)\, dk, \tag{20.23}$$

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 still finite. (If the integral itself were infinite, we could not compute it.) Unfortunately, computers are notoriously bad at dealing with infinite numbers, and if an integration point gets too near the singularity, overwhelming subtractive cancelation or overflow may occur. Consequently, we apply some results from complex analysis before evaluating singular integrals numerically.[3]

In Figure 20.4 we show three ways in which the singularity of an integrand can be avoided. The paths in Figures 20.4A and 20.4B move the singularity slightly off the real k axis by giving the singularity a small imaginary part $\pm i\epsilon$. The Cauchy principal-value prescription $\mathcal{P}$ (Figure 20.4C) is seen to follow a path that "pinches" both sides of the singularity

[3] [S&T 93] describe a different approach using Maple and Mathematica.

at k_0 but does not to pass through it:

$$\mathcal{P}\int_{-\infty}^{+\infty} f(k)\,dk = \lim_{\epsilon\to 0}\left[\int_{-\infty}^{k_0-\epsilon} f(k)\,dk + \int_{k_0+\epsilon}^{+\infty} f(k)\,dk\right]. \tag{20.24}$$

The preceding three prescriptions are related by the identity

$$\int_{-\infty}^{+\infty} \frac{f(k)\,dk}{k-k_0\pm i\epsilon} = \mathcal{P}\int_{-\infty}^{+\infty} \frac{f(k)\,dk'}{k-k_0} \mp i\pi f(k_0), \tag{20.25}$$

which follows from Cauchy's residue theorem.

20.4.2 Numerical Principal Values

A numerical evaluation of the principal value limit (20.24) is awkward because computers have limited precision. A better algorithm follows from the theorem

$$\mathcal{P}\int_{-\infty}^{+\infty} \frac{dk}{k-k_0} = 0. \tag{20.26}$$

This equation says 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 $k\to -k$ permits us to rewrite (20.26) as

$$\mathcal{P}\int_{0}^{+\infty} \frac{dk}{k^2-k_0^2} = 0. \tag{20.27}$$

We observe that the principal-value exclusion of the singular point's contribution to the integral is equivalent to a simple subtraction of the zero integral (20.27):

$$\mathcal{P}\int_{0}^{+\infty} \frac{f(k)\,dk}{k^2-k_0^2} = \int_{0}^{+\infty} \frac{[f(k)-f(k_0)]\,dk}{k^2-k_0^2}. \tag{20.28}$$

Notice that there is no $\mathcal{P}$ on the RHS of (20.28) because the integrand is no longer singular at $k=k_0$ (it is proportional to the df/dk) and can therefore be evaluated numerically using the usual rules. The integral (20.28) is called the *Hilbert transform* of f and also arises in inverse problems.

20.4.3 Reducing Integral Equations to Matrix Equations (Method)

Now that we can handle singular integrals, we can go about reducing the integral equation (20.20) to a set of linear equations that can be solved with matrix methods. We start by rewriting the principal-value prescription as a definite integral [H&T 70]:

$$R(k',k) = V(k',k) + \frac{2}{\pi}\int_0^\infty dp\,\frac{p^2V(k',p)R(p,k) - k_0^2V(k',k_0)R(k_0,k)}{(k_0^2-p^2)/2\mu}. \tag{20.29}$$

We convert this integral equation to linear equations by approximating the integral as a sum over N integration points (usually Gaussian) k_j with weights w_j:

$$R(k,k_0) \simeq V(k,k_0) + \frac{2}{\pi}\sum_{j=1}^{N} \frac{k_j^2 V(k,k_j) R(k_j,k_0) w_j}{(k_0^2-k_j^2)/2\mu}$$

$$-\frac{2}{\pi}k_0^2 V(k,k_0) R(k_0,k_0) \sum_{m=1}^{N} \frac{w_m}{(k_0^2-k_m^2)/2\mu}. \tag{20.30}$$

We note that the last term in (20.30) implements the principal-value prescription and cancels the singular behavior of the previous term. Equation (20.30) contains the $(N+1)$ unknowns $R(k_j,k_0)$ for $j=0,N$. We turn it into $(N+1)$ simultaneous equations by evaluating it for $(N+1)$ k values on a grid (Figure 20.2) consisting of the observable momentum k_0 and the integration points:

$$k = k_i = \begin{cases} k_j, & j=1,N \quad \text{(quadrature points)}, \\ k_0, & i=0 \quad \text{(observable point)}. \end{cases} \tag{20.31}$$

There are now $(N+1)$ linear equations for $(N+1)$ unknowns $R_i \equiv R(k_i,k_0)$:

$$R_i = V_i + \frac{2}{\pi}\sum_{j=1}^{N} \frac{k_j^2 V_{ij} R_j w_j}{(k_0^2-k_j^2)/2\mu} - \frac{2}{\pi}k_0^2 V_{i0} R_0 \sum_{m=1}^{N} \frac{w_m}{(k_0^2-k_m^2)/2\mu}. \tag{20.32}$$

We express these equations in matrix form by combining the denominators and weights into a single denominator vector D:

$$D_i = \begin{cases} +\frac{2}{\pi}\frac{w_i k_i^2}{(k_0^2-k_i^2)/2\mu}, & \text{for } i=1,N, \\ -\frac{2}{\pi}\sum_{j=1}^{N}\frac{w_j k_0^2}{(k_0^2-k_j^2)/2\mu}, & \text{for } i=0. \end{cases} \tag{20.33}$$

The linear equations (20.32) now assume that the matrix form

$$R - DVR = [1-DV]\,R = V, \tag{20.34}$$

where R and V are *column vectors* of length $N+1$:

$$[R] = \begin{pmatrix} R_{0,0} \\ R_{1,0} \\ \ddots \\ R_{N,0} \end{pmatrix}, \quad [V] = \begin{pmatrix} V_{0,0} \\ V_{1,0} \\ \ddots \\ V_{N,0} \end{pmatrix}. \tag{20.35}$$

We call the matrix $[1-DV]$ in (20.34) the wave matrix F and write the integral equation as the matrix equation

$$[F][R] = [V], \qquad F_{ij} = \delta_{ij} - D_j V_{ij}. \tag{20.36}$$

With R the unknown vector, (20.36) is in the standard form $AX = B$, which can be solved by the mathematical subroutine libraries discussed in Chapter 8, "Solving Systems of Equations with Matrices; Data Fitting."

20.4.4 Solution via Inversion, Elimination

An elegant (but alas not efficient) solution to (20.36) is by matrix inversion:

$$[R] = [F]^{-1}[V]. \tag{20.37}$$

Because the inversion of even complex matrices is a standard routine in mathematical libraries, (20.37) is a *direct solution* for the R matrix. Unless you need the inverse for other purposes (like calculating wave functions), a more efficient approach is to use Gaussian *elimination* to find an $[R]$ that solves $[F][R] = [V]$ without computing the inverse.

Listing 20.2 **Scatt.py** 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 matrix library routines.

```
# Scatt.py:  Soln of Lippmann Schwinger in p space for scattering

from visual.graph import *
from gauss import gauss                          # gauss.pyc for gauss method
import numpy.linalg as lina              # Numpy's LinearAlgebra will be lina

graphscatt = gdisplay(x=0, y=0, xmin=0, xmax=6,ymin=0, ymax=1, width=600, height=400,
title='S wave cross section vs E', xtitle='kb', ytitle='[sin(delta)]**2')
sin2plot = gcurve(color=color.yellow)

M = 27;              b = 10.0;            n = 26
k = zeros((M),float);         x = zeros((M),float);   w = zeros((M),float)
Finv = zeros((M,M),float);  F = zeros((M,M), float); D = zeros((M),float)
V = zeros((M), float);        Vvec = zeros((n+1,1),float)
scale = n/2;                  lambd = 1.5
gauss(n, 2, 0., scale, k, w)                          # Set up points & wts
ko = 0.02

for m in range(1,901):
    k[n] = ko
    for i in range (0, n):  D[i]=2/pi*w[i]*k[i]*k[i]/(k[i]*k[i]-ko*ko) #D
    D[n] = 0.
    for  j in range(0,n):   D[n]=D[n]+w[j]*ko*ko/(k[j]*k[j]-ko*ko)
    D[n] = D[n]*(-2./pi)
    for i in range(0,n+1):                  # Set up F matrix and V vector
        for j in range(0,n+1):
            pot = -b*b * lambd * sin(b*k[i])*sin(b*k[j])/(k[i]*b*k[j]*b)
            F[i][j] = pot*D[j]                                      # Form F
            if i==j: F[i][j] = F[i][j] + 1.
        V[i] = pot                                            # Define V
    for  i in range(0,n+1):  Vvec[i][0]= V[i]
    Finv = lina.inv(F)                     # Use LinearAlgebra fir inverse
    R = dot(Finv, Vvec)                                  # Matrix multiply
    RN1 = R[n][0]
    shift = atan(-RN1*ko)
    sin2 = (sin(shift))**2
    sin2plot.plot(pos = (ko*b,sin2))                  # Plot sin**2(delta)
    ko = ko + 0.2*pi/1000.
print("Done")
```

20.4.5 Scattering Implementation

For the scattering problem, we use the same delta-shell potential (20.16) discussed in §20.2.2 for bound states:

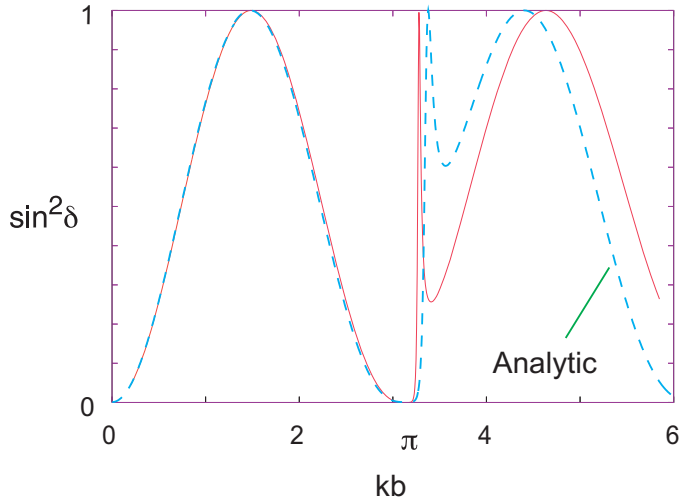
$$V(k', k) = \frac{-|\lambda|}{2\mu k' k} \sin(k'b) \sin(kb). \tag{20.38}$$

This is one of the few potentials for which the Lippmann–Schwinger equation (20.20) has an analytic solution [Gott 66] with which to check:

$$\tan\, \delta_0 = \frac{\lambda b \sin^2(kb)}{kb - \lambda b \sin(kb) \cos(kb)}. \tag{20.39}$$

Our results were obtained with $2\mu = 1$, $\lambda\, b = 15$, and $b = 10$, the same as in [Gott 66]. In Figure 20.5 we give a plot of $\sin^2 \delta_0$ *versus* kb, which is proportional to the scattering cross section arising from the $l = 0$ phase shift. It is seen to reach its maximum values at energies corresponding to resonances. In Listing 20.2 we present our program for solving the scattering integral equation using the Numpy Linear Algebra matrix library and the **gauss** method for quadrature points. For your implementation:

Figure 20.5 The energy dependence of the cross section for $l = 0$ scattering from an attractive delta-shell potential with $\lambda b = 15$. The dashed curve is the analytic solution (20.39), and the solid curve results from numerically solving the integral Schrödinger equation, either via direct matrix inversion or via LU decomposition.



1. Set up the matrices `V[]`, `D[]`, and `F[][]`. Use at least $N = 16$ Gaussian quadrature points for your grid.
2. Calculate the matrix F^{-1} using a library subroutine.
3. Calculate the vector R by matrix multiplication $R = F^{-1}V$.
4. Deduce the phase shift δ from the $i = 0$ element of R:

$$R(k_0, k_0) = R_{0,0} = -\frac{\tan\delta}{\rho}, \quad \rho = 2\mu k_0. \tag{20.40}$$

5. Estimate the precision of your solution by increasing the number of grid point in steps of two (we found the best answer for $N = 26$). If your phase shift changes in the second or third decimal place, you probably have that much precision.
6. Plot $\sin^2\delta$ *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 Figure 20.5. Note that a *resonance* occurs when δ_l increases rapidly through $\pi/2$, that is, when $\sin^2\delta_0 = 1$.
7. Check your answer against the analytic results (20.39).

20.4.6 Scattering Wave Function (Exploration)

1. The F^{-1} matrix that occurred in our solution to the integral equation

$$R = F^{-1}V = (1 - VG)^{-1}V \tag{20.41}$$

is actually quite useful. In scattering theory it is known as the *wave matrix* because it is used in expansion of the wave function:

$$u(r) = N_0 \sum_{i=1}^{N} \frac{\sin(k_i r)}{k_i r} F(k_i, k_0)^{-1}. \tag{20.42}$$

Here N_0 is a normalization constant and the R matrix gives standing-wave boundary conditions. Plot $u(r)$ and compare it to a free wave.

Appendix A

Glossary

absolute value- The value of a quantity expressed as a positive number, for example, $|f(x)|$.
accuracy- The degree of exactness provided by a description or theory. *Accuracy* usually refers to an absolute quality, while *precision* usually refers to the number of digits used to represent a number.
address- 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.
algorithm- A set of rules for solving a problem in a finite number of steps. Usually independent of the software or hardware.
allocate- To assign a resource for use, often memory.
alphanumeric- The combination of alphabetic letters, numerical digits, and special characters, such as %, $, and /.
analog- The mapping of a continuous physical observable to numbers, for example, a car's speed to the numbers on its speedometer.
animation- A process in which motion is simulated by presenting a series of slightly different pictures (frames) in succession.
append- To add on, especially at the end of an object or word.
application- A self-contained, executable program containing tasks to be performed by a computer, usually for a practical purpose.
architecture- The overall design of a computer in terms of its major components: memory, processor, I/O, and communication.
archive- To copy programs and data to an auxiliary medium or file system for long-term, compact storage.
argument- A parameter passed from one program part to another, or to a command.
arithmetic unit- The part of the central processing unit that performs arithmetic.
array (matrix)- 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 called an array element.
assignment statement- A command that sets a value to a variable or symbol.
B- The abbreviation for byte (8 bits).
b- The abbreviation for bit (binary integer).
background- (1) A technique of having a programming run at low priority ("in the background") while a higher-priority program runs "in the foreground." (2) The part of a video display not containing windows.
base- The radix of a number system. (For example, 10 is the radix of the decimal system.)
basic machine language- Instructions telling the hardware to do basic operations such as store or add binary numbers.
batch- The running of programs without user interaction, often in the background.
baud- Technically, the number of signal elements per unit time, but often denoting 1 bit per second.
binary- Related to the number system with base 2.
BIOS- Basic input/output system.

bit- Contraction of "binary integer"; the digit 0 or 1 in binary representation.
Boolean algebra- A branch of symbolic logic dealing with logical relations as opposed to numerical values.
boot- To "bootstrap", *i.e.* to start a computer by loading the operating system.
branch- To pick a path within a program based on the values of variables.
bug- A mistake in a computer program or operating system; a malfunction.
bus- A communication channel (a bunch of wires) used for transmitting information quickly among computer parts.
byte- Eight bits of storage. Two bytes used to store a single character in extended unicode.
byte code- Compiled code read by all computer systems but still needing to be interpreted (or recompiled); often in a class file.
cache- Small, very fast memory used as temporary storage between very fast CPU registers and main memory, or between disk and RAM.
calling sequence- The data and setup needed to call a method or subprogram.
central processing unit (CPU)- The part of a computer that accepts and acts on instructions; where calculations are done and communications controlled.
checkpoint- A statement within a program that stops normal execution and provides output to assist in debugging.
checksum- The summation of digits or bits used to check the integrity of data.
child- An object created by a parent object.
class- (1) A group of objects or methods having a common characteristic. (2) A collection of data types and associated methods. (3) An instance of an object. (4) The byte code version of a program.
clock- The electronics that generate periodic signals to control execution.
code- A program or the writing of a program (often compiled).
column- The vertical line of numbers in an array.
column-major order- The method used by Fortran to store matrices in which the leftmost subscript attains its maximum value before the subscript to the right is incremented. (Python, Java and C use row-major order.)
command- A computer instruction; a control signal.
command key- A keyboard key, or combination of keys, that performs a predefined function.
compilation- The translation of a program written in a high-level language to (more) basic language.
compiler- A program that translates source code from a high-level computer language to more basic machine language.
concatenate- To join together two or more objects head to tail.
concurrent processing- The same as parallel processing; the simultaneous execution of several related instructions.
conditional statement- A statement executed only under certain conditions.
control character- A character that modifies or controls the running of a program (e.g., the control key + *c*).
control statement- A statement within a program that transfers control to another section of the program.
copy- To transfer data *without* removing the original.
CPU- See *central processing unit*.
crash- The abnormal termination of a program or a piece of hardware.
cycle time (clock speed)- The time needed for a CPU to execute a simple instruction.
data- Information stored in numerical form; plural of datum.
data dependence- A situation that occurs when two statements are addressing identical storage locations.
dependence- Relation among program statements in which the results depend on the order

in which the statements are executed.

data type- Definitions that permit proper interpretation of a character string.

debug- To detect, locate, and remove mistakes in software or hardware.

default- The assumption made when no specific directive is given.

delete- To remove and leave no record.

DFT- Discrete Fourier transform.

digital- The representation of quantities in discrete form; contrast analog.

dimension of an array The number of elements that may be referenced by an array index. The *logical dimension* is the largest value actually used by the program.

directory- A collection of files given their own name.

discrete- Related to distinct elements.

disk, disc- A circular magnetic medium used for storage.

double precision- The use of two memory words to store a number.

download- To transfer data *from* a remote computer *to* a local computer.

DRAM- See dynamic RAM. Contrast with *SRAM*.

driver- A set of instructions needed to transmit data to or from an external device.

dump- Data resulting from the listing of all information in memory.

dynamic RAM- Computer memory needing frequent refreshment.

E- A symbol for "exponent." To illustrate, `1.97E2` $= 1.97 \times 10^2$.

element- An item of data within an array; a component of a language.

enable- To make a computer part operative.

ethernet- A high-speed local area network (LAN) composed of specific cable technology and communication protocols.

executable program- A set of instructions that can be loaded into a computer's memory and executed.

executable statement- A statement that causes a certain computational action, such as assigning a value to a variable.

fetch- To locate and retrieve information from storage.

FFT- Fast Fourier transform.

flash memory- Memory that does not require power to retain its contents.

floating point- The finite storage of numbers in scientific notation.

FLOP- Floating-point operation.

foreground- Running high-priority programs before lower-priority programs.

Fortran- Acronym for **for**mula **tran**slation; a classic computer language.

fragmentation- File storage in many small, dispersed pieces.

garbage- Meaningless numbers, usually the result of error or improper definition.Also, obsolete data in memory waiting to be removed ("collected").

giga, G- Prefix indicating 10^9, (1 billion in US).

GUI- Graphical user interface; a windows environment.

hard disk- A circular, spinning, storage device using magnetic memory.

hardware- The physical components of a computer system.

hashing- A transformation that converts keystrokes to data values.

heuristic- A trial-and-error approach to problem solving.

hexadecimal- Base 16; {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}.

hidden line surface- The part of a graphics object normally hidden from view.

high-level language- A programming language similar to normal language.

host computer- A central or remote computer serving other computers.

HPC- High-performance computing.

icon- A small on-screen symbol that activates an application.

increment- The amount added to a variable, especially an array index.

index- The symbol used to locate a variable in an array; the subscript.

infinite loop- The endless repetition of a set of instructions.
input- The introduction of data from an external device into main storage.
instructions- Commands to the hardware to do basic things.
instruction stack- The ordered group of instructions currently in use.
interpolation- Finding values between known values.
interpreter- A language translator that sequentially converts each line of source code to machine code and immediately executes each line.
interrupt- A command that stops the execution of a program when an abnormal condition is encountered.
iterate- To repeat a series of steps automatically.
jump- A departure from the linear processing of code; a branch, a transfer.
just-in-time compiler- A program that recompiles a class file into more efficient machine code.
kernel- The inner or central part of a large program or of an operating system that is not modified (much) when run on different computers.
kill- To delete or stop a process.
kilo, k- Prefix indicating 1 thousand, 10^3.
LAN- Local area network.
LAPACK- A linear algebra package (a subroutine library).
language- Rules, representations, and conventions used to communicate information.
LHS- Left-hand side.
library (lib)- A collection of programs or methods usually on a related topic.
linking- Connecting separate pieces of code to form an executable program.
literal- A symbol that defines itself, such as the letter A.
load- To read information into a computer's memory.
load module- A program that is loaded into memory and run immediately.
log in (on)- To sign onto a computer; to begin a session.
loop- A set of instructions executed repeatedly as long as some condition is met.
low-level language- Machine-related commands not for humans.
machine language- Commands understood by computer hardware.
machine precision- The maximum positive number that, when added to the number stored as 1, does not change it.
macro- A single higher-level statement resulting in several lower-level ones.
main method- The section of an application program where execution begins.
main storage- The fast electronic memory; physical memory.
mantissa- Significant digits in a floating-point number; for example, 1.2 in 1.2E3.
mega, M- A prefix denoting a million, or $1,048,576 = 2^{20}$.
method- A subroutine used to calculate a function or manipulate data.
MIMD- Multiple-instruction, multiple-data computer.
modular programming- The technique of writing programs with many reusable independent parts.
modulo (mod)- A function that yields a remainder after the division of numbers.
multiprocessors- Computers with more than one processor.
multitasking- The system by which several jobs reside in a computer's memory simultaneously; they may run in parallel or sequentially.
NAN- Not a number; a computer error message.
nesting- Embedding a group of statements within another group.
object- A software component with multiple parts or properties.
object-oriented programming- A modular programming style focused on classes of data objects and associated methods that interact with the objects.
object program (code)- A program in basic machine language produced by compiling a high-

level language.

octal- Base 8; easy to convert to or from binary.

ODE- Ordinary differential equation.

1-D- One-dimensional.

operating system (OS)- The program that controls a computer and runs applications, processes I/O, and shells.

OOP- Object-oriented programming.

optimization- The modification of a program to make it run more quickly.

overflow- The error resulting from trying to store too large a number.

package- A collection of related programs or classes.

page- A segment of memory that is read as a single block.

parallel (concurrent) processing- Simultaneous or independent processing in different CPUs.

parallelization- Rewriting an existing program to run in parallel.

partition- The section of memory assigned to a program during its execution.

PC- Personal computer.

PDE- Partial differential equation.

physical memory- The fast electronic memory of a computer; main memory; contrast with *virtual memory*.

physical record- The physical unit of data for input or output that may contain a number of logical records.

pipeline (segmented) arithmetic units- An assembly-line approach to central processing; the CPU simultaneously gathers, stores, and processes data.

pixel- A picture element; a dot on the screen. See also *voxel*.

Portable Document Format, .pdf- A document format developed by Adobe that is of high quality and readable by an extended browser.

PostScript, .ps- A language developed by Adobe for printing high-quality text and graphics.

precision- The degree of exactness with which a quantity is presented. High-precision numbers are not necessarily *accurate*.

program- A set of instructions that a computer interprets and executes.

protocol- A set of rules or conventions.

pseudocode- A mixture of normal language and coding that provides a symbolic guide to a program.

queue- An ordered group of items waiting to be acted upon in turn.

radix- The base number in a number system that is raised to powers.

RAM- Random-access (central) memory that is reached directly.

random access- Reading or writing memory independent of storage order.

record- A collection of data items treated as a unit.

recurrence/recursion- Repetition producing new values from previous ones.

registers- Very high-speed memory used by the central processing unit.

reserved words- Words that must be used in a restricted fashion in an application program.

RHS- Right-hand side.

RISC- A CPU design for a reduced instruction set computer.

row-major order- The method used by Python, Java and C to store matrices in which the rightmost subscript varies most rapidly and attains its maximum value before the left subscript is incremented.

run- To execute a program.

scalar- A data value or number, for example, π.

serial/scalar processing- Calculations in which numbers are processed in sequence. Contrast with *vector processing* and *parallel processing*.

shell- A command-line interpreter; the part of the operating system where the user enters

commands.
SIMD- A single instruction, multiple-data computer.
simulation- The modeling of a real system by a computer program.
single precision- The use of a single computer word to store a variable.
SISD- A single-instruction, single-data computer.
software- Programs or instructions.
source code- A program in a high-level language needing compilation to run.
SRAM- See static RAM.
Static RAM.- Memory that retains its contents as long as power is applied. Contrast with *DRAM*.
stochastic- A process in which there is an element of chance.
stride- The number of array elements stepped through as an operation repeats.
string- A connected sequence of characters treated as a single object.
structure- The organization or arrangement of a program or a computer.
subprogram- Part of a program invoked by another program unit; a *subroutine*.
supercomputer- The class of fastest and most powerful computers available.
superscalar- A later-generation RISC computer.
syntax- The rules governing the structure of a language.
TCP/IP- Transmission control protocol/internet protocol.
telnet- Protocols for computer–computer communications.
tera, T- Prefix indicating 10^{12}.
top-down programming- Designing a program from the most general view of the problem down to the specific subroutines.
unary- An operation that uses only one operand; monadic.
underflow- The result of trying to store too small a number.
unit- A device having a special function.
upload- Data transfer *from* a local *to* a remote computer; the opposite of *download*.
URL- Universal resource locator; web address.
utility programs- Programs to enhance other programs or do chores.
vector- A group of numbers in memory arranged in 1-D order.
vector processing- 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 to 2-D and 3-D pictures or graphs.
volume- A physical unit of a storage medium, such as a disk.
voxel- A volume element on a regular 3-D grid.
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.

Appendix B

Installing Python, Matplotlib, NumPy

The codes in this text were developed with Python and the packages matplotlib, NumPy, and Visual. There are versions for Windows, Linux and Macs.

i) Go to
`http://vpython.org/`

ii) Download `Python-n.m.j`, where `n.m.j` refer to the version number. For Windows, install it by double-clicking on the downloaded file. Remember, you need to install Python *before* VPython.

iii) Next, from the same site download Vpython (`VPython-Win-Pyn.m.j.exe` for Windows). Install it.

iv) You should have NumPy installed automatically now.

v) Finally, go to
`http://sourceforge.net/projects/matplotlib/`

vi) Download matplotlib in a version to match the version of Python you installed, and install it.

Appendix C

Software Directories

Applets Directory Contents
(see **index.html**; requires browser or Java **appletviewer**)

Applet	*Chapter*	*Applet*	*Chapter*
Area	1	Parabolic motion	4
The chaotic pendulum	12	Hypersensitive Pendula	12
Planetary orbits	9	Normal Mode	18
String motion	18	Normal Mode	18
Cellular automata for Sierpiński	13	Solitons	18
Spline interpolation	8	Relativistic scattering	9
Lagrange interpolation	8	Young's two slit interference	18
Wavelet compression	11	Starbrite (H-R diagram)	4
HearData: a sound player for data	12	Photoelectric effect	9
Visualizing physics with Sound	9	Create Lissajous figures	12
Radioactive Decay	5	Spline Interpolation	8
Wavepacket-wavepacket collision movies	18	Heat equation	17
ABM predictor corrector	9	Wave function (SqWell), (HarOs)	18
Wave function (Asym), (PotBarrier)	18	Fractals (Sierpiński), (fern), (tree)	13
Fractals (film) (column) (dla)	13	Feynman path integrals (QMC)	15
Four-centers chaotic scattering	12	Shock wave	19
Schrö. Eq, With Split operator	18		

Python Codes Contents by Section

Section	*Program*	*Section*	*Program*
1.6	AreaFormatted	1.6.2	Area
1.6.3	Directives	1.6.5	CommandLineArgs
1.6.6	FileCatchThrow	1.7.5	Limits
2.2.2	Bessel	3.4	EasyMatPlot
3.4	GradesMatPlot	3.4	MatPlot2figs
3.4.3	SurfacePlot	3.5	EasyVisual
3.5	3GraphVisual	3.5	HarmosAnimate
3.5	3Danimate	3.5	3Dshapes
4.4	ComplexDummy	4.4	ComplexOverload
4.4	ComplexSelf	4.7.1	Beats
4.7.2	Moon	4.7.2	OOPPlanet

Python Codes Contents by Section *Continued*

Section	*Program*	*Section*	*Program*
4.12	Ball	4.12	Path
4.12	Baton	4.12	Dumbbell
4.12	LeadBaton	4.14.7	KomplexPolar
4.14.7	KomplexPolarTest	4.18	Accm2D
5.4.2	Walk	5.5.3	Decay
5.5.3	DecaySound	6.2	TrapMethods
6.2.5	IntegGauss	6.7.3	vonNeuman
7.9	Bisection	7.10	NewtonCD
8.4.2	Eigen	8.4.4	NewtonNDanimate
8.5.2	Lagrange	8.5.6	SplineInteract
8.7.1	Fit	9.5.2	rk2
9.5.2	rk4	9.5.2	rk45
9.5.2	ABM	9.11	QuantumNumerov
9.11	QuantumEigen	9.14.2	ProjectileAir
10.4.1	DFTcomplex	10.4.1	DFT
10.4.1	DFTreal	10.8	FFT
10.8	FFTappl	10.7	NoiseSincFilter
11.4.2	CWT	11.5.3	DWT
12.5	Bugs	12.8	LyapLog
12.8.1	Entropy	12.19.1	PredatorPrey
13.4.1	Column	13.3.2	Fern
13.3.2	Fern3D	13.3.2	Tree
13.3.2	Tree2	13.3.2	Column
13.7.1	DLA	13.9	Gameoflife
14.15.4	Tune	14.15.4	Tune4
15.2	Ising	15.6.1	WangLandau
15.8.3	QMC	15.9	QMCbouncer
16.3	MD1D	16.3	MD2D
17.4.2	LaplaceLine	17.14	LaplaceFEM
17.17.4	EqHeat	17.19.1	HeatCNTridiag
18.2.3	EqStringAnimate	18.6.2	HarmosAnimate
18.10	FDTD	18.10	CircPolarztn
19	Soliton	19.9.3	Beam
19.1	AdvecLax	20.2.3	Bound
20.4.5	Scatt		

Animations **Contents**
(requires player VLC or QuickTime for mpeg, avi; browser for gifs)

Directory	*Chapter*	*Directory*	*Chapter*
DoublePendulum (also two pendulums; see also applets)	12	Fractals (see also applets)	13
MapleWaveMovie (need Maple; includes source files)	18	Laplace (DX movie)	17
MD	16	TwoSlits (includes DX source files)	18
2-Dsoliton (includes DX source files)	18, 19	Utilities (scripts, colormaps)	3
Waves (animated gifs need browser)	18		

Appendix D

Compression via DWT with Thresholding

An important application of discrete wavelet transformation is image compression. Anyone who has stored high-resolution digital images or videos knows that such files can be very large. DWT can reduce image size significantly with just a minimal loss of quality by storing only a small number of smooth components and only as many detailed components as needed for a fixed resolution. To compress an image, we compute and store the DWT of each row of the image, setting all wavelet coefficients smaller than a certain threshold to zero (*thresholding*). When the image is needed, the inverse wavelet transform is used to reconstruct each row of the original image. For example, Table D.1 shows file sizes resulting from using DWT for image compression. We note (columns 1 and 2) that there is a factor-of-2 reduction in size arising from storing the DWT rather than the data themselves, and that this is independent of the threshold criteria (it still stores the zeros). After the compression program *WinZip* removes the zeros (columns 3 and 4), we note that the DWT file is a factor of 3 smaller than the compressed original, and a factor of 11 smaller than the noncompressed original.

A usual first step in dealing with digital images is to convert from one picture format to another.[1] We started with a 3073×2304 *pixel* (picture element) **jpg** (Joint Photographic Experts Group) color image. For our example (Figure D.1 left) we reduced the image size to 512×512 (a power of 2) and converted it to gray scale. You can do this with the free program **IrfanView** [irfanview] (Windows), the GNU image editor **The Gimp** (Unix/Linux) [Gimp], or *Netpbm* [Netpbm]. We used Netpbm's utility **jpegtopnm** to convert the **.jpg** to the portable gray map image in Netpbm format:

```
> jpegtopnm marianabw.jpg > mariana.pnm
```
Convert jpeg to pnm

If you open **mariana.pnm** you will find strings such as

```
P5
512 512
255
yvttsojgrvv|yngcawbRQ] ...
```

This format is not easy to work with, so we converted it to **mariana**,

```
> pnmtopnm -plain mariana.pnm>mariana
```
Convert pnm format to integers

Except for the first three lines that contain information about the internal code (width, height, 0 for black, and 255 for white), we now have integers:

```
P2
512 512
255
143 174 209 235 249 250 250 255 255 255 255 255 255 255 255 255 255
255 255 255 255 254 253 253 254 254 255 255 255 255 255 255 255 255
. . .
```

[1] We thank Guillermo Avendanño-Franco for help with image programs.

Table D.1 Compression of a Data File Using DWT with a 20% Cutoff Threshold

Original File	DWT Reconstructed	WinZipped Original	WinZipped DWT
25,749	11,373	7,181	2,447

Figure D.1 *Left:* The original picture of Mariana. *Middle:* The reconstituted picture for a compression ratio of 8 ($\epsilon = 9$). *Right:* The reconstituted picture for a compression ratio of 46 ($\epsilon = 50$). The black dots in the images are called "salt and pepper" and can be eliminated.

Because Java is not very flexible with I/O formats, we wrote a small C program to convert **mariana** to the one-column format **mariana.dat**:

```
 #include <string.h>
 #include <stdlib.h>
// Reads mariana.dat in decimal ascii, form 1-column mariana.dat
main()  {
  int i, j, k;
  char cd[10], c;
  FILE *pf, *pout;
  pf=fopen("mariana", "r");
  pout=fopen("mariana.dat","w");
  fscanf(pf,"%s",&cd);        printf("%s\n",cd);
  fscanf(pf,"%s",&cd);        printf("%s\n",cd);
  fscanf(pf,"%s",&cd);        printf("%s\n",cd);
  fscanf(pf,"%s",&cd);        printf("%s\n",cd);
  for ( k = 0; k < 512; k++ )  {
  for ( j = 0; j < 512; j++ ) fscanf(pf,"%d",&i); fprintf(pout,"%d\n",i);
  }
  fclose(pf);  fclose(pout);
}
```

Now that we have an accessible file format, we compress and expand it:

1. Read the one-column file and form the array **fg[512][512]** containing all the information about the image.
2. Use the program **DaubMariana.py** to apply the Daubechies 4-wavelet transform to each row of **fg**. This forms a new 2-D array containing the transformed matrix.
3. Use **DaubMariana.py** again, now transforming each column of the transformed matrix and saving the result in a different array.
4. Compress the image by selecting the tolerance level **eps=9** and eliminating all signals that are smaller than **eps**:
   if abs(fg[i][j]) < eps, fg[i][j]=0
5. Apply the inverse wavelet transform to the modified array, column by column and then row by row.
6. Save the reconstituted image to **Marianarec** via output redirection:
   > **java DaubMariana > Marianarec**
7. The program also creates the file **comp-info.dat** containing information about the compression:

```
Number of nonzero coefs before compression: 262136
If abs(coef)<9coef=0
Number of nonzero coefficients after compression: 32753
Compression ratio:8.003419534088481
```

If the program is compiled and run again, the resulting output will contain the compression image in a file with many zeros:

```
P2
512  512
255
143 1543 2296 −405 594 −311 50 68 444 −375 0 20 0 274 −138 423
 371 −19 −51 −100 24 0 118 120 62 −59 404 −111 61 −82 306 −255
204 0 81 −60 0 −29 83 −214 55 −12 −34 24 −32 46 −37 29 −14 45 47
. . .
0 0 0 0 0 0 16 −24 0 0 0 −13 0 −32 0 0 0 0 0 9 −29 28 −21 13
0 −11 0 18 9 −18 0 −11 11 0 0 0 0 0 10 −20 0 0 0 0 −11 0 0
0 0 0 0 0 27 −29 0 −18 0 21 24 −54 15 0 0 0 0 0 0 0 0 0 0
−12 9 −9 9 0 19 0 0 0 0 0 0 −9 0 0 0 0 0 0 0 72 −53 0 0 0 0
0 0 44 35 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 −12 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 −19 0 11 0 −18
. . .
```

Although this is an image file, it is not meant for viewing as an image (it will be mainly black).

D.1 MORE ON THRESHOLDING

Often in practical applications, a good number of the wavelet coefficients are nearly equal to zero. When these coefficients are set to zero via some thresholding scheme [D&J 94], DWT files containing long strings of zeros result. Through a type of compression known as *entropy coding*, the amount of memory needed to store files of this sort can be greatly reduced.

Before we go on to compression algorithms, it is important to note that there are different types of thresholding. In *hard thresholding*, a rather arbitrary tolerance is selected, and any wavelet coefficient whose absolute value is below this tolerance is set to zero. Presumably, these small numbers have only small effects on the image. In *soft thresholding*, a tolerance h is selected, and, as before, any wavelet coefficient whose absolute value is below this tolerance is set to zero. However, all other entries d are replaced with $sign(d)\,|\,|d| - h|$. Soft thresholding can be thought of as a translation of the signal toward zero by the amount h. Finally, in *quantile thresholding*, a percentage p of entries are selected, and p percent of those entries with the smallest absolute values are set to zero.

DWT with thresholding is useful in analyzing signals and for compressing signals so that less memory is needed to store the transforms than to store the original signals. However, we have yet to take advantage of the frequent occurrence of zeros as wavelet coefficients. *Huffman entropy coding* is well suited for compressing data that contain many zeros. With this method, an integer sequence q is changed to a shorter sequence e that is stored as 8-bit integers. Strings of zeros are coded by the numbers 1–100, 105, and 106, while nonzero integers in q are coded by 101–104 and 107–254. The idea is to use two or three numbers for coding, with the first being a signal that a large number or a long zero sequence is coming. Entropy coding is designed so that the numbers that are expected to appear the most often in q need the least amount of space in e.

A step in compression, known as *quantization*, converts a sequence w of floating-point numbers to a sequence q of integers. The simplest technique is to round the floats to the nearest integer. Another option is to multiply each number in w by a constant k and then round to the nearest integer. Quantization is called *lossy* because information is lost when a float is

converted to an integer. In Table D.1 we showed the effect of compression using the *WinZip* data compression algorithm. This is a hybrid of LZ77 and Huffman coding also known as *Deflate*.

D.2 WAVELET IMPLEMENTATION AND ASSESSMENT

1. Write a program to plot *Daub4* wavelets. (Our sample program is **`Daub4.py`**.) Observe the behavior of the wavelet functions for different values of the coefficients. In order to do this, place a 1 in the coefficient vector for the wavelet structure you want and place 0's in all other locations. Then perform the inverse transform to produce the physical domain representation of the wavelet.
2. Run the code **`Daub4.py`** for different threshold values.
3. Run the code **`DaubCompress.py`** that uses other functions to give input data.
4. Write a Python program that compresses a 512×512 image using *Daub4* wavelets. To do this, extend method **`wt1`** so that it performs a 2-D wavelet transform. Note that you may need some methods from the **`java.awt.image`** package to plot the images. First, create an **`Image`** object using **`Image img`**. The **`MemoryImageSource`** class is used to create an image from an array of pixel values using the constructor **`MemoryImageSource(int w, int h, int pix[], int depls, int scan)`**. Here **`w`** and **`h`** are the dimensions of the image, **`pix`**[] is an array containing the pixels values, **`depls`** is the deployment of data in **`pix`**[], and **`scan`** is the length of a row in **`pix`**[]. Finally, draw the image using **`drawImage(Image img, int x, int y, this)`**, where **`x`** and **`y`** are the coordinates of the left corner of the image. Alternatively, you can use a program such as Matlab or Maple to display images from an array of integers.
5. Modify the program **`DaubCompress.py`** so that it outputs the DWT to a file.
6. Pick a different function to analyze, the more complicated the better.
7. Plot the resulting DWT data in a meaningful way.
8. Show in your plots the effects of increasing the threshold parameter in order to cut out more of the smaller transform values.
9. Examine the reconstituted signal for various threshold values including zero and note the effect of the cutoff on the image quality.

Bibliography

[Abar 93] ABARBANEL, H. D. I., M. I. RABINOVICH, AND M. M. SUSHCHIK (1993), *Introduction to Nonlinear Dynamics for Physicists,* World Scientific, Singapore. 283

[A&S 72] ABRAMOWITZ, M., AND I. A. STEGUN (1972), *Handbook of Mathematical Functions,* 10th Ed. U.S. Govt. Printing Office, Washington. 122

[Add 02] ADDISON, P. S. (2002), *The Illustrated Wavelet Transform Handbook,* Institute of Physics Publishing, Bristol and Philadelphia. 239, 244, 247

[ALCMD] MORRIS, J., D. TURNER, AND K.-M. HO AL_CMD, *Ames Laboratory Classical Molecular Dynamics*. `http://codeblue.umich.edu/hoomd-blue/`.

[A&T 87] ALLAN, M. P. AND J. P. TILDESLEY (1987), *Computer Simulations of Liquids,* Oxford Science Publications, Oxford, UK. 374

[Amd 67] AMDAHL, G., *Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities*, Proc. AFIPS., 483 (1967). 374

[Anc 02] ANCONA, M. G. (2002), *Computational Methods for Applied Science & Engineering,* Rinton Press, Princeton, N.J. 323

[A&W 01] ARFKEN, G. B., AND H. J. WEBER (2001), *Mathematical Methods for Physicists,* Harcourt/Academic Press, San Diego. 410

[Argy 91] ARGYRIS, J., M. HAASE, AND J. C. HEINRICH (1991), Comput. Meth. Appl. Mech. Eng., **86**, 1. 240, 386

[Arm 91] ARMIN, B., AND H. SHLOMO, EDS. (1991), *Fractals and Disordered Systems*, Springer-Verlag, Berlin.

[Ask 77] ASKAR, A., AND A. S. CAKMAK (1977), J. Chem. Phys. **68**, 2794. 290

[Bai 05] BAILEY, M. *OSU ChromaDepth Scientific Visualization Gallery,* `web.engr.oregonstate.edu/~mjb/chromadepth/`. 430, 433

[Bana 99] BANACLOCHE, J. G., (1999), *A Quantum Bouncing Ball*, Am. J. Phys. **67**, 776-782. 65

[Barns 93] BARNSLEY, M. F. AND L. P. HURD, (1993), *Fractal Image Compression*, A. K. Peters, UK. 370

[Becker 54] BECKER, R. A., (1954), *Introduction to Theoretical Mechanics*, McGraw-Hill, New York. 294, 305

[Berry] BERRYMAN, A. A., *Predator-Prey Dynamics*, `classes.entom.wsu.edu/543/`. 427

[B&R 02] BEVINGTON, P. R., AND D. K. ROBINSON (2002), *Data Reduction and Error Analysis for the Physical Sciences*, 3rd Ed., McGraw-Hill, New York.

[Bin 01] BINDER, K. AND D. W. HEERMANN, (2001) *Monte Carlo Methods*, Springer-Verlag, Berlin. 163, 170, 171

[Bleh 90] BLEHER, S., C. GREBOGI, AND E. OTT (1990), *Bifurcations in Chaotic Scattering*, Physica D, **46**, 87.

[Burde 74] BUNDE, A. AND S. HAVLIN (EDS) (1991), *Fractals and Disordered Systems*, Springer, Berlin. 203

[Burg 74] BURGERS, J. M., (1974), *The Non-Linear Diffusion Equation; Asymptotic Solutions and Stattistical Problems*, Reidel, Boston. 291

[B&H 95] BRIGGS, W. L., AND V. E. HENSON (1995), *The DFT, An Owner's Manual*, SIAM, Philadelphia. 445

[C&P 85] R. CAR AND M. PARRINELLO (1985), Phys. Rev. Lett. **55,** 2471. 216

[C&P 88] CARRIER, G. F., AND C. E. PEARSON (1988), *Partial Differential Equations*, Academic Press, San Diego. 374

[C&S 10] CHABAY, R. W. AND SHERWOOD, B. A. (2010), *Matter and Interactions*, 3rd Edition, Wiley, New York.

[C&L 81] CHRISTIANSEN, P. L., AND P. S. LOMDAHL (1981), Physica **2D,** 482. ii

[C&O 78] CHRISTIANSEN, P. L., AND O. H. OLSEN (1978), Phys. Lett. **68A,** 185; (1979), Physica Scripta **20**, 531.

[CiSE] *Computing in Science & Engineering*, `http://www.computer.org/portal/web/cise/home`.

[CPUG] *CPUG*, Computational Physics degree program for Undergraduates, Oregon State University, `science.oregonstate.edu/ rubin/CPUG`. 2, 312

[C&N 47] CRANK, J., AND P. NICOLSON (1946), Procd. Cambridge Phil. Soc. **43,** 50. 10

[Chrom] ChromaDepth Technologies CHROMADEPTH TECHNOLOGIES `www.chromatek.com/`. 413

[Clark] CLARK UNIVERSITY, *Statistical & Thermal Physics Curriculum Development Project*, `stp.clarku.edu/`; *Density of States of the 2D Ising Model*. 65

[Co,65] COOLEY, J. W. AND J. W. TUKEY, (1965), Math. Comput., **19**, 297. 355, 357

[Cour 28] COURANT, R., K. FRIEDRICHS, AND H. LEWY (1928), Mathematische Annalen **100**, 32. 231

[Cre 81] CREUTZ, M. AND B. FREEDMAN, *A statistical approach to quantum mechanics*, (1981), Ann. Phys. (N.Y.), **132**, 427-462.

[CYG] Cygwin, a Linux-like environment for Windows, `x.cygwin.com/`.

[Da,42] DANIELSON, G. C. AND C. LANCZOS, (1942), J. Franklin Inst., **233**, 365. 65

[Daub 95] DAUBECHIES, I. (1995), *Wavelets and other phase domain localization methods*, Proc. Int. Cong. Mathematicians, **1, 2**, Basel, 56 (1995). Birkhäuser. 231

[DeJ 92] DE JONG, M. L. (1992), *Chaos and the Simple Pendulum,* The Physics Teacher **30**, 115. 254

[Dong 05] DONGARRA, J., T. STERLING, H. SIMON, AND E. STROHMAIER (2005), *High-Performance Computing,* IEEE/AIP Comp. in Science & Engr. **7**, 51. 280

[Dong 11] DONGARRA, J., *On the Future of High Performance Computing: How to Think for Peta and Exascale Computing*, Conference on Computational Physics 2011, Gatlinburg, 2011; *Emerging Technologies for High Performance Computing*, GPU Club presentation, 2011, University of Manchester, `http://www.netlib.org/utk/people/JackDongarra/SLIDES/gpu-0711.pdf`. 322, 325

[Donn 05] DONNELLY, D. AND B. RUST (2005), *The Fast Fourier Transform for Experimentalists,* IEEE/AIP Comp. in Science & Engr. **7**, 71. 331, 342, 343

[D&J 94] DONOHO, D.L. AND I. M. JOHNSTONE, (1994a), *Ideal denoising in an orthonormal basis chosen from a library of bases*, Compt. Rend. Acad. Sci. Paris Ser. A, **319**, 1317. 231

[Eclipse] ECLIPSE, an open development platform, **www.eclipse.org/**. 490

[Erco] ERCOLESSI, F., *A Molecular Dynamics Primer*, **www.ud.infn.it/~ercolessi/md/**. v

[E&P 88] EUGENE, S. H., AND M. PAUL (1988), Nature **335**, 405. 374, 379

[F&S] FALKOVICH, G., AND K. R. SREENIVASAN (2006), *Lesson from Hydrodynamic Turbulence*, Physics Today 43. 290

[Fam 85] FAMILY, F., AND T VICSEK (1985), J. Phys. A **18**, L75. 445, 462

[Feig 79] FEIGENBAUM, M. J. (1979), J. Stat. Physics **21**, 669. 296

262, 265

[F&W 80] FETTER, A. L., AND J. D. WALECKA (1980), *Theoretical Mechanics of Particles and Continua*, McGraw-Hill, New York. 443, 453

[F&H 65] FEYNMAN, R. P., AND A. R. HIBBS (1965), *Quantum Mechanics and Path Integrals*, McGraw-Hill, New York. 345, 361

[Fitz 04] FITZGERALD, R. (2004), *New Experiments Set the Scale for the Onset of Turbulence in Pipe Flow*, Phys. Today, **57**, 21. 462

[Fos 96] FOSDICK L. D, E. R. JESSUP, C. J. C. SCHAUBLE, AND G. DOMIK (1996), *An Introduction to High Performance Scientific Computing*, MIT Press, Cambridge. 374

[Fox 94] FOX, G., *Parallel Computing Works!*, (1994) Morgan Kaufmann, San Diego. 312

[Gara 05] GARA, A., M. A. BLUMRICH, D. CHEN, G. L.-T. CHIU, P. COTEUS, M. E. GIAMPAPA, R. A. HARING, P. HEIDELBERGER, D. HOENICKE, G. V. KOPCSAY, T. A. LIEBSCH, M. OHMACHT, B. D. STEINMACHER-BUROW, T. TAKKEN, AND P. VRANAS, *Overview of the Blue Gene/L system architecure,* (2005) IBM J. Res & Dev **49**, 195. 327, 328, 330

[Gar 00] GARCIA, A. L. (2000), *Numerical Methods for Physics*, 2nd ed., Prentice Hall, Upper Saddle River. 393

[Gibbs 75] GIBBS, R. L. (1975), *The quantum bouncer*, Am. J. Phys., **43**, 25-28. 370

[Good 92] GOODINGS, D. A. AND T. SZEREDI (1992), *The quantum bouncer by the path integral method*, Am. J. Phys., **59**, 924-930. 370

[Gimp] GIMP, the GNU image manipulation program, `www.gimp.org/`. 488

[GNU] Gnuplot, a portable command-line driven interactive data and function plotting utility, `www.gnuplot.info/`.

[Gold 67] GOLDBERG, A., H. M. SCHEY, AND J. L. SCHWARTZ (1967), Am. J. Phys. **3**, 177.

[Gos 99] GOSWANI, J. C., A. K. Chan, (1999) *Fundamentals of Wavelets*, John Wiley & Sons, Inc., New York.

[Gott 66] GOTTFRIED, K. (1966), *Quantum Mechanics*, Benjamin, New York. 140, 471, 477

[G,T&C 06] GOULD, H., J. TOBOCHNIK AND W. CHRISTIAN (2006), *An Introduction to Computer Simulation Methods*, 3rd ed., Addison-Wesley, Reading, MA. 126, 267, 270, 280, 353, 374

[Grace] GRACE, a WYSIWYG 2D plotting tool for the X Window System (descendant of ACE/gr, Xmgr), `plasma-gate.weizmann.ac.il/Grace/`. 65, 66

[Graps 95] GRAPS, A. (1995), *An Introduction to Wavelets,* IEEE/AIP Comp. in Science & Engr. **2**, 50.

[Gurney] GURNEY, W. S. C. AND R. M. NISBET, (1998), *Ecological Dynamics*, Oxford Uni. Press, Oxford, New York.

[H&T 70] HAFTEL, M. I., AND F. TABAKIN (1970), Nucl. Phys. **158**, 1. 475

[Har 96] HARDWICH, J., *Rules for Optimization,* `www.cs.cmu.edu/~jch/java`. 331

[Hart 98] HARTMANN, W. M. (1998), *Signals, Sound, and Sensation*, AIP Press, Springer-Verlag, New York. 227, 228

[Hi,76] HIGGINS, R. J., (1976), Am. J. Phys., **44**, 766. 234

[Hock 88] HOCKNEY, R.W. AND J.W. EASTWOOD (1988), *Computer Simulation Using Particles*, Adam Hilger, Bristol, UK. 374

[Huang 87] HUNAG, K. (1987), *Statistical Mechanics*, Wiley, New York. 348

[irfanview] IRFANVIEW, `www.irfanview.com/`. 488

[Jack 88] JACKSON, J. D. (1988), *Classical Electrodynamics*, 3rd ed., Wiley, New York. 387, 388

[J&S 98] JOSÉ, J. V, AND E. J. SALATAN, (1988) *Classical Dynamics*, Cambridge Univ. Press, Cambridge, UK. 181, 277

[jEdit] JEDIT, a mature programmer's text editor, `www.jedit.org/`. v

[K&R 88] KERNIGHAN, B. AND D. RITCHIE (1988) *The C Programming Language*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ.

[Koon 86] KOONIN, S. E. (1986), *Computational Physics*, Benjamin, Menlo Park, CA. 129, 195, 431, 462

[KdeV 95] KORTEWEG, D. J., AND G. DEVRIES (1895), Phil. Mag. **39**, 4. 449

[Krey 98] KREYSZIG, E. (1998), *Advanced Engineering Mathematics*, 8th ed., Wiley, New York. 390

[Kutz] KUTZ N., *Scientific Computing*, www.amath.washington.edu/courses/581-autumn-2003/.

[Lamb 93] LAMB, H., (1993), *Hydrodynamics*, 6th ed., Cambridge University Press, Cambridge, UK. 460

[L&L,F 87] LANDAU, L. D., AND E. M. LIFSHITZ (1987), *Fluid Mechanics*, 2nd Ed., Butterworth-Heinemann, Oxford, UK. 443

[L&L,M 76] LANDAU, L. D., AND E. M. LIFSHITZ (1976), *Quantum Mechanics*, Pergamon, Oxford, UK. 212, 270, 271, 280, 433, 444

[L&L,M 77] LANDAU, L. D., AND E. M. LIFSHITZ (1976), *Mechanics*, 3rd ed., Butterworth-Heinemann, Oxford, UK.

[L 05] LANDAU, R. H. (2005), *A First Course in Scientific Computing*, Princeton Univ. Press, Princeton. 45, 398

[L 96] LANDAU, R. H. (1996), *Quantum Mechanics II, A Second Course in Quantum Theory*, 2nd ed., Wiley, New York. 196, 371, 468, 469

[Lang] LANG, W. C., K. FORINASH (1998), *Time-frequency analysis with the continuous wavelet transform*, Amer. J. of Phys, **66**, 794. 246

[Lang 08] LANGTANGEN, H. P. (2008), *Python Scripting for Computational Science*, Springer-Verlag, Heidelberg. 2

[Lang 09] LANGTANGEN, H. P. (2009), *A Primer on Scientific Programming with Python*, Springer-Verlag, Heidelberg. 2

[LAP 00] ANDERSON, E., Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, AND D. SORENSEN (2000), *LAPACK Users' Guide*, 3rd ed., SIAM, Philadelphia; www.netlib.org. 3, 47, 49

[Li] LI, Z., *Numerical Methods for Partial Differential Equations - Finite Element Method*, www4.ncsu.edu/~zhilin/. 399, 400, 404

[Libb 03] LIBOFF, R. L., (2003), *Introductory Quantum Mechanics*, Addison Wesley, Reading, MA.

[Lot 25] LOTKA, A. J., (1925) *Elements of Physical Biology*, Williams & Wilkins, Baltimore. 284

[LP&B 08] LANDAU, R. H., P ÁEZ, M. J. AND BORDEIANU, C. C. (2008), *A Survey of Computational Physics*, Princeton University Press, Princeton. 46

[MacK 85] MACKEOWN, P. K. (1985), Am. J. Phys. **53**, 880. 345

[M&N 87] MACKEOWN, P. K., AND D. J. NEWMAN (1987) *Computational Techniques in Physics*, Adam Hilger, Bristol, UK. 345

[MLP 00] MAESTRI, J. J. V., R. H. LANDAU, AND M. J P/'AEZ (2000) *Two-Particle Schrödinger Equation animations of wave packet-wave packet scattering*, , Am. J. Phys. **68**, 1113-1119. 430, 433

[Mallat 89] MALLAT, P.G. (1982), *A Theory for Multiresolution Signal Decomposition: The Wavelet Representation*, IEEE Transaction on Pattern Analysis and Machine Intelligence Vol.11, No. 7, 674-693.

[Mand 67] MANDELBROT, B. (1967), *How long is the coast of Britain?*, Science, **156**, 638. 298

[Mand 82] MANDELBROT, B. (1982), *The Fractal Geometry of Nature*, 29, Freeman, San Francisco. 290

[Mann 90] MANNEVILLE, P., (1990), *Dissipative Structures and Weak Turbulence*, Academic Press, San Diego. 266

[Mann 83] MANNHEIM, P. D. (1983), Am. J. Phys. **51**, 328. 345

[M&T 03] MARION, J. B. AND S. T. THORNTON (2003), *Classical Dynamics of Particles and Systems*, 5th ed., Harcourt Brace Jovanovich, Orlando, FL. 204, 207, 271

[Math 02] MATHEWS, J., (2002), *Numerical Methods for Mathematics, Science and Engineering*, Prentice Hall, Upper Saddle River. 186

[Math 92] MATHEWS, J. (1992), *Numerical Methods for Mathematics, Science and Engineering*, Prentice Hall, Englewood Cliff, NJ. 188

[M&W 65] MATHEWS, J. AND R. L. WALKER (1965), *Mathematical Methods of Physics*, Benjamin, Reading, MA. 170

[Metp 53] METROPOLIS, M., A. W. ROSENBLUTH, M. N. ROSENBLUTH, A. H. TELLER, AND E. TELLER (1953), J. Chem. Phys. **21**, 1087. 349

[Mold] REFSON, K. *Moldy, A General-Purpose Molecular Dynamics Simulation Program*, `www.ccp5.ac.uk/moldy/moldy.html`. 374

[M&L 85] MOON, F. C. AND G.-X. LI (1985), Phys. Rev Lett. **55**, 1439. 284

[M&F 53] MORSE, P. M., AND H. FESHBACH (1953), *Methods of Theoretical Physics*, McGraw-Hill, New York. 387

[NAMD] NELSON, M., W. HUMPHREY, A. GURSOY, A. DALKE, L. KALE, R. D. SKEEL, AND K. SCHULTEN, (1996), *NAMD - Scalable Molecular Dynamics*, J. of Supercomputing Applications and High Performance Computing, `www.ks.uiuc.edu/Research/namd/`. 374

[Nes 02] NESVIZHEVSKY, V.V., H. G. BORNER, A. K. PETUKHOV, H. ABELE, S. BAESSLER, F. J. RUESS, T. STOFERLE, A. WESTPHAL, A. M. GAGARSKI, G. A. PETROV, AND A. V. STRELKOV, *Quantum states of neutrons in the Earth's gravitational field*, (2002), Nature **415**, 297. 370

[Netpbm] NETPBM, a package of graphics programs and programming library, `netpbm.sourceforge.net/doc/`. 488

[Ott 02] OTT, E., (2002), *Chaos in Dynamical Systems*, Cambridge University Press, Cambridge. 267

[Otto] OTTO A., *Numerical Simulations of Fluids and Plasmas* `how.gi.alaska.edu/ao/sim/chapters/chap6.pdf`. 399

[Pan 96] PANCAKE, C. M., (1996), *Is Parallelism for You?*, IEEE Computational Sci. & Engr, **3**, 18. 312, 325

[P&D 81] PEDERSEN, N. F., AND A. DAVIDSON (1981), Appl. Phys. Lett. **39**, 830.

[Peit 94] PEITGEN, H.-O., H. JÜRGENS, AND D. SAUPE (1992), *Chaos and Fractals*, Springer-Verlag, New York. 305

[Penn 94] PENNA, T. J. P. (1994), Comput. in Phys. **9**, 341.

[Perlin] PERLIN, K., NYU Media Research Laboratory, `mrl.nyu.edu/~perlin`. 307, 309

[P&R 95] PHATAK, S. C., AND S. S. RAO (1995), *Logistic map: A possible random-number generator,* Phys. Rev. E **51**, 3670. 265

[PhT 88] PHYSICS TODAY, Special issue on chaos, December 1988. 290

[P&B 94] PLISCHKE, M., AND B. BERGERSEN (1994), *Equilibrium Statistical Physics*, 2nd Ed., World Scientific Pub. Co., Singapore. 348

[Polikar] POLIKAR, R., *The Wavelet Tutorial*, `users.rowan.edu/~polikar/WAVELETS/WTtutorial.html`.

[Poll 06] KENNEDY, R., *The Case of Pollocks Fractals Focuses on Physics*, New York Times, 2 December 2006; 5 December 2006. 304

[Potv 93] POTVIN, J. (1993), Comput. in Phys. **7**, 149. 345

[Pov-Ray] *Persistence of Vision Raytracer*, `www.povray.org`. 309

[Pres 94] PRESS, W. H., B. P. FLANNERY, S. A. TEUKOLSKY, AND W. T. VETTERLING (1994), *Numerical Recipes*, Cambridge University Press, Cambridge, UK. 129, 134, 146, 166, 167, 170, 172, 181, 186, 228, 393, 410, 415, 422, 446

[Pres 00] 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. 129, 148, 181, 188, 371

[Quinn 04] QUINN, M. J. (2004), *Parallel Programming in C with MPI and OpenMP*, McGraw Hill Higher Education, New York, NY. 312, 319, 323

[Ram 00] RAMASUBRAMANIAN, K. AND M. S. SRIRAM, (2000), *A comparative study of computation of Lyapunov spectra with different algorithms*, Physica D, **139** 72. 266

[Rap 95] RAPAPORT, D.C (1995), *The Art of Molecular Dynamics Simulation*, Cambridge University Press, Cambridge, UK. 374

[Rash 90] RASBAND, S. N. (1990), *Chaotic Dynamics of Nonlinear Systems*, Wiley, New York. 260, 261, 270

[Raw 96] RAWITSCHER, G., I. KOLTRACHT, H. DAI, AND C. RIBETTI (1996), Comput. in Phys., **10**, 335. 418

[R&M93] REITZ, J.R., F. J. MILFORD, AND CHRISTY, R. W. (1993), *Foundations of Electromagnetic Theory*, Fourth Ed., Addison-Wesley, Reading, PA.

[Rey 83] REYNOLDS, O. (1883), Proc. R. Soc. London, **35**, 84. 462

[Rhei 74] RHEINBOLD, W. C. (1974), *Methods for Solving Systems of Nonlinear Equations*, SIAM, Philadelphia.

[Rich 61] RICHARDSON. L. F., (1961), *Problem of contiguity: an appendix of statistics of deadly quarrels,* General Systems Yearbook, **6**, 139. 298

[Riz] RIZNICHENKO G. Y., *Mathematical Models in Biophysics*. http://www.biophysics.org/Portals/1/PDFs/Education/galina.pdf.

[Rowe 95] ROWE, A. C. H. AND P. C. ABBOTT (1995), *Daubechies Wavelets and Mathematica*, Comput. in Phys. **9**, 635-548.

[Russ 44] RUSSELL, J. S. (1844), *Report of the 14th Meeting of the British Association for the Advancement of Science*, John Murray, London. 443

[Sand 94] SANDER, E., L. M. SANDER, AND R. M. ZIFF (1994), Comput. in Phys. **8**, 420. 290

[Schd 00] SCHMID, E. W., G. SPITZ, AND W. LÖSCH (2000), *Theoretical Physics on the Personal Computer*, 2nd Ed., Springer-Verlag, Berlin.

[Schw 02] SCHWARZCHILD, B., (2002), Phys. Today, **55**, 20. 195

[Schk 94] SCHECK, F. (1994), *Mechanics, from Newton's Laws to Deterministic Chaos*, 2nd ed., Springer-Verlag, New York.

[Shannon 48] SHANNON, C. E., , (1948), *The Bell System Technical Journal,* **27,** 379. 181, 271

[Shar] SHAROV, A., *Quantitative Population Ecology*. http://home.comcast.net/ sharov/PopEcol/popecol.html. 267

[Shaw 92] SHAW C. T. (1992), *Using Computational Fluid Dynamics*, Prentice Hall, Englewood Cliff, NJ.

[S&T 93] SINGH, P. P., AND W. J. THOMPSON (1993), Comput. in Phys. **7**, 388. 399, 443

[Sipp 96] SIPPER., M. (1997), *Evolution of Parallel Cellular Machines* Springer-Verlag, Heidelberg; cell-auto.com/. 474

[Smi 91] SMITH, D. N. (1991), *Concepts of Object-Oriented Programming*, McGraw-Hill, New York. 305

[Smi 99] SMITH, S. W. (1999), *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, San Diego, California. 88, 230

[Stez 73] STETZ, A., J. CARROLL, N. CHIRAPATPIMOL, 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. 229

[Sull 00] SULLIVAN, D. (2000), *Electromagnetic Simulations Using the FDTD Methods*, IEEE Press, New York. 167, 169

[SunJ] SUN JAVA DEVELOPER'S SITE, java.sun.com/. 435, 437, 438

[Tab 89] TABOR, M. (1989), *Chaos and Integrability in Nonlinear Dynamics*, Wiley, New York. v

[Taf 89] TAFLOVE, A AND S. HAGNESS. (2000), *Computational Electrodynamics: The Finite Difference Time Domain Method*, 2nd Ed., Artech House, Boston. 181, 284, 446

[Tait 90] TAIT, R. N., T. SMY, AND M. J. BRETT (1990), Thin Solid Films **187**, 375. 437, 438

[Thij 99] THIJSSEN J. M. (1999), *Computational Physics*, Cambridge University Press, Cambridge, UK. 301

[Thom 92] THOMPSON, W. J. (1992), *Computing for Scientists and Engineers*, Wiley, New York. 374, 378

[Tick 04] TICKNER, J. (2004), *Simulating nuclear particle transport in stochastic media using Perlin noise functions*, Nucl. Instru. & Mtds, B, **203**, 124. 166, 170, 172

[Vall 00] VALLÉE, O., (2000) *Comment on a quantum bouncing ball by Julio Gea Banacloche*, Am J. Phys., **68**, 672-673. 307

[VdeV 94] VAN DE VELDE, E. F. (1994), *Concurrent Scientific Computing*, Springer-Verlag, New York. 370

[VdB 99] VAN DEN BERG, J. C. (ED.) (1999), *Wavelets in Physics*, Cambridge University Press, Cambridge. 312

[Vida 99] VIDAKOVIC, B. (1999), *Statistical Modeling by Wavelets*, Wiley. 244

[Viss 91] VISSCHER, P. B. (1991), Comput. in Phys. **5**, 596.

[Vold 59] VOLD, M. J. (1959), J. Collod. Sci. **14**, 168. 430, 431

[Volt 26] VOLTERRA, V. (1926) *Variazioni e fluttuazioni del numero d'individui in specie animali conviventi*, Mem. R. Accad. Naz. dei Lincei. Ser. VI, **2**. 296

[VUE] VUE *The Visual Understanding Environment*, a tool for managing and integrating digital resources in support of teaching, learning and research, `vue.tufts.edu`. 284

[Ward 04] WARD, D. W AND K. A. NELSON (2004), *Finite Difference Time Domain (FDTD) Simulations of Electromagnetic Wave Propagation using a Spreadsheet*, ArXiv Physics 0402091 1-8. 8

[WL 04] LANDAU, D. P, S.-H. TSAI AND M. EXLER, (2004), *A new approach to Monte Carlo simulations in statistical physics: Wang-Landau sampling*, Am. J. Phys. **72**, 1294;
LANDAU, D. P, AND F. WANG, (2001), *Determining the density of states for classical statistical models: A random walk algorithm to produce a flat histogram*, Phys. Rev. E **64**, 056101. 435

[WW 04] 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. 355

[Whine 92] WHINERAY, J., (1992), *An energy representation approach to the quantum bouncer*, Am. J. Phys. **60**, 948-950.

[Wiki] WIKIPEDIA, the free encyclopedia, `en.wikipedia.org/`. 370

[Will 97] WILLIAMS, G. P., (1997), *Chaos Theory Tamed*, Joseph Henry Press, Washington, D.C. 230

[W&S 83] WITTEN, T. A., AND L. M. SANDER (1981), Phys. Rev. Lett. **47**, 1400; (1983), Phys. Rev. B **27**, 5686. 266

[Wolf 85] WOLF, A., J. B. SWIFT, H. L. SWINNEY, AND J. A. VASTANO, (1985), *Determining Lyapunov Exponents from a Time Series* Physica D, **16**, 285. 302

[Wolf 83] WOLFRAM S. (1983), *Statistical Mechanics of Cellular Automata*, Rev. Mod. Phys. **55**, 601. 266

[Yang 52] YANG, C. N. (1952), *The Spontaneous Magnetization of a Two-Dimensional Ising Model* Phys. Rev. **85**, 809. 307

[Yee 66] YEE, K. (1966), IEEE Transactions on Antennas and Propagation **AP-14**, 302. 347, 348

[Z&K 65] ZABUSKY, N. J., AND M. D. KRUSKAL (1965), Phys. Rev. Lett. **15**, 240. 436

[Zucker] ZUCKER, M. *The Perlin Noise FAQ*; see too JÖNSSON, A., *Generating Perlin Noise*, http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html. 449

Index

ABM, 189
Abstract data types, 74
Abstraction, 74, 88, 94, 102
Accuracy, 22, 479
ACE/gr, 65–69
Adams-Bashforth-Moulton, 188
Address, 479
Advection, 443–445
Airy functions, 371
Algorithm, 14, 18, 479
Alias, 219
Amdahl's law, 323
Analog, 479
  filters, 227
Animations, 11, 63, 279, 280, 378, 381, 431, 432, 449, 479
Antiferromagnet, 347
Applets, 86, 203, 208, 209
Applications, 479
Architecture, 146, 155, 312–321, 325, 329, 333, 334, 479
Archive, 479
Arithmetic unit, 316–319, 479
Arrays, 152, 479
  dimension, 481
Asymptotes, 262
Attractors, 261–279
  predictable, 273, 277
  strange, 277
Autocorrelation function, 223–227

Backtracking, 144, 150
Ballistic deposition, 296–297, 301
  correlated, 301
Base, 22, 479
BASIC, 13, 14
Basic machine language, 13, 479
Batch, 479
Baud, 479
Beating, 192, 193
Beowulf, 321
Bessel functions, 36, 37, 39, 39, 473
Bias, 24
Bifurcation, 261–268, 279, 280, 282
  diagram, 263
  dimension of, 305
Binary numbers, 21, 479
Binary point, 24
Binning, 264
BIOS, 479
Bisection algorithm, 140–142, 196
Bits, 21, 479, 480
  reversal, 233
Blue Gene, *see* IBM Blue, 329
Boltzmann distribution, 348
Boolean, 23, 480
Boot, 480
Bound states, 140, 142, 194–202, 206, 361, 363–370, 384, 468–472
Boundary conditions, 181, 378, 386
Box counting, 299–301, 304
Box–Muller method, 132–134
Break command, 398
Broadcast, 329
Buffer, 314
Buffers, 314
Burgers' equation, 445–448
Bus, 320, 480
Butterfly operation, 233
Byte, 21, 479, 480
  code, 14, 333, 480

C language, 14, 58
Cache, 314, 340, 480
  data, 340
  misses, 340–341
  programming for, 340–341
Calling sequence, 480
Canonical ensemble, 347, 375
canonical ensemble, 348
Capacitors, 71, 395–398
Catenary, 426–429
Cauchy principal value, 475
Cellular automata, 305–307
Cellular automaton, 305
Central difference, 137
Central processing unit, *see* CPU, 316
Central storage, 314
Chaos, 259, 266, 267, 270, 275–280
  Fourier analysis of, 282
  pendulum, 270
  of pendulum, 270
  phase space, 277
  in phase space, 275–279
Chi squared measure, 171
Child, 102, 480
CISC, 316, 317
Classes, 74–99, 480
  daughter, 85, 96
  member, 102
  members, 102
  multiple, 100
  structure, 102
  subclass, 85
  superclass, 96
  variables, 74–90, 94
Clock speed, 480
Code, 480
Column-major order, 152, 314, 480
Command-line interpreter, 13
Communications, 325
  time, 323
Compilers, 14, 480
  just-in-time, 482
Complex numbers, 72
Composition, 94, 96
Compression, 238
  lossless, 242
Computational
  physics, 1, 7
  science, 1, 7, 8
Computer

basics, 13
languages, 13
Concurrent processing, *see* Parallel
Constructors, 76–101
Control structures, 15, 480
Convolution, 225, 227
Conway, 306
Correlations, 224, 301
auto, 223, 224
coefficient, 172
growth, 301
Courant stability condition, 438, 439, 446, 447
Course grain parallel, 320
Covariance, 172
CPU, 312–318, 324, 329, 333, 340, 480
designs, 316
RISC, 316
time, 317
Crank-Nicolson method, 413–417
Cubic splines, 165, *see* Splines
Cumulative distribution, 133
Curie temperature, 347
Curve fitting, *see* Data fitting
Cycle time, 317, 480

Data, 88
cache, *see* Cache
compression, 238
dependency, 319, 480
encapsulation, 88
fitting, 162, 163
hiding, 98
parallel, 319
shared, 325
streams, 320
structures, 74
types, 22, 72, 74, 94, 481
Daughter class, 85
Deadlock, 327
Decay
exponential, 168
spontaneous, 168
Density of states, 354
Dependency, 319, 480, 481
Deposition, 296
ballistic, 297
correlated ballistic, 301
Derivatives, 135–139, 181
central difference, 364, 380
forward difference, 184
second, 166, 182, 198, 380
Derivatives), 139
DFT, 216, 226
Differential equations, 178–210
algorithms, 183
boundary conditions, 181
dynamical form, 181
Euler's rule, 184
initial conditions, 181
order, 180, 181
partial, 181, *see* PDEs, 386
Runge–Kutta algorithm, 185
types, 180, 386
types of, 180, 386
Differentiation, 135, 136
Diffusion-limited aggregation, 301, 302
Digital, 22, 481
Dimension, 481
array, 152, 481
fractional, 290–293, 299, 300, 304
Hausdorf-Besicovitch, 290
logical, 153
physical, 152, 153
schemes, 153
Directories, 481
Discrete Fourier transform, 216–226
Dispersion, 377, 444, 448, 449
relation, 444, 448
Distributed memory, 321
Dot operator, 76, 77
Double
pendulum, 280–282
precision, 24, 28
Double Pendulum, 280
Double precision, 24, 28, 481
Doubles, 24, 94
`double`, 24, 94
Drag, *see* friction, 208
DRAM, 314, 481
`drand`, 107
Driving force, 194
Duffing oscillator, 284

Eigenenergies, 195
Eigenvalues, 151, 155, 158, 159, 181, 194–202, 223, 363, 429, 468
Electrostatic potential, 388
Elliptic integral, 272
Encapsulation, 88, 98, 102
Entropy, 267
Equations
Korteweg–de Vries, 449
Burgers', 445
differential, 178
discrete, 113, 260
integral, 468, *see* Integral, 469, 474
motion, 204, 208, 209
of motion, 207
Schrödinger, 194, 429
Van der Pool, 283
Ergodic, 349
Errors, 32–44, 137, 138
algorithmic, 33, 40, 120
approximation, *see* algorithmic, 40
empirical, 40, 123
integration, 123
in integration, 120, 123
minimum, 42
multiplicative, 35, 36
N-D integration, 128
random, 33
roundoff, 31, 33, 35–37, 40–42, 44, 105, 117, 121, 124, 128, 183, 184, 451
total, 40
types, 32
Ethernet, 481
Euler's rule, 184, 185, 364, 366
Exceptions, 20
Exchange energy, 346
Executive
system, 14
unit, 314
Exponent, 481
Exponential decay, 112, 168
Extinction, 262
Extract part, 89, 94
Extrapolated difference, 137

Fast Fourier transform, *see* FFT, *see* FFT
Feigenbaum constant, 265
Feigenbaum constants, 265
Ferromagnet, 347
Fetch, 318
Feynman
path integrals, 361–373
postulates, 362
propagator, 361
FFT, 218, 231–234, 481
Filters, 227, 228
analog, 227
digital, 228, 252
sinc, 230
windowed, 228–229
Fine grain, 321
Fine grain parallel, 320
Finite

difference equation, 113
difference time domain, 435–442
differences, 113, 391, 430, 458, 461
elements, 399–406
Fitting, 162–177
best, 163
global, 170, 171
goodness, 171
least squares, 170–177
linear, 174
linear least square, 171–176
local, 170
Newton Raphson, 177
nonlinear, 176–177
Fixed points, 261
Fixed points in maps, 261, 274
Fixed-point numbers, 22
Floating-point numbers, 22, 33, 481
`float`, *see* Floating-point, *see* Floating
FLOP, 481
FLOPS, 190, 329
Fluid Dynamics, 443–467
Forth, 316
Fortran, 14
vs Python, 336
Forward difference, 136
Fourier
analysis, 212
autocorrelation relation, 226
chaos, 282
decompositon, 212
discrete transform, *see* Discrete, 216
fast transform, *see* FFT
fast transform (FFT), 218
integral, 211, 215
PDE solution, 388
PDE solution via, 389
sawtooth, 214
of sawtooth, 214
series, 211, 212, 214
short-time transform, 241
theorem, 212
transform, 211, 215
Fractals, 290–311
coastline, 298
dimension, *see* Dimension, 290
plants, 293, 294
Pollock painting, 304
trees, 296
Friction, 192–193, 274, 278, 283, 453, 462
in oscillations, 193, 194
in waves, 424, 425
in pendulum, 270–277
in projectile motion, 207–208
Functional, 362
integration, 362–373
Functions
complex, 73

Galerkin decomposition, 401–403
Game of Life, 306
Garbage, 32
Gaussian
distribution, 133
elimination, 477
quadrature, 117, 121–122
Gibbs overshoot, 214, 230
Giga, 481
Global Array Languages, 319
Global array languages, 319
Global optimization, 315
Glossary, 479
Gnuplot, 45, 57–65, 394, 395, 430, 452, 466
Grace, 65–69
Granularity, 320
Green's function, 227, 361
Grid points, 123, 438, 447, 448, 451, 472, 477
Growth models, 265, 284–307
Guests, 324, 325
GUI, 481

Hénon–Heiles potential, 284
Half-wave function, 214
Hamilton's principle, 362
Hardware, 312–342
Harmonics, 212
Heat bath, 375
Heat equation, 406–417
Hexadecimal, 481
High performance computing, 312–342
Hilbert transform, 475
HPC, *see* High performance computing
Huygens's principle, 361
Hyperbolic point, 274

I/O, *see* Input/output
IBM Blue Gene, 327–329
IEEE floating point, 22–24
Impedance, 73
Importance sampling, 130
Inheritance, *see* Objects, 96, 102
Initial conditions, 181
Input/output
command line, 19
files, 18
Input/Output (I/O), 17–21
Instances, 74–77
Instructions
stack, 314, 482
streams, 320
Integral equations, 469–478
Integration, 116–135
error, 123
error in, 120, 123
from splines, 167
mapping points, 122
Monte Carlo, 126–134
multi-dimensional, 128
rejection techniques for, 126
scaling, 122
Simpson's rule, 119–121
splines, 167
trapezoid rule, 117–121
variance reduction, 129
via mean value, 127
via von Neumann rejection, 130
von Neumann rejection, 130
Integro-differential equation, 468
Intermittency, 262
Interpolation
Lagrange, 163–165
splines, 166
Interpreter, 14, 482
Inverse, 151
Ising model, 345–361
2-D, 348, 354

Jacobi method, 392
Jacobian matrix, 150
JAMA, 471, 477
Java
matrix storage, 314
virtual machine, 333
Just-In-Time compiler, 333

Kernel, 13, 361, 482
Korteweg-de Vries equation, 449

Lag time, 224
Lagrange interpolation, 163–165

Languages
BASIC, 14
compiled, 14, 480
computer, 13
high-level, 13, 481
interpreted, 14
low-level, 482
LAPACK, 154, 155
Laplace's equation, 388–406, 461
Latency, 314, 323, 329
Lattice computations, 345, 364, 370
Lattice points, *see* Grid points
Lax-Wendroff algorithm, 446–448
Leap frog, *see* Time stepping
Least-squares fitting, 170–177
Length of coastline, 298
Lifetime, 168
Limit cycle, 274
Limit cycles, 277
linalg, 175
Linear
algebra, 146, 157
congruent method, 105
least square fitting, 171
regression, 171
superposition, 181
LinearAlgebra, 154, 172
Link, 14
Linux, 14
Lippmann–Schwinger equation, 473
Load, 14
balance, 325
balancing, 325
module, 14
Logical size, 153
Logistic map, 259–266
Loop unrolling, 337, 339
Lorenz attractor, 284
Lotka-Volterra model, 285–290
Lyapunov coefficients, 266–267

Machine
numbers, 23, 33
precision, 28–482
Macro, 482
Magnetic materials, 345–361
Mantissa, 23, 482
Master and Slave, 326
Matplotlib, 49
Matrices, 146–162, 479
column-major order, 152
computing, 152
diagonalization, 151
equations, 474
inversion, 150, 151, 476–478
row-major order, 483
subroutine libraries, 154–158
tri-diagonal, 414
Maxwell's Equations, 435–442
Mean value theorem, 127
Memory, 312–315, 334
architecture, 146, 313, 314
conflicts, 323, 332
distributed, 321
dynamic allocation, 153
pages, 152
physical, 153
physical , 153
virtual, 315, 318
Message passing, 320–322, 325–327
Messages, 321
passing, 320, 321
Methods
dynamic, 89
override, 96
Metropolis algorithm, 345, 348–351, 354, 364
Microcanonical ensemble, 347, 375
microcanonical ensemble, 348
Microcode, 316
Miller's device, 38
MIMD, 320–322, 325
Mode locking, 194, 277
Molecular dynamics, 374–385
Momentum space, 468–478
Monte Carlo
error in, 128
integration, 126–134
simulations, 36, 104, 109, 126–134, 302, 345, 349, 367, 375, 378
techniques, 104, 112
Mother class, 85
Multiple-core processors, 317
Multiresolution analysis, 249
Multitasking, 315, 324–325

NAN, 26
Navier-Stokes equation, 444, 453–461
Netlib, 155
Neutrons, ultracold, 370
Newton Raphson, 144, 148
algorithm, 142
Newton Raphson backtracking, 144
Newton–Raphson
algorithm, 203
Newton-Cotes methods, 117
Nodes, 166, 320
Noise, 307
Perlin, 307
Perlin addition, 307
reduction, 224
Noise reduction, 223, 224, 230
Nonlinear
dynamics, 259, 261, 290
limit cycles, 277
maps, 260, 266
ODE, 181
oscillations, *see* Oscillations
pendulum, 270
Nonlocal potentials, 468, 473
Nonstatic, *see* Static/Nonstatic, *see* Static/nonstatic
Normal
distribution, 133
mode expansions, 212, 389, 420
numbers, 24
Normal mode expansions, 212, 388
Normal numbers, 24
Numbers
binary, 21
complex, 72–74, 80, 99
fixed-point, 22
floating-point, 22, 481
hexadecimal, 21, 481
IEEE, 24
machine, 23
normal, 24
normal/subnormal, 24
octal, 21
range, 21
ranges of, 21
representation, 21
subnormal, 24
uniform, 108
Numerov method, 197–200
Nyquist criterion, 220
Nyquist-Shannon interpolation, 230

Objects, 72–75, 482
code, 14
composition, 94, 95
hierarchies, 96
inheritance, 96, 102
oriented programming, 71
oriented programs, 81–103
properties, 74

Octal numbers, 21
ODEs, 178, 180–191, 197–200
  second order, 208
One cycle population, 261
OOP, 71, *see* Objects oriented programs
OpenDX, 65, 309, 466
Operands, 314
Operating system, 13, 14
Optimization, 39, 146, 155, 191, 331–342
Oscillations
  anharmonic, 179
  anharmonic/harmonic, 191, 212
  damped, 193
  driven, 194
  Fourier analysis of, 211
  from errors, 163, 165, 230
  harmonic, 190
  in phase space, 273
  isochronous, 190, 191
  nonlinear, 178–211
  of pendulum, 270–284
  populations, 261
`other.`object, 78
Over relaxation, *see* Relaxation
Overdetermined, 151
Overflows, 22, 23, 26–28
Overhead, 323, 325, 334
Overload, 79

Padding of signal, 220
Page, 152, 314
  fault, 315
Parallel computing, 312, 319–329
  granularity, 320
  master, slave, 326
  message passing, 325
  perfect, 326
  performance, 322
  pipeline, 326
  programming, 325
  strategy, 324
  subroutines, 320, 324, 325
  synchronous, 326
  types, 319
  types of, 319
Parallelism, 319
Parent class, 96, 102
Partial differential equations, *see* PDEs, 386
Pascal, 316
Path integration, 345, 362–373
PDEs, 181, 386–467
  elliptic, 388
  explicit solution, 430
  explicit solution of, 430
  hyperbolic, 418
  implicit solution, 430
  nonlinear, 449
  parabolic, 387, 406, 407
  types of, 386
  weak form of, 400
`.pdf`, 483
Pendulum, 270–282
  analytic solution, 271
  bifurcation diagram, 279
  chaotic, 270, 277–279
Performance, *see* Tuning
Period doubling, 261, *see* Bifurcation
Periodic boundary conditions, 378
Perlin Noise, 307
Perlin noise, 307–311
Phantom bit, 24
Phase space, 273–279, 283, 286, 453
Phase transition, 345
Pipeline, 316, 318
Pipelined CPU, 316
Planetary motion, 203, 208–210
Plots, 46–65
  animation, 63
  complex, 60
  contour, 57, 61
  field, 45, 62
  parametric, 83
  phase-space, 83
  surface, 57, 60, 394
  vector, 62
Pointers, 154
Poisson's equation, 388, 391–392, 399, 400
Polymorphism, 89
Population dynamics, 260–263, 265, 265, 268, 284–290
PostScript, 395
`.ps`, *see* PostScript
Potentials
  delta shell, 471
  Lennard-Jones, 376
  momentum space, 471
Pov-Ray, 309
Power
  residue method, 105
Power spectrum, 226
Precision, 32
  empirical, 29
  machine, 28, 29
  tests, 192
  tests of, 192
Predator-prey models, 284–290
Predictor-corrector methods, 188
Primitive data types, 74
Principal values, 475
`print`, 18
Problem solving
  paradigm, 8
Programming, 14, 16
  design, 15
  parallel, 325
  for parallel, 325
  structured, 15
  for virtual memory, 315
  virtual memory, 315
Programs, 486
Projectile motion, 86, 203, 206–208
Prompt
  Gnuplot, 57
Propagator, 365
`.ps`, 395, 483
Pseudocode, 14, 16, 31, 483
Pseudorandom, *see* Random numbers
Pyramid FFT, 251
Pyramid scheme, 250
Python
  vs Fortran, 336
  I/O, 17

Quadrature, 116
Quantum, 140
  bouncer, 370
  mechanics, 197
  scattering, 473
quantum bouncer, 370

Race condition, 327
Radioactive decay, 112
Radix, 22
RAM, 152, 314–340
Random, 104
  generators, 105, 132
  linear congruent, 105
  numbers, 104, 105, 265
  pseudo, 105
  sequences, 104, 105, 107
  tests, 109
Random numbers, 104–109, 265, 293
  generators, 105, 265
  linear congruent, 105
  nonuniform, 132

tests of, 108–109
for random walk, 40, 109, 111, 112, 301–302
Ray tracing, 309
Recursion, 36–39
Reference calls, 75, 94
Registers, 29, 314, 340, 483
working, 29
Rejection techniques, 126, 130, 131, 350
Relaxation, 392–393, 398, 458–466
Resonances, 80, 193
nonlinear, 192
Reynolds number, 462
RISC, 316–329, 483
rk, 184
rk4/rk45, 185–187, 190, 205
RLC circuits, 71
Romberg extrapolation, 125
Root mean square, 109–110
Row-major order, 152, 483
Runge-Kutta, 184–188, 190

Sampling, 126, 216, 349
importance, 130
Sawtooth function, 214
scanf, 58
Scattering, 473, 478
Schrödinger equation, 194–203, 429, 433, 468–478
time dependent, 429
Script, 63
Searching, 135, 196, *see* Trial and error
Section, 318
Section size, 318
Secular equation, 151
Seed, 261
Seeds, 105, 261, 262
Self
affine connection, 293
affinity, 296
limiting, 283
similar, 265, 292, 293
Self similarity, 293
Separatrix, 192, 272, 453
Serial, 323
Serial computing, 320, 323–325
Series summation, 30
Shannon Entropy, 267
Shells, 13
Shock waves, 443–449
Sierpiński gasket, 290–293, 307
Sign bit, 26
Signal processing, 224
Significant figures (parts), 33, 34
Significant figures/parts, 34
SIMD, 320
Simpson's rule, 119, 120
Simulation, 104
Sinc filter, 220, 228
Single precision, 24, 28
single, *see* Single precision, 24
Singular integrals, 474
SISD, 320
SLATEC, 154, 155
Slave, 326
SMP, 317, 319
Solitons, 443–453
crossing, 453
KdeV, 450
water wave, 449
Splines, 166
cubic, 165
natural, 166, 167
Spontaneous decay, 112, 114, 168–170, 172, 260
SRAM, 314
Stable states, 262
Stack, 482
Statements
assignment, 479
Static/nonstatic, 74, 89, 90
Static/Nonstatic methods, 74–98
Statistical Mechanics, 347–348, 374–378
Stochastic, 112
Storage, 318
Strange attractors, 277
Stride, 340, 341, 484
Subnormal numbers, 24
Subroutines, 14, 484
libraries, 146, 154–158
Subscripts, *see* Dimension
schemes, 152
Subtasks, 325
Subtractive cancelation, 34–35, 38, 43, 135, 136, 167, 172, 183, 190, 474
Successive over-relaxation, *see* Relaxation
Supercomputer, 484
Supercomputers, 312–329
Superscalar, 484
Swap space, 313–315
Symmetric multi processor, *see* SMP
Symmetric multiprocessors, 319
Tasks, 319–325
TCP/IP, 484
Template methods, 89
Templates, 89
Texture, 307
Thermodynamics, 345–361, 366, 375, 377–378
Three body problem, 210
Thresholding, 490
Time delay, 206
Time stepping, 406, 408–409, 418, 421–422, 430–432, 445–448
Top-down programming., 15
Transients, 194, 262, 264, 279, 280
Trapezoid rule, 117–119
Trial and error, 135, 140, 149–150, 170, 171, 176–177, 181, 196–197, 200–203, 349, 350, 367, 372, 398, 400
Trial and error searching, 141
Trivial solutions, 151, 471
Tuning, *see* Optimization
Two cycle, 261
Two's complement, 22
Two's compliment, 28
Typename, *see* Data

Uncertainty principle, 240, 241
Underflows, 23, 26–28
Uniform
distribution, 104–105, 107–109, 122, 133, 266
sequences, 105, 108
tests, 108–109
weights, 133
Unix, 13, 14

Value pass, 75
Variables
class, 74–90, 94, 98
dynamic, 89
static/nonstatic, 89
Variance, 129, 172
reduction, 129, 349
Vector processors, 318
Vectors, 157, 318, 398, 484
fields, 435
Velocity Verlet algorithm, 380
Verlet algorithm, 380
Virtual machine, 333
Virtual memory, 152, 315, 318, 332–334
Viscosity, 453, 462

Visualization, 11, 45–65, 264
of vectors, 398
Volume rendering, 46
von Neumann
rejection, 130, 350
stability assessment, 383, 397, 409, 410, 412, 414, 417, 421–423, 437, 438
Vorticity, 459–466

Wang-Landau Sampling (WLS), 354–361
Wave
electromagnetic, 418, 435–442
equation, 418–421, 425, 429
functions, 363, 368, 473, 478
on catenary, 426–429
on string, 418–429
packets, 215, 418, 429–434
shallow water, 449
string, 418
Wavelets, 238–258, 258
basis, 243
basis sets, 243
continuous, 246
Daubechechies, 254
Daubechies, 254
discrete transform (DWT), 247, 256
multiresolution analysis, 249, 251
pyramid scheme, 250
transform, 242
Weak form of PDE, 400
Windows, 14
Word length, 21
Working set size, 332