

Numerical Methods in Economics

Kenneth L. Judd

The MIT Press
Cambridge, Massachusetts
London, England

© 1998 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was set in Times New Roman on the Monotype "Prism Plus" PostScript Imagesetter by Asco Trade Typesetting Ltd., Hong Kong.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Judd, Kenneth L.

Numerical methods in economics / Kenneth L. Judd.

p. cm.

Includes bibliographical references (p.) and index.

ISBN 0-262-10071-1 (hc)

1. Economic—Statistical methods. I. Title

HB137.J83 1998

330'.01'5195—dc21

98-13591

CIP

Dedicated to the memory of Joyce Lewis Judd

Contents

Preface	xv
I INTRODUCTION	1
1 Introduction	3
1.1 What Economists Can Compute	3
1.2 Roles of Computation in Economic Analysis	6
1.3 Computation in Science	13
1.4 Future of Computing	15
1.5 Objectives and Nature of This Book	17
1.6 Basic Mathematics, Notation, and Terminology	20
1.7 Software and Supplemental Material	25
1.8 Further Reading	26
Exercises	27
2 Elementary Concepts in Numerical Analysis	29
2.1 Computer Arithmetic	29
2.2 Computer Processing and Algorithms	31
2.3 Economics of Computation	33
2.4 Efficient Polynomial Evaluation	34
2.5 Efficient Computation of Derivatives	35
2.6 Direct versus Iterative Methods	39
2.7 Error: The Central Problem of Numerical Mathematics	39
2.8 Making Infinite Sequences Finite	41
2.9 Methods of Approximation	44
2.10 Evaluating the Error in the Final Result	45
2.11 Computational Complexity	48
2.12 Further Reading and Summary	50
Exercises	50
II BASICS FROM NUMERICAL ANALYSIS ON R^n	53
3 Linear Equations and Iterative Methods	55
3.1 Gaussian Elimination, LU Decomposition	55
3.2 Alternative Methods	58
3.3 Banded Sparse Matrix Methods	61
3.4 General Sparse Matrix Methods	62

3.5	Error Analysis	66
3.6	Iterative Methods	70
3.7	Operator Splitting Approach	75
3.8	Convergence of Iterative Schemes	77
3.9	Acceleration and Stabilization Methods	78
3.10	Calculating A^{-1}	84
3.11	Computing Ergodic Distributions	85
3.12	Overidentified Systems	88
3.13	Software	88
3.14	Further Reading and Summary	89
	Exercises	89
4	Optimization	93
4.1	One-Dimensional Minimization	94
4.2	Multidimensional Optimization: Comparison Methods	99
4.3	Newton's Method for Multivariate Problems	103
4.4	Direction Set Methods	109
4.5	Nonlinear Least Squares	117
4.6	Linear Programming	120
4.7	Constrained Nonlinear Optimization	121
4.8	Incentive Problems	128
4.9	Computing Nash Equilibrium	133
4.10	A Portfolio Problem	135
4.11	A Simple Econometric Example	137
4.12	A Dynamic Optimization Problem	140
4.13	Software	142
4.14	Further Reading and Summary	142
	Exercises	143
5	Nonlinear Equations	147
5.1	One-Dimensional Problems: Bisection	147
5.2	One-Dimensional Problems: Newton's Method	150
5.3	Special Methods for One-Dimensional Problems	158
5.4	Elementary Methods for Multivariate Nonlinear Equations	159
5.5	Newton's Method for Multivariate Equations	167
5.6	Methods That Enhance Global Convergence	171
5.7	Advantageous Transformations	174

5.8	A Simple Continuation Method	176
5.9	Homotopy Continuation Methods	179
5.10	A Simple CGE Problem	187
5.11	Software	191
5.12	Further Reading and Summary	192
	Exercises	193
6	Approximation Methods	195
6.1	Local Approximation Methods	195
6.2	Ordinary Regression as Approximation	202
6.3	Orthogonal Polynomials	203
6.4	Least Squares Orthogonal Polynomial Approximation	207
6.5	Uniform Approximation	211
6.6	Interpolation	216
6.7	Approximation through Interpolation and Regression	219
6.8	Piecewise Polynomial Interpolation	224
6.9	Splines	225
6.10	Examples	228
6.11	Shape-Preserving Approximation	231
6.12	Multidimensional Approximation	235
6.13	Finite Element Approximations	240
6.14	Neural Networks	244
6.15	Further Reading and Summary	247
	Exercises	248
7	Numerical Integration and Differentiation	251
7.1	Newton-Cotes Formulas	251
7.2	Gaussian Formulas	257
7.3	Singular Integrals	267
7.4	Adaptive Quadrature	269
7.5	Multidimensional Quadrature	269
7.6	Example: Portfolio Problems	277
7.7	Numerical Differentiation	279
7.8	Software	282
7.9	Further Reading and Summary	282
	Exercises	283

8	Monte Carlo and Simulation Methods	285
8.1	Pseudorandom Number Generation	285
8.2	Monte Carlo Integration	291
8.3	Optimization by Stochastic Search	296
8.4	Stochastic Approximation	301
8.5	Standard Optimization Methods with Simulated Objectives	303
8.6	Further Reading and Summary	305
	Exercises	306
9	Quasi-Monte Carlo Methods	309
9.1	Equidistributed Sequences	311
9.2	Low-Discrepancy Methods	313
9.3	Fourier Analytic Methods	321
9.4	Method of Good Lattice Points	325
9.5	Estimating Quasi-Monte Carlo Errors	329
9.6	Acceleration Methods and qMC Schemes	330
9.7	Further Reading and Summary	330
	Exercises	331
III	NUMERICAL METHODS FOR FUNCTIONAL PROBLEMS	333
10	Finite-Difference Methods	335
10.1	Classification of Ordinary Differential Equations	335
10.2	Solution of Linear Dynamic Systems	337
10.3	Finite-Difference Methods for Initial Value Problems	340
10.4	Economic Examples of IVPs	346
10.5	Boundary Value Problems for ODEs: Shooting	350
10.6	Finite-Horizon Optimal Control Problems	351
10.7	Infinite-Horizon Optimal Control Problems and Reverse Shooting	355
10.8	Integral Equations	362
10.9	Further Reading and Summary	365
	Exercises	366
11	Projection Methods for Functional Equations	369
11.1	An Ordinary Differential Equation Example	369
11.2	A Partial Differential Equation Example	375
11.3	General Projection Method	377
11.4	Boundary Value Problems	388

11.5	Continuous-Time Growth Problem	392
11.6	Computing Conditional Expectations	393
11.7	Further Reading and Summary	395
	Exercises	396
12	Numerical Dynamic Programming	399
12.1	Discrete-Time Dynamic Programming Problems	399
12.2	Continuous-Time Dynamic Programming Problems	406
12.3	Finite-State Methods	409
12.4	Acceleration Methods for Infinite-Horizon Problems	415
12.5	Discretization Methods for Continuous-State Problems	424
12.6	Methods for Solving Linear-Quadratic Problems	431
12.7	Continuous Methods for Continuous-State Problems	433
12.8	Parametric Approximations and Simulation Methods	436
12.9	Shape-Preserving Methods	437
12.10	Continuous-Time Problems	440
12.11	Further Reading and Summary	442
	Exercises	443
IV	PERTURBATION METHODS	445
13	Regular Perturbations of Simple Systems	447
13.1	Mathematics of Regular Perturbation Methods	448
13.2	Comparative Statics	451
13.3	Perturbing an IVP	453
13.4	Perturbing a BVP: Comparative Perfect Foresight Dynamics	456
13.5	Continuous-Time Deterministic Control	462
13.6	Stochastic Control	471
13.7	Perturbing Discrete-Time Systems	474
13.8	Perturbing Jump Process Control Problems	480
13.9	Global Quality Test for Asymptotic Approximations	482
	Exercises	484
14	Regular Perturbations in Multidimensional Systems	487
14.1	Multidimensional Comparative Statics and Tensor Notation	487
14.2	Linearization of Multidimensional Dynamic Systems	490
14.3	Locally Asymptotically Stable Multidimensional Control	496

interpolation
belongs to
“parametric
approximations”

value
function
iteration
should
be here

14.4	Perturbations of Discrete-Time Problems	502
14.5	Multisector Stochastic Growth	504
14.6	Further Reading and Summary	509
	Exercises	509
15	Advanced Asymptotic Methods	511
15.1	Bifurcation Methods	511
15.2	Portfolio Choices for Small Risks	513
15.3	Gauge Functions and Asymptotic Expansions	516
15.4	Method of Undetermined Gauges	517
15.5	Asymptotic Expansions of Integrals	522
15.6	Hybrid Perturbation-Projection Methods	528
15.7	Further Reading and Summary	532
	Exercises	533
V	APPLICATIONS TO DYNAMIC EQUILIBRIUM ANALYSIS	535
16	Solution Methods for Perfect Foresight Models	537
16.1	A Simple Autonomous Overlapping Generations Model	538
16.2	Equilibrium in General OLG Models: Time Domain Methods	540
16.3	Fair-Taylor Method	547
16.4	Recursive Models and Dynamic Iteration Methods	549
16.5	Recursive Models with Nonlinear Equation Methods	558
16.6	Accuracy Measures	562
16.7	Tax and Monetary Policy in Dynamic Economies	563
16.8	Recursive Solution of an OLG Model	567
16.9	“Consistent” Capital Income Taxation	568
16.10	Further Reading and Summary	571
	Exercises	571
17	Solving Rational Expectations Models	573
17.1	Lucas Asset-Pricing Model	574
17.2	Monetary Equilibrium	577
17.3	Information and Asset Markets	578
17.4	Commodity Storage Models	581
17.5	A Simple Stochastic Dynamic Growth Model	588
17.6	Projection Methods with Newton Iteration	589

17.7	Fixed-Point Iteration	599
17.8	Time Iteration	601
17.9	Generalizations	602
17.10	Further Reading and Summary	605
	Exercises	606
	References	609
	Index	623

Preface

Computers are increasingly important in human activities. In order to harness the full power of this technology, economists will need to learn a broad range of techniques from the numerical mathematics literature. This book presents these techniques and shows how they can be useful in economic analysis.

This book is the result of my studying, teaching, and using numerical methods in economics. I have used drafts of this book in courses at the Stanford Graduate School of Business, and the Departments of Economics at Stanford University, the University of California at Berkeley, and Tel Aviv University. I thank them for letting me use their students as test subjects in this endeavor.

This book is intended to be suitable for second-year doctorate student in economics. It is assumed that the reader is familiar with just the basics of linear algebra, multivariate calculus, probability, econometrics, optimization, competitive general equilibrium theory, Nash equilibrium, optimal control, and dynamic programming. I draw on this material for examples, and I limit them to be relatively simple.

Many graduate student research assistants have assisted me in this book and the research behind it. I first thank Antonio Bernardo, Clark Burdick, Pamela Chang, Kwang-Soo Cheong, Jess Gaspar, Sy-Ming Guu, Felix Kubler, Michael Lee, Bo Li, Karl Schmedders, Andrew Solnick, and Sevin Yeltekin. I must single out Ben Wang for his exceptionally careful reading of the manuscript and numerous suggestions. I have also benefited from discussions with many others. I must mention William Brock, Larry Christiano, John Geweke, Wouter den Haan, Mordecai Kurz, Michael Magill, Steve Matthews, John Rust, Ariel Pakes, Harry Paarsch, Stan Reiter, Tom Sargent, Chris Sims, and Joseph Traub and apologize to the others who are not listed here.

I thank the Hoover Institution for providing me with the resources and time needed for a long-term project such as this. I also thank the National Science Foundation for its generous support. Most of all, I am indebted to the taxpayers of the state of Wisconsin, whose support of higher education made it possible for me to pursue a life of study and research.

Finally, I am grateful for the understanding and patience of Teresa Terry Judd, who married me despite my absorption in writing this book.

I INTRODUCTION

1 Introduction

Computers are changing economic science, offering economists the opportunity to examine economic issues with far greater generality and precision. Some changes are evolutionary, such as more sophisticated empirical analysis and improvements in applied general equilibrium. More important and radical are the new ways economists are using computation to explore economic theories in far greater detail than pure theory alone can, to determine which problems are important and which are not, and to examine new models of agent behavior and social interactions.

If economists are to make full use of the power of computational technology, they must have a firm grasp of the key mathematical techniques used in computational analyses. This book presents many techniques from numerical analysis and applied mathematics that have been or are likely to be useful in computational analyses of economic problems. The purpose of this text is to teach the reader the basic methods so that he can effectively apply the computer to substantive problems in economics. Along the way the reader will hopefully learn some economics as well.

This chapter describes the book's goals, discusses the many roles of computation in economics, and gives an overview of the topics covered in this book. Computation is not new in economics. However, in recent years economists have expanded the range of problems they can analyze computationally and the roles for computation in economic analysis. We first discuss what economists now know how to compute and how they are applying computational methods. We then present some examples of computational successes in other sciences that illustrate how important computation has become. We next argue that the computational power available today is negligible compared to what will soon be available, and that the advantages of computational approaches will grow rapidly. We conclude this chapter with some basic mathematical notation and theorems we will rely on.

1.1 What Economists Can Compute

Many economic models can now be computed in reliable and efficient fashions. Some methods are well known, but the recent upsurge in interest in computational methods has generated a large amount of new work, allowing us to solve models previously considered intractable. We review this literature to indicate the substantial potential of computational methods.

The computational general equilibrium (CGE) literature is the most mature computational literature in economics. This literature took off in the 1960s; see Shoven and Whalley (1992) and Dixon and Parmenter (1996) for surveys. Recent work has studied variational inequalities, a more general class of problems; see the book by

Nagurney (1993) for recent work in this area and economic applications. The recent work of Brown et al. (1994) and Schmedders (1996) now makes it possible to compute general equilibrium with incomplete asset markets.

Some economists have made use of linear-quadratic models of dynamic equilibrium in both competitive and game-theoretic contexts, with and without perfect information. Even today progress is being made in improving the solution methods for the Riccati equations that arise in linear-quadratic control problems. Hansen and Sargent (1995) and Anderson et al. (1997) presents many of these techniques. The linear-quadratic approach to approximating nonlinear stochastic models, formulated in Magill (1977), has been extensively used in the real business cycle (RBC) literature. Using computational methods, Kydland and Prescott (1982) argued that fairly simple dynamic general equilibrium models can display the type of economic fluctuations we observe in the macroeconomy. Prior to that many argued that macroeconomic data were inconsistent with the standard competitive model, arguing instead for Keynesian alternatives. While the full RBC research program remains controversial and unfinished, it is a major contender in current intellectual battles and is an example of research which relies heavily on computation.

The limitations of the linear-quadratic framework has spurred work on nonlinear dynamic models. Dynamic programming is an integral part of dynamic economic theory. In the economics literature the most common approach to numerical solutions is to discretize a problem and apply methods developed in the 1950s. Recent work has produced far faster methods; see the review in Rust (1996).

Perfect foresight models have been developed to study deterministic dynamic economic equilibrium. These models typically reduce to two-point boundary value problems, mathematical problems for which there are numerous solution methods. The work of Auerbach and Kotlikoff (1985), Fair and Taylor (1983), Bovenburg and Goulder (1991), Boucekkine (1995), Gilli and Pauletto (1998), Juilliard et al. (1998), and Hughes Hallet and Piscatelli (1998) are typical examples of such models.

Many economists have developed computational methods for nonlinear stochastic rational expectations models. Gustafson (1958) began the literature with a study of equilibrium grain storage; Wright and Williams (1982, 1984) and Miranda and Helmburger (1988) developed efficient methods to compute rational expectations equilibrium even in the presence of frequently binding constraints. Tauchen (1986) applied Fredholm integral equation methods to solve asset pricing models. Judd (1992) used projection methods from the numerical analysis literature to develop efficient schemes for solving rational expectations models. Laitner (1984, 1987), Srikant and Basar (1991), Budd et al. (1993), Bensoussan (1988), Fleming (1971), Fleming and Souganides (1986), Judd (1996), and Judd and Guu (1996) have com-

puted high-order Taylor expansions of rational expectations models, including dynamic games. Dixit (1991), Samuelson (1970), and Judd and Guu (1996) derived approximation methods for asset problems. Ausubel (1990), Bernardo and Judd (1995), and Bernardo (1994) have solved models of asymmetric information much more general than the special exponential-Gaussian example.

There has also been success in developing algorithms for computing Nash equilibria of games. Lemke and Howson (1964) computed Nash equilibria of two-person games, and Wilson (1971) extended this to general n -person games. Wilson (1992) also developed an algorithm to compute stable equilibria. Despite the large body of work on this topic, there have been few applications of these methods to specific problems. Dynamic games with linear-quadratic structures are easily solved, but outside of few exceptions (Judd 1985; Reynolds 1987; Fershtman and Kamien 1987), this approach is almost never taken in theoretical industrial organization. In contrast, these methods are used in the international policy competition literature and monetary policy games, as in Canzoneri and Henderson (1991). More recently Maguire and Pakes (1994) applied numerical methods to dynamic games of entry and exit, and Miranda and Rui (1997) applied orthogonal polynomial approximation methods for computing Nash equilibria of general nonlinear dynamic games. Judd and Conklin (1995, 1996) have developed methods for finding all subgame perfect equilibria in infinite-horizon games, including problems with state variables.

There has also been an increasing use of Monte Carlo simulation schemes to study dynamic economic systems with nontraditional models of behavior. Standard dynamic general equilibrium analysis makes strong assumptions concerning agents' ability to analyze information and make forecasts. Some have argued that real people are far less capable than the rational agents that populate these models. Some of this work is called *agent-based computational economics, artificial life, or evolutionary models*. This work focuses on modeling the decision rules used by economic agents and investigates the aggregate implications of these rules. The Santa Fe web page on artificial life at <http://alife.santafe.edu/> and Leigh Tesfatsion's web page on agent-based economics at <http://www.econ.iastate.edu/tesfatsi/abe.htm> are excellent resources.

This quick, and very limited, review shows that we now have numerical methods for solving a wide variety of basic problems. In fact it is difficult to think of a problem in economic theory for which there does not now exist a reasonable algorithm to use. However, after reviewing the array of available methods, I am disappointed with the limited role any of them, even the well-known "golden oldies," plays in economic analysis. I suspect one reason is that many economists are unaware of these methods. This book endeavors to make the basic methods of numerical analysis and applied

mathematics accessible to the typical economist and help him to exploit them in economic analyses.

1.2 Roles of Computation in Economic Analysis

Advances in computation will affect economic analysis in two ways. Of course, standard applications will be made more efficient. More interesting, however, are the possible novel roles computation may have in economic analysis and where it fits in methodologically. The power of modern computing makes it possible to analyze both the qualitative and quantitative dimensions of models far more complex and realistic than those usually employed in economic theory. In this section we discuss the roles computation can play in economic analysis.

Quantitative versus Qualitative Analysis

An important weakness of standard theoretical analysis is that the results are only qualitative. Consider general equilibrium theory, arguably the best example of an economic theory. Existence of equilibrium is an important question, but existence does not tell us much about equilibrium. Efficiency results are also important, but they usually come at the cost of many unrealistic assumptions such as zero transaction costs and perfect information. When we relax some assumptions, we may end up with equilibrium being generically inefficient, but the stark judgment—efficient or inefficient—is of little practical value. It may be that equilibrium is inefficient, but if the amount of inefficiency is small or if there are no practical corrective policies, then equilibrium is essentially efficient.

Computation can help identify what is important and what is not. Theory can identify qualitative features of an equilibrium but cannot easily indicate the quantitative significance of any feature. Since theory is so general, its results do not always have much quantitative significance. For example, general equilibrium theory assumes arbitrary concave utility functions, permitting any mixture of curvature and substitutability consistent with concavity. Only when we make some specific assumptions motivated by observations and compute their implications can we reach quantitatively substantive conclusions.

However, we often need only a little empirical information in order to find strong results. For example, consider the theoretical monetary analysis in Fischer (1979). He investigated a Sidrauski model of money demand and rational expectations and asked whether it displayed the Tobin effect, that is, whether inflation affected growth by encouraging agents to save through real investment instead of monetary assets. He showed that a Tobin effect existed in that model. However, Balcer and Judd (1985)

computed the Tobin effect in his model for a large range of empirically reasonable values for the critical parameters in a generalization of Fischer's model and found that increasing annual inflation from zero to one hundred percent would increase net investment by at most *one-tenth of a percent*. An effect that small would not seem worth studying and would be undetectable in the data. Furthermore, if we did find a significant relation between inflation and growth in real-world data, these quantitative results tell us that the explanation does not lie in the elements of Fischer's model. Balcer and Judd did not use just a few parameter estimates; instead, their computations included any parameter values even remotely consistent with the data. Therefore the Fischer analysis does show that the Tobin effect is consistent with rational expectations but surely not in a quantitatively significant fashion in actual economies.

The theory literature is full of qualitative analyses that never consider the quantitative importance of their results. These analyses are valuable in providing basic insight and illustrating new concepts. However, many of these papers also claim real-world relevance, proclaim success when their model produces the desired *qualitative* correlation only, and totally ignore the question of whether the analysis is *quantitatively* plausible. Rather little of the theoretical literature is subjected to any such quantitative testing.

Theoretical analyses also often make simplifying assumptions that lead to non-robust conclusions. An excellent example of this occurs in the executive compensation literature. Jensen and Murphy (1990) found that the management of a typical corporation earns, at the margin, only three dollars per thousand dollars of firm profits. They argued that this was too small to create the proper incentives for managers and that this low level of performance incentives could not be explained by managerial risk aversion. This view seems reasonable initially, since we know that marginal compensation should be one dollar per dollar of profits if managers were risk neutral and should be zero if managers were infinitely risk averse. Since managers are more likely to be closer to zero risk aversion than infinite risk aversion, one might guess that managerial incentives should exceed three dollars per thousand. Risk neutrality assumptions are common in the incentives literature because they eliminate the complexities that come with nonlinear utility functions and are presumed to be reasonable approximations to finitely risk averse managers. In response, Haubrich (1994) actually computed some optimal contracts with plausible estimates of risk aversion and showed that optimal contracts would give managers small marginal incentives; he showed that for many reasonable cases, the marginal incentive would be in fact *three dollars per thousand!*

Conventional theoretical analysis is very good at telling us what is possible and at exploring the qualitative nature of various economic phenomena. However, only

after we add quantitative elements to a theory can we determine the actual importance of an analysis. That step requires computation.

Deductive versus Computational Analyses of A Theory

Conventional theory can answer many important questions, but generally it cannot answer *all* important questions. Computation can pick up where the usual theoretical approaches end. To see how that can be, we must have a broad view of what constitutes theoretical analysis.

A theory is a collection of definitions and assumptions. After specifying a theory, one wants to determine its implications. In economics this is typically done by proving theorems; call this *deductive theory*. Some questions are best answered with theorems. Existence theory and efficiency analysis are important because they teach us how to approach many critical issues. Only deductive methods can prove the existence of equilibrium. More generally, only deductive methods can determine the qualitative structure of a theory. For example, deductive methods tell us if equilibrium depends on parameters in a continuous fashion, or if an agent's equilibrium decision rule is increasing, continuous, and/or concave in the agent's state variable.

Some questions can be addressed in general, but many other questions are not. We are often unable to prove general results and instead turn to tractable special cases. These special cases are just examples of the general theory and are likely unrepresentative of the general theory. Their appeal often lies more in their tractability and in the theorems that follow than in their reasonableness. It is at this point that computation has much to offer, for it can provide approximate solutions to a much broader range of examples of a general theory than can special cases with tractable solutions.

One problem with computation is that the solutions are approximate, whereas theorems are exactly true. Many approach economic theory in the same way we approach a mathematical theory. However, economics and mathematics have very different logical architectures. Pure mathematics is a cumulative activity where the result of one theorem is used in proving many others. The path from definitions and assumptions to final proof is very long for most of mathematics. When the structure of a theory is so deep, it is imperative that the foundations and intermediate development be completely justified. It is understandable why theorem-proving is and will remain the dominant mode of analysis in mathematics.

This is not the case in economics. All economic theories consist of a shallow layer of economic modeling lying on top of a deep mathematical foundation. The usual proof of an economic theorem relies little on previous economic theorems and draws instead on mathematical theorems.

It is therefore natural to use a computational approach to study an economic theory. By this I mean describing a theory in terms of its definitions and assumptions, focus on a representative set of examples, solve them computationally, and look for patterns in the computational results. A computational approach will have to focus on computationally tractable cases, but those cases will be far more general than cases that are analytically tractable and amenable to theorem-proving. In the past this approach would not have been too useful because one could only solve a few instances of a theory, far too few to rely on. Current technologies allow us to compute a thousand times as many instances as we could twenty years ago, and we will be able to increase that by another factor of a thousand in the near future. While one may not want to rely on apparent patterns in 10 examples, it is harder to ignore a robust pattern across 10,000 examples, and difficult to ignore 10,000,000 confirming cases of a hypothesis.

While most agree that theoretical models are generally too simple, they may feel that computational analysis is not a good alternative. Many argue that the results of a computational study are unintuitive and incomprehensible because the computer program that generates the result is essentially an impenetrable black box.

The black box criticism should give us pause, but it is not damning. First, this is often more a comment on the poor fashion most computational work is exposed. When a computation gives an answer, we do want to know which economic forces and considerations determined the answer. One way to address this demand is to conduct several alternative computations. Many numerical papers offer little in the way of sensitivity analysis.

Second, we need to recall Einstein's recommendation—a model should be “as simple as possible, but not simpler.” Economists study complex questions, a fact that is true if we are macroeconomists or tax economists studying national economies and their policies, microeconomists studying firms, or labor economists studying decision-making in a family. This consideration is often ignored in applied economic theory where unicausal analyses dominate. For example, the industrial organization literature is filled¹ with models that explore one interesting feature of economic interaction in isolation. A typical model may study moral hazard *or* adverse selection, *or* entry *or* investment *or* learning *or* R&D, *or* asymmetric information about cost *or* demand, *or* sharing information about cost, *or* sharing information about demand, and so on. To see how limiting this is, suppose that meteorologists took this approach to studying the weather; they would ignore complex models and their “black box” computer implementations and instead study evaporation, *or* convection, *or* solar heating, *or*

1. See, for example, Judd (1985).

the effects of the earth's rotation. Both the weather and the economy are phenomena greater than the sum of their parts, and any analysis that does not recognize that is inviting failure. Simple, focused theoretical studies give us much insight, but they can only serve as a step in any substantive analysis, not the final statement.

Necessity of Approximation

When we face up to the complexities of economic analysis, we see that the issue is not *whether* we use approximations but *where* in our analysis we make those approximations, what kind of approximation errors we tolerate, and how we interpret the inevitable approximation errors. Simple, unicausal models make approximation errors by ignoring all but one feature of the real world. They are at best instructive parables. While we may be able to prove theorems about those one-dimensional models, the results have only approximate, if any, validity concerning the real world. Computational methods can be much more realistic, but they bring with them approximation errors of a numerical kind. We are generally presented with a trade-off: Achieve logical purity while sacrificing realism and inviting specification error, or bring many elements of reality to the analysis and accept numerical error. The proper choice will depend on the context. As computational costs fall and algorithms improve, computational methods will more often be that choice.

Partners in Analysis

Our purpose in these last sections was to argue that neither the purely deductive nor the purely computational approach to analyzing a theory is adequate. Both have much to offer, and both have weaknesses. We next discuss some of the synergistic ways in which computational methods and conventional economic theory can interact.

The goal of deductive theory in economics is to describe the nature of economic interaction. A complete characterization of even simple theories is often impossible using deductive methods. However, deductive analyses can often provide a partial characterization, which can then be used by computational methods to produce efficient numerical approaches which yield economically substantive results.

A particularly good example of this kind of partnership is in the literature on dynamic contracts. Spear and Srivastava (1987) studied a general model of dynamic principal-agent contracting under repeated moral hazard. Initially the problem appears to be infinite-dimensional, intractable from both a theoretical and computational view. Spear and Srivastava came up with an insight that reduced the problem to a one-dimensional dynamic programming problem. This reduction did not make pure theory much easier, but it drastically simplified the computational problem.

Later Phelan and Townsend (1991) computed examples that illustrated many properties of those contracts.

Deductive theory can be very useful in reducing extremely complex problems to equivalent problems with a much simpler structure. When we combine these results with other qualitative properties established by deductive theory, such as differentiability and monotonicity, we can develop efficient computational approaches. These technical complementarities will become increasingly important as economists examine ever more complex economic models.

Theory without Theorems

In some cases there may be no comprehensible theorem. A problem may have a very complicated pattern of results that defies summarization in a tidy theorem. What are we to do then? The following is a good example of what is probably not an unusual situation and how a computational approach to analyzing an economic theory can succeed where deductive theory will fail.

Quirmbach (1993) asked a basic and important question in the economics of innovation and antitrust policy. Suppose that several firms pursue a research and development project that has a probability of success equal to τ independent across firms. The successful firms then all produce the new product (patenting is presumed unavailable). The issue is how the market structure and conduct of the output market affects the ex ante R&D effort and net expected social welfare. Some argue that excessive ex post competition will reduce profits among the successful innovators, and discourage ex ante R&D effort. This suggests that antitrust policy should be lax when it comes to high-tech industries. Quirmbach asked what form of ex post oligopolistic interaction and regulation would lead to the greatest social welfare.

In the typical paper one would make enough specific assumptions for the demand function, the cost function, and the specification of imperfect competition in order to prove theorems. Few attempts are made to generalize the results for general tastes, technology, or mode of competition. Instead of attempting to prove a theorem ranking the ex post market structures, Quirmbach computed the social welfare for hundreds of examples. Figure 1.1 displays a typical plot of social welfare W against τ found by Quirmbach, which illustrates many critical facts. First, Quirmbach found no “theorems,” if by “theorem” we mean a precise, compact, and understandable statement summarizing the results. Each market structure dominates the others at some parameters. It is not even possible to describe the dependence on τ , since Bertrand and Cournot ranks switch due to discontinuities in the Bertrand performance. Any “theorem” that tries to summarize Quirmbach’s results would be long, twisted, and incomprehensible.

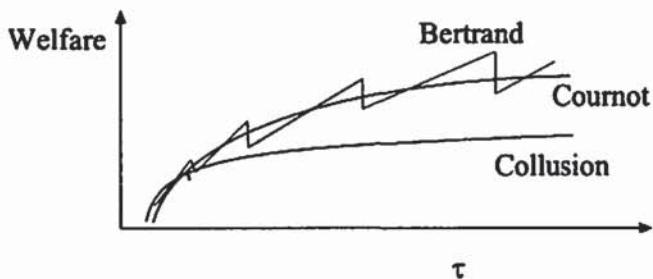


Figure 1.1
Quirmbach's results for social welfare and market structure

Second, despite the absence of simple theorems, Quirmbach's computations produced important and robust findings. Even though he could not rank the market structures absolutely, perfect collusion is usually much worse. When collusion does outperform Bertrand and Cournot, the difference is insignificant. Even though no simple theorem can summarize these facts, Quirmbach's computations contain much economic content and clearly reject the argument that collusion should be tolerated because of innovation considerations.

I suspect that robust analyses of many important questions in economics would resemble the situation in figure 1.1. The normal approach of finding a model simple enough to come up with clean results can easily lead us away from the real truth. Using efficient computational methods to produce solutions to numerous cases can avoid these problems.

An Economic Theory of Computational Economic Theory

Even if there are some theorems, it is unclear if they dominate the kind of computationally intense analysis displayed in Quirmbach. The most controversial use of computers in economic theory would be the use of computations instead of proofs to establish general propositions. One example is not a proof of a proposition; neither do a million examples constitute a proof of any proposition. While the latter is far more convincing than one example, a theorem about a continuum of special tractable cases does not prove anything general. Furthermore, what is the marginal value of a proof once we have a million confirming examples? In some cases that marginal value is small, and may not be worth the effort. In fact in some cases computation alone can provide a proof.

As economists, we believe that the allocation of scarce resources in economic research will follow the laws of economics. The objective of economic science is understanding economic systems. Theories and their models will continue to be used to summarize our understanding of such systems and to form the basis of empirical

studies. We have argued that the implications of these theories can be analyzed by deductive theorem-proving, or they can be determined by intensive computations. The inputs of these activities include the time of individuals of various skills and the use of computers, either as word processors or number crunchers. Theorem-proving intensively uses the time of highly trained and skilled individuals, a resource in short supply, whereas computation uses varying amounts of time of individuals of various skill levels plus computer time.

Decision-makers sometimes use economic research. Many of these end-users care little about the particular mode of analysis. If a million instances covering the space of reasonably parameterized models of a smooth theory follow a pattern, most decision-makers will act on that information and not wait for the decisive theorem. In the absence of a proof, most will agree that the computational examples are better than having nothing. Most end-users will agree that the patterns produced by such computations are likely to represent general truths and tendencies, and so form a reasonable guide until a conclusive theorem comes along.

The picture drawn here is one where the alternative technologies of deductive analysis and intensive computation can produce similar services for many consumers. Economic theory tells us what will likely happen in such a circumstance. In the recent past the theorem-proving mode of theoretical analysis was the only efficient method, but now the cost of computation is dropping rapidly relative to the human cost of theorem-proving. I anticipate that by the end of the next decade, it will be typical for an individual to outline a theory to a computer which, after a modest delay, will deliver the results of a computationally intensive analysis.

The clear implication of economic theory is that the computational modes of theoretical analysis will become more common. In some cases computational modes will dominate theorem-proving. In other cases increased computing power will increase the value of theoretical analysis. In either outcome it is increasingly valuable for economists to become familiar with computational methods. This text aims to help economists in that task.²

1.3 Computation in Science

Computer solutions of complex problems in physics give us examples of how numerical methods can make substantial contributions to scientific analysis. One of

2. This section owes much to and freely borrows from a George Stigler talk on the mathematization of economics. While less dramatic, the computerization of economics may be similar in terms of how it affects the style, emphasis, and allocation of effort in economic research.

my favorite examples is the work on Jupiter's Red Spot. Astronomers generally believed that there must be some peculiar feature of Jupiter causing this long-lived hurricane in the Jovian atmosphere. A computer simulation showed that the Red Spot is really not unexpected once we analyze a sufficiently sophisticated model of Jupiter's atmosphere. Assuming only the basic gravitational properties of Jupiter and the thermal and fluid properties of its atmosphere, computer simulations showed that such disturbances occur naturally and are stable on Jupiter even though they are not stable on Earth. Therefore the Red Spot could be explained from basic principles, a conclusion possible only through intensive computation.

Closer to home is the question of why Earth has a magnetic field, a problem that Einstein called one of the five most important questions in science. Numerical work predicted that the magnetic field is caused by Earth's molten iron core rotating faster than the rest of the planet by about one day per 400. This prediction was recently borne out by seismic evidence.

Astronomical and meterological examples of computational modeling are appropriate for economists. Since astronomy, meterology, and economics are all primarily observational sciences,³ they all have to take what they observe and infer the causes without the benefit of controlled experiments. Computer modeling operates as a substitute for experimentation in screening possible explanations for plausibility and consistency.

These examples are academic problems where computation was successful. However, computation is important for real problems and limits on computational capabilities can have real consequences. Consider, for example, the development of nuclear weaponry. Both Allied and German scientists worked hard to design fission devices at the beginning of the Second World War. A key scientific problem was estimating critical mass, the amount of U-235 necessary for a self-sustaining reaction. All agreed that the question could be analyzed using the theory of quantum mechanics. The challenge they faced was computing solutions to the key equations. They were severely limited by their computational "equipment"—primarily fingers, pencil, paper, and adding machines. This forced them to make extensive approximations in their analysis to reduce the mathematics to something simple enough for their computational capabilities. Historians have often wondered why the German bomb effort failed, only recently discovering that Heisenberg and his colleagues failed in their critical mass calculations. They had used a poor approximation strategy,

3. The parallel is not exact. The nonexperimental nature of most economics research is a political fact, not a necessary feature of economic science. If economic research were given a budget equal to that given to the search for the Higgs boson, top quark, and other exotic particles, economics could also be an experimental science.

estimated that any bomb would require tons of U-235, and concluded that it was not feasible for any combatant to build and deliver one. Allied scientists used better methods and estimated that 25 pounds would suffice. With this estimate in hand, the Manhattan Project was authorized. It turned out that 25 pounds was more than adequate. Computational technology continues to play a major role in nuclear matters. The testing of unclear weapons may soon be banned by international treaty. A necessary ingredient for any such agreement is that all nuclear powers be confident that their computer simulations can reliably predict the performance of nuclear devices, making testing unnecessary.

Note that these examples are all cases where scientists and engineers have an accepted theory—fluid dynamics, electrodynamics, quantum mechanics—for their problem, and they want to know what that theory implies. They do not aim for closed-form solutions of the underlying equations, nor do they focus on proving theorems. Instead, they used numerical methods to explore the implications of their theory. These examples illustrate not only the usefulness of computational strategies but also the importance of using state-of-the-art computational methods that allow one to examine the theory in a robust and realistic manner.

1.4 Future of Computing

Economists are discussing computation today more than in the past mostly because of the substantial advances in computational tools. Advances on the hardware and algorithm fronts indicate that we will be able to solve an increasingly large collection of economic problems. To appreciate the potential of computational methods, we need to be familiar with the basic reasons for these advances and their likely trends.

First, the raw speed of computer components is increasing exponentially. Progress in semiconductor technology has been roughly following *Moore's law*: a doubling of component density (and speed) every eighteen months for silicon chips. This feature of standard semiconductor technology will hit limits in the future, but not for another decade. Furthermore there is no reason to think that we are close to fundamental physical limits for computation (see Bennett and Landauer 1985). Even when we have reached the limit with silicon, there are many other technologies that may allow us to continue this pace. Beyond that, quantum computing promises to solve in minutes problems that would take current computers millions of years to do.

Figure 1.2 displays the recent progress in computer technology. We measure raw computational speed in terms of floating point operations per second (flops). There is substantial variation in computational power across computers, but the basic trends are clear. The first computers of the 1940s could perform a few thousand flops. The

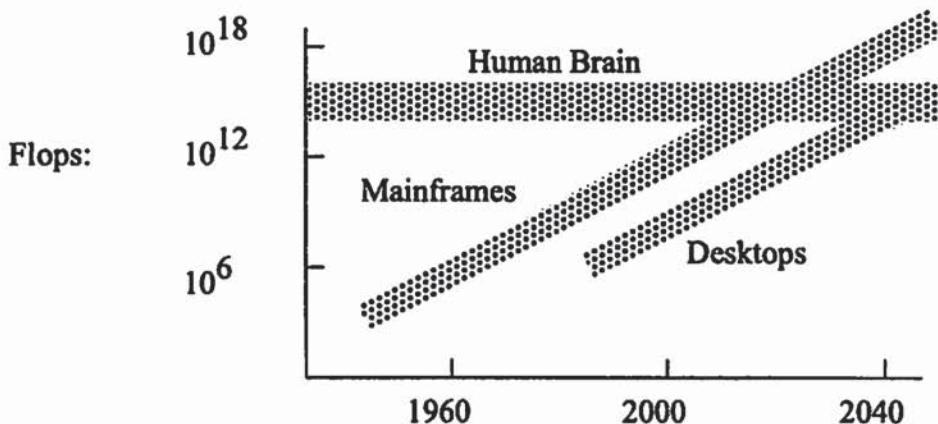


Figure 1.2
Trends in computing speeds

leading supercomputer today has *teraflop* power (one trillion flops). Over this period the computational power of the leading machines (denoted as “mainframes” in figure 1.2) has increased (roughly) by a factor of 1,000 every twenty years. Smaller and cheaper machines (denoted “desktops” in figure 1.2) were introduced in the 1970s and lag the leading computers by about twenty years. For example, today’s desktops are more powerful than the first supercomputers of the 1970s.

By comparison, the human brain with its “wet” hardware is a far more powerful computer, with estimated performance at up to a quadrillion (10^{15}) flops—a *petaflop* machine. It is not surprising that it can perform many tasks, such as hand-eye coordination, which computers based on “dry” technologies cannot currently do well. Unfortunately, there is no discernible positive trend in the capabilities of the human brain.

Figure 1.2 shows that at current trends the raw power of many computers will reach that of the human brain in a few decades. Extrapolation of trends is risky, but most experts agree that current trends will continue for at least twenty years. Furthermore the most important index is the real cost of computing power. Economies of scale in production combined with the economies of scale promised by parallel processing will improve this index even in the absence of hardware progress. Barring many unexpected technical problems, affordable petaflop computers will be here in the next few decades.

Second, there also is progress in algorithm development. In fact advances in numerical analysis have improved algorithm speed as much as hardware advances for many important problems. Rice (1983) discusses this for algorithms solving two- and three-dimensional elliptic partial differential equations, a class of numerical problems that arise naturally in continuous-time stochastic economic modeling. He argues that we were able to solve these problems 4 million to 50 billion times faster in 1978 than

in 1945, of which a factor of 2,000 to 25 million can be attributed to software improvements, as much if not more than can be attributed to hardware improvements.

Even if we have petaflop machines, it may be difficult to write programs that harness that power. On this front, genetic algorithms and genetic programming propose methods for describing a problem to the computer and then letting a computer find programs for solving it.

There are clear synergies in this process. When faster computers become available, algorithmic research focuses on developing methods that become useful only when applied to large problems. New hardware architecture, such as parallel processing, stimulates algorithmic research that can exploit it. Furthermore, as one uses better algorithms on faster machines, it becomes easier to design even faster machines and even better algorithms. Even if the genetic programming approach is costly, writing a program is a fixed cost that is easier to justify the more that program is used.

Progress in speed combined with algorithmic advances will allow us to accurately solve increasingly complex problems. This means that we will analyze larger data sets with more powerful estimation techniques and solve larger CGE models. It also will allow economists to change how they approach economic analysis in the directions discussed in section 1.2.

1.5 Objectives and Nature of This Book

I now make clearer what this book is and what it is not. The book is primarily an attempt to introduce economists to numerical methods and other computational procedures that are of proven or potential value in solving economic problems. The choice of topics and the depth of their development reflects this focus.

This book is *not* a formal, theorem-proving presentation of numerical analysis. The objective is to familiarize the reader with the basic algorithms, which algorithms are suitable for which problems, and provide intuition and experience to guide the readers in their applications of numerical methods to economic problems. References are provided for the readers who want to become more familiar with the formal development of the subject. This text is mostly a user's manual for those who want to use methods from numerical analysis and applied mathematics.

Another reason why this book presents few theorems is that numerical analysis is as much an art as a precise science. The so-called theorems in numerical analysis have limited direct applicability, and their assumptions are often impossible to verify. For example, there are many theorems that state that an algorithm converges *if one begins with a good initial guess*, leaving one with the problem of finding good initial guesses. We will also often want to use algorithms for which we have no good theory.

The “truths” of numerical analysis are not black and white but come in many shades of gray. One can appreciate this only after direct experience with a wide variety of problems and algorithms.

This book differs substantially from standard numerical analysis texts in its choice of material and in the detail with which it is developed. For example, standard numerical analysis texts give far greater detail on numerical linear algebra algorithms than does chapter 3. My objective is to have the reader become familiar with the basic problems and concepts of numerical linear algebra, but since there are many good numerical packages available, there is no need for the reader to write the code to solve linear algebra problems. In fact the reader is encouraged to use standard packages to solve numerical problems.

In contrast, integration theory in chapters 7, 8, and 9 is developed well beyond what is typical in introductory numerical analysis texts. Many economics problems involve multidimensional integration, leading us to include advanced methods for multidimensional integration. Since there do not exist efficient general multidimensional integration algorithms, the economic analyst will generally need to choose the method appropriate for the particular problem and write an integration code using the most appropriate techniques. Given the importance of integration in many economic models, economists will often develop their own, specialized integration methods (this is already the case in Bayesian econometrics); such work requires more than a shallow grasp of numerical integration. Chapter 9 on quasi-Monte Carlo methods may appear particularly advanced, obscure, and excessively mathematical; however, its topic is one of the few numerical methods ever discussed in the *Wall Street Journal*.

In between the chapters on linear equations and integration are discussed the basics of optimization, nonlinear equations, and approximation theory. Much of the optimization material will be familiar, but the book covers optimization more completely than is usual in an econometrics course. Chapter 5 discusses the range of methods available to solve nonlinear equations, laying the foundation not only for solving Arrow-Debreu general equilibrium models but also for solving a variety of dynamic, stochastic models. Chapter 6 on approximation theory discusses techniques for approximating functions ranging from regression-style methods familiar to economists to multidimensional neural networks.

While these methods are all very powerful, few problems in economics can be solved solely through an application of nonlinear equations, optimization, or approximation theory. This is obvious once we think about the typical economics problem—individual agents maximize utility (an optimization problem), supply equals demand (a nonlinear equation problem), individuals follow a decision rule (an approximation problem), and agents form expectations (an integration problem).

Combining these methods to solve interesting economic problems appears at first to be daunting, but the numerical functional analysis approach called projection methods (a.k.a. method of weighted residuals, finite element, Galerkin, etc.) provides us with a framework in which we can solve a vast range of economic problems; projection methods are introduced in chapter 11, and applied frequently thereafter.

This book is *not* a comprehensive description of computational practice in economics. Many “innovations” in computational economics are straightforward applications of standard numerical methods, inferior reinventions of earlier work, or ad hoc procedures with little mathematical foundation. This book instead endeavors to show how *formal*, *standard*, and *proven* concepts and methods from numerical analysis are or *can be* used in economics. If a technique has been used in the economics literature, it may be used as an example. If the economics literature uses a different ad hoc procedure, we may compare it with the standard method from numerical analysis, particularly if this is necessary to avoid confusion.

Some economists have made substantial contributions to the development of efficient numerical procedures. I suspect that in the future economists will make contributions in areas of special interest to economists, just as econometricians have made contributions to statistical practice. However, we must first learn the basics of numerical and computational mathematics. Much of what is presented in this book has never been used before in economics. This is not a backward-looking textbook describing the numerical methods that have been used in economics, nor an intellectual history of computation in economics, but instead the book attempts to focus on methods that will be of value.

This book is idiosyncratic, reflecting very much the author’s experience, interests, training, and point of view. I do not attempt to present every contribution to computational economics. Instead, my priority is to give a unified treatment of important topics, and base it on standard mathematical practice.

This book is not intended to be a statement of what is and is not “computational economics.” There are many computational topics that are not included here because of lack of space. Accordingly the title of this book is not *Computational Economics*.

The book minimizes overlap with the highly developed areas of computational economics. Little space is given to CGE, except as an example of nonlinear equation solving. This was done for two reasons. First, I do not want to spend too much effort on a specific application of computation, and second, excellent books are available that extensively cover this topic. A course that emphasizes CGE could easily supplement this book with a text on CGE methods. This book instead introduces a wide variety of applications although not all the possible interesting applications. The problems at the end of each chapter and in the book’s web site introduce other applications.

The other factor determining the book's coverage is the aim that it serve as a text for a second-year graduate course in numerical methods in economics. This book contains enough material for a year-long course. I know of no graduate program that offers such a course. An important objective of this text is thus to show what such a course might look like.

For all these reasons the coverage of recent work is very limited. For example, I do not discuss the artificial life, agent-based computational economics, and similar learning and evolution literature. However, I do describe some of the basic tools used in those literatures such as neural networks and genetic algorithms. Furthermore the Monte Carlo and quasi-Monte Carlo chapters are relevant to anyone who uses simulation methods.

This book presents the basic tools of numerical analysis and related computational methods; how they are put together to analyze economic issues is up to the user. It can serve only as an introduction; hence this book also aims to be a guide to the numerical literature for those who want to learn about methods particularly useful for economic analysis. Advanced topics and further developments are quickly discussed at the end of each chapter to help the reader continue his study.

1.6 Basic Mathematics, Notation, and Terminology

There is a large amount of notation and terminology in this book. At this point, along with a list of some basic notation, I want to discuss the approach I take to terminology, and then review some basic mathematical results that arise frequently.

Scientific Notation

This book will use a modification of scientific notation to represent many numbers. In general, $a(m)$ represents $a \times 10^m$. This is done to eliminate needless zeros. For example, 0.0000012 becomes 1.2(−6).

Vectors and Matrices

Unless indicated otherwise, a vector $x \in R^n$ will be a column vector,

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

The *transpose* of x is the row vector (x_1, x_2, \dots, x_n) and is denoted by x^\top . The symbol x_i will denote component i of the vector x . Superscripts will generally denote distinct

vectors; for example, x^l is the l th vector in a sequence of vectors, x^1, x^2, \dots . The special vector $e^j \in R^n$ will denote the vector where all components are zero except for component j which is unity; that is, e^j is the unit vector in dimension j . There will also be other special vectors and matrices: 0_n and $0_{n \times n}$ denote the vector and matrix of zeros, $1_n \in R^n$ is the vector of ones, and I_n is the $n \times n$ identity matrix.

If $f: R^n \rightarrow R$ is a scalar-valued function of R^n , its *gradient* is the row vector function $\nabla f: R^n \rightarrow R^n$ defined by

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \dots, \frac{\partial f}{\partial x_n}(x) \right)$$

and is denoted f_x . The *Hessian* of f is the matrix-valued function $f_{xx}: R^n \rightarrow R^{n \times n}$ defined by

$$f_{xx}(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_2 \partial x_1}(x) & \dots & \frac{\partial^2 f}{\partial x_n \partial x_1}(x) \\ \frac{\partial^2 f}{\partial x_1 \partial x_2}(x) & \frac{\partial^2 f}{\partial x_2 \partial x_2}(x) & \dots & \frac{\partial^2 f}{\partial x_n \partial x_2}(x) \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) & \frac{\partial^2 f}{\partial x_2 \partial x_n}(x) & \dots & \frac{\partial^2 f}{\partial x_n \partial x_n}(x) \end{pmatrix}.$$

The system of m functions in n variables $f: R^n \rightarrow R^m$ is a column vector of scalar functions,

$$\begin{pmatrix} f^1 \\ f^2 \\ \vdots \\ f^m \end{pmatrix},$$

and its Jacobian is the matrix-valued function $f_x: R^n \rightarrow R^{m \times n}$,

$$f_x(x) = \begin{pmatrix} \frac{\partial f^1}{\partial x^1}(x) & \frac{\partial f^1}{\partial x^2}(x) & \dots & \frac{\partial f^1}{\partial x^n}(x) \\ \frac{\partial f^2}{\partial x^1}(x) & \frac{\partial f^2}{\partial x^2}(x) & \dots & \frac{\partial f^2}{\partial x^n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f^m}{\partial x^1}(x) & \frac{\partial f^m}{\partial x^2}(x) & \dots & \frac{\partial f^m}{\partial x^n}(x) \end{pmatrix}.$$

Table 1.1
Mathematical notation

R	Real numbers
R^n	n -dimensional Euclidean space
$ x $	Absolute value of x for $x \in R$
$\lceil x \rceil$	Smallest integer greater than or equal to $x \in R$, or the ceiling of x
$\lfloor x \rfloor$	Largest integer smaller than or equal to $x \in R$, or the floor of x
$\ x\ $	Euclidean norm of $x \in R^n$
∂D	Boundary of the set $D \subset R^n$
$X \times Y$	Tensor product of the sets X and Y : $\{(x, y) x \in X, y \in Y\}$
X^k	k -wise tensor product of the set X
$C^k(X)$	Space of k -times differentiable functions f , $f : X \subset R^n \rightarrow R$; we often drop X when it is clear from context
$L^p(X)$	Space of functions $f : X \rightarrow R$ such that $\int_X f ^p d\mu < \infty$
$L^2(X)$	Space of square integrable functions $f : X \rightarrow R$
$L^\infty(X)$	Space of essentially bounded integrable functions $f : X \rightarrow R$
$\ f\ _p$	L^p norm of $f \in L^p(X)$
$N(\mu, \Sigma)$	Normal distribution with mean μ and variance-covariance matrix Σ
$U[a, b]$	Uniform distribution on $[a, b]$

Algorithms

Most chapters contain descriptions of algorithms. I use an informal style of presentation which is hopefully self-explanatory. One could write programs based on these algorithms. However, these should be treated as expositional devices since far better implementations are available from software archives, including that of this book.

Notation

Table 1.1 lists other standard notation.

Linear Algebra

Let A be a square matrix, $A \in R^{n \times n}$. Many of the important properties of a matrix are summarized by its eigenvalues and eigenvectors.

DEFINITION Let C denote the complex plane. $\lambda \in C$ is an *eigenvalue* of A if and only if there is some $x \in C^n$, $x \neq 0$, such that $Ax = \lambda x$. Let $\sigma(A)$ denote the set of eigenvalues of A ; we refer to it as the *spectrum* of A . If $Ax = \lambda x$ for a nonzero x , then x is an *eigenvector* corresponding to $\lambda \in \sigma(A)$. The *spectral radius* of A is $\rho(A) \equiv \max\{|\lambda| : \lambda \in \sigma(A)\}$.

If x is an eigenvector, then so is any scalar multiple of x ; hence eigenvectors are defined only up to a proportionality constant. Also $\lambda \in \sigma(A)$ if and only if $\det(A - \lambda I) = 0$, the *characteristic equation* of A , implying that $\lambda \in \sigma(A)$ if and only if $A - \lambda I$ is singular. It is often useful to decompose a matrix into the product of other matrices. A particularly useful decomposition is the *Jordan decomposition*. Suppose that A is a $n \times n$ nonsingular square matrix and has n distinct eigenvalues. Then $A = NDN^{-1}$ where D is a diagonal matrix with the distinct eigenvalues on the diagonal, and the columns of N are right eigenvectors of A . Equivalently we can express $A = N^{-1}DN$, where the diagonal elements of D are the left eigenvalues of A and the rows of N are left eigenvectors of A .

Order of Convergence Notation

Our convergence concepts are compactly expressed in the “Oh” notation. We say that $f: R^k \rightarrow R^l$ is $\mathcal{O}(\|x\|^n)$ if $\lim_{x \rightarrow 0} \|f(x)\|/\|x\|^n < \infty$, and that $f: R^k \rightarrow R^l$ is $o(\|x\|^n)$ if $\lim_{x \rightarrow 0} \|f(x)\|/\|x\|^n = 0$.

Taylor's Theorem

The most frequently used theorem in numerical analysis is Taylor's theorem. Given its central role, we first state the R version:

THEOREM 1.6.1 (Taylor's theorem for R) If $f \in C^{n+1}[a, b]$ and $x, x_0 \in [a, b]$, then

$$\begin{aligned} f(x) &= f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2}f''(x_0) \\ &\quad + \cdots + \frac{(x - x_0)^n}{n!}f^{(n)}(x_0) + R_{n+1}(x), \end{aligned} \tag{1.6.1}$$

where

$$\begin{aligned} R_{n+1}(x) &= \frac{1}{n!} \int_{x_0}^x (x - t)^n f^{(n+1)}(t) dt \\ &= \frac{(x - x_0)^{(n+1)}}{(n+1)!} f^{(n+1)}(\xi) \end{aligned}$$

for some ξ between x and x_0 .

Taylor's theorem essentially says that one can use derivative information at a single point to construct a polynomial approximation of a function at a point. We next state the R^n version:

THEOREM 1.6.2 (Taylor's theorem for R^n) Suppose that $f: R^n \rightarrow R$ and is C^{k+1} . Then for $x^0 \in R^n$,

$$\begin{aligned} f(x) &= f(x^0) + \sum_{i=1}^n \frac{\partial f}{\partial x_i}(x^0)(x_i - x_i^0) \\ &\quad + \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \frac{\partial^2 f}{\partial x_i \partial x_j}(x^0)(x_i - x_i^0)(x_j - x_j^0) \\ &\quad \vdots \\ &\quad + \frac{1}{k!} \sum_{i_1=1}^n \cdots \sum_{i_k=1}^n \frac{\partial^k f}{\partial x_{i_1} \cdots \partial x_{i_k}}(x^0)(x_{i_1} - x_{i_1}^0) \cdots (x_{i_k} - x_{i_k}^0) \\ &\quad + \mathcal{O}(\|x - x^0\|^{k+1}). \end{aligned} \tag{1.6.2}$$

Functions and Functional Operators

A function involving Euclidean spaces is denoted $f: R^n \rightarrow R^m$. We will often want to express the distinction between an argument of a function and a parameter. The notation $f(\cdot; a): R^n \rightarrow R^m$ denotes a function with domain R^n corresponding to a specific value for the parameter vector $a \in R^l$. If both $x \in R^n$ and $a \in R^l$ are viewed as variables, then we can also view $f(x; a)$ as $f: R^n \times R^l \rightarrow R^m$.

We will work with functions on function spaces. For example, suppose that X and Y are both spaces of functions from $Z \subset R^n$ to $W \subset R^m$. Then $\mathcal{F}: X \rightarrow Y$ will denote a function that maps functions in X to functions in Y . Sometimes we will write $g = \mathcal{F}(f)$ to state that g is the image of f under \mathcal{F} . Sometimes we will write the expression $g(z) = \mathcal{F}(f)(z)$ which says that when we evaluate the function $\mathcal{F}(f) \in Y$ at $z \in Z$ the image is $g(z)$. Of course $g = \mathcal{F}(f)$ if and only if $g(z) = \mathcal{F}(f)(z)$ for all $z \in Z$. Note that the slightly different expression $\mathcal{F}(f(z))$ is not defined because $f(z) \in W \subset R^m$, whereas the domain of \mathcal{F} is X , a space of functions.

One of the more famous functionals is the *Laplace transform*. In general, if $f(t): R \rightarrow R$, then the Laplace transform of $f(t)$ is $L\{f\}: R \rightarrow R$, where $L\{f\}(s) \equiv \int_0^\infty e^{-st} f(t) dt$. The key property used in linear differential equations is that $L\{f'\}(s) = sL\{f\}(s) - f(0)$, transforming the derivative of f to algebraic operations on the Laplace transform of f .

Terminology

This book follows standard terminology from the numerical analysis literature. The larger problem facing anyone writing in this area is that there are many places in the

economic literature where idiosyncratic terms have been created and used. It is important that economists adopt terminology consistent with the numerical analysis literature; otherwise, an economist who wants to use the existing numerical literature will find it difficult to locate and understand desired material. Therefore the book often ignores terminology used in the economic literature and uses appropriate terms from the numerical analysis literature; where there is possible confusion, it provides the map between the terms that economists have used and the mathematical terms.

However, the book does use precise terminology to the extent reasonable. Some economists have used vague, overly broad terms to describe numerical procedures, making it difficult to distinguish varieties that are substantially different from a numerical perspective. Proper and precise terminology is essential to communicating the method used in a numerical study. The mathematical literature takes great pains to distinguish even slightly different methods. For example, Newton's method for optimization is different from the quasi-Newton *family*, which includes the Davidon-Fletcher-Powell and Broyden-Fletcher-Goldfarb-Shanno algorithms. In this book I similarly use and, where necessary, invent terminology describing various techniques used in economics. The terminological choices made here are governed by the desire for precision and consistency with the mathematical literature.

1.7 Software and Supplemental Material

This book focuses on the main ideas behind numerical methods. For that reason I do not spend time on discussing particular programs, for that would involve coding details and force me to choose some language. Given the variety of techniques discussed, the book would have to teach and use several languages.

Nevertheless, numerical mathematics without software is not useful. To address these needs, I have created an archive of supplemental material for this book. To find it, the reader should consult the author's web page at <http://www-hoover.stanford.edu/bios/judd.html>. This web site includes code in various languages that helps solve the problems at the end of each chapter and the examples in each chapter. This archive also includes a variety of additional material, including supplemental text, bibliography, examples, problems, and solutions. This collection of supplementary material will grow over time to respond to readers' questions and requests, and thus will provide material that was left out of the text. The reader should also consult the web site for up-to-date information on typos.

To gain any real expertise in numerical analysis, the reader must know some basic computing language and do the exercises. The reader will need to find a computer to replicate the examples and do the exercises in this book, and should get acquainted

with software that can implement basic linear algebra operations, solve systems of nonlinear equations, and solve constrained optimization problems. None of the exercises require large programs, so Fortran, Gauss, Matlab, C, or almost any other language can be used in completing them. Because of the availability of NAG, IMSL, LAPACK, MINPACK, NPSOL, MINOS, and other software, Fortran is still the dominant language for serious numerical analysis. It is possible that C will catch up in the future, but Fortran90 is more flexible than Fortran77 and has adopted many features of C, possibly slowing down the drift to C. Packages such as Gauss and Matlab are very flexible, excellent for solving linear problems, and most likely adequate for anything done in this book. However, they are slower than Fortran or C in executing nonlinear problems.

There is a substantial amount of high-quality commercial software. IMSL and NAG are the two most complete collections of software and available at any serious computing facility. The advantages of such libraries is that they have been extensively tested and improved over time. The disadvantage is their cost.

There are also many programs freely available and not included in IMSL or NAG. Of particular usefulness is the NETLIB facility run by Argonne Labs; consult its web page at <http://www.netlib.org/>. One very useful collection of advanced programs in netlib is called “toms,” a collection of algorithms on a wide variety of problems published originally in the ACM Transactions on Mathematical Software.

I use public domain software wherever possible to solve examples, since this will facilitate the dissemination of programs. The disadvantage is that such software is not as reliable, nor as sophisticated as the commercial packages. Any serious programming should use the most recent versions of IMSL or NAG or a similar collection.

All of the public domain programs I use are available at the book’s web site. Readers are free to take any or all programs. These programs are not copyrighted. As always, you use them at your risk. The programs are free; I accept no responsibility for their use nor offer any guarantee of their quality.

There is much free material available on the web. Bill Goffe’s web site “Resources for Economists on the Internet” at <http://econwpa.wustl.edu/EconFAQ/> is an excellent source.

1.8 Further Reading

Several methodological issues are discussed at greater length in the author’s “Computational Economics and Economic Theory: Complements or Substitutes?” In particular, that paper lays out more completely the way in which computational

methods can be used to analyze economic theories. See also Kendrick's "The Wishes" essay on computational economics. Kendrick (1993) is a NSF report on a research agenda for computational economics.

Exercises

These exercises do not presume too much familiarity with economic theory and applications, though they make frequent use of some basic economics problems, savings-consumption and portfolio problems in particular, that are familiar to most readers. A growing list of more interesting exercises are available at the book's web site.

These first exercises are warmup problems; readers having any difficulty should stop and learn a computing language.

1. Write a subroutine (procedure, etc.), called ADDXY, that takes as input two scalars, x and y , and produces $x + y$ as output. Similarly write a routine MXY that takes x and y and produces xy , and a routine DXY that takes x and y and produces x/y . Write a program that reads two scalars, x and y , applies each of MXY, DXY, and ADDXY to x and y , and prints the answers in a readable fashion. For example, the printout should consist of statements like "THE SUM OF — AND — IS —", "THE PRODUCT OF — . . . , and so on.
2. Write a routine FEVAL that takes as input a function of two variables F and two scalars x and y and produces as output the value of $F(x, y)$. Write a program that reads two scalars x and y , applies FEVAL to each of MXY, DXY, and ADDXY together with the scalars x and y , and prints the answers in a readable fashion. For example, the printout should consist of statements like "THE SUM OF — AND — IS —", "THE PRODUCT OF — . . . , and so on.
3. Write a routine that declares a one-dimensional array A of length 10, reads in the values of the components of A from an input file, computes the sum of the components of A , and outputs the sum.
4. Write a routine that declares a two-dimensional array A with 5 rows and 3 columns, reads in the values of the components of A from an input file, computes the sum of the components of each row of A , and outputs these sums.
5. Write a routine that declares a two-dimensional array A with 5 rows and 3 columns, a row vector b of length 5, a column vector c of length 3, reads in the values of the components of A , b , and c from an input file, computes Ac and bA , and outputs the resulting vectors.
6. Write a program to determine the relative speeds of addition, multiplication, division, exponentiation, and the sine function and the arc tangent function on your computer. Do so in various languages and computers, and compare the results across languages and computers.

2 Elementary Concepts in Computation

The study of mathematical algorithms to solve problems arising in real analysis and functional analysis is called *numerical analysis*. Other methods from applied mathematics used to approximate solutions to such problems include *perturbation methods* and *asymptotic analysis*. This chapter gives the basic facts on computers and the elementary ideas used in numerical methods and approximation procedures. Impatient readers may feel that the material in this chapter is too elementary for them. That would be a mistake. While this material is basic, a firm grasp of it and the associated terminology is essential to understanding the later chapters.

Because it is concerned with actual computation on actual computers, numerical analysis begins with some simple ideas. However, there are important differences between pure mathematics and numerical methods. In pure mathematics we deal with pure, abstract objects, many of which have only impure realizations on a computer. In particular, any computer can represent only a finite number of integers and is capable of only finite precision arithmetic. In contrast, in numerical problems we must keep in mind not only the abstract mathematical problems which we are trying to solve but the impure fashion which these problems are represented and solved on the computer.

Further, and more fundamentally, time is a critical constraint on calculation. To paraphrase a renowned pigskin philosopher, speed is not everything, it is the only thing. In pure mathematics, we frequently use processes of infinite length, whereas in real-life computations we have no such luxury. Space limits also constrain us, but in practice it is the time constraint that binds. If it were not for the constraints of time and space, there would be no errors in computation, and numerical and pure mathematics would be identical. Our finite endowment of time makes pure mathematical objects unattainable and forces us to accept the impure, error-prone approximations of real-life computing.

We will consider later in this chapter some ways to economize on the time constraint, and also the errors, their sources in the imperfect computer representations, their propagation, strategies to minimize their impact on the final results, and ways to give economic meaning to the inevitable errors.

2.1 Computer Arithmetic

To understand the limitations of computational methods, let us begin by looking at the way numbers are represented in a computer. Integers are stored in binary form, but only a finite number of integers can be represented. Rational numbers are stored in the form $\pm m2^{\pm n}$ where m , the *mantissa*, and n , the *exponent*, are integers; of course m and n are limited by the fact that the combination of m and n must fit into

the space that the computer allocates for a real number. Individual calculations can be more precise as the permissible range for m is increased, and we can represent a greater range of magnitudes on the computer as the permissible range for n is increased. Even though we pretend to store real numbers, the only numbers explicitly stored in a computer are rational numbers of this form.

The range of numbers that are machine-representable varies greatly across machines; one should always have a good idea of their value when working on a computer. *Machine epsilon* is the smallest *relative* quantity that is machine representable. Formally this is the smallest ϵ such that the machine knows that $1 + \epsilon > 1 > 1 - \epsilon$. It is also important to know *machine infinity*, that is, the largest number such that both it and its negative are representable. *Overflow* occurs when an operation takes machine representable numbers but wants to produce a number which exceeds machine infinity in magnitude. A *machine zero* is any quantity that is equivalent to zero on the machine. *Underflow* occurs when an operation takes nonzero quantities but tries to produce a nonzero magnitude less than machine zero. The analyst must either know these important constants for his machine or make conservative guesses. Much of the software in the public domain contains a section where the user must specify these arithmetic constants.

The other important distinction is between *single precision* and *double precision*. Single precision usually has 6 to 8 decimal digits of accuracy while double precision has 12 to 16 digits. This distinction arose in the early days when machine words were small and the hardware could execute only single precision arithmetic. Today's arithmetic processing units make this distinction of little use; accordingly, modern standards strongly suggest that all work be done in at least double precision.

There are ways to work with even higher precision, but they are generally costly. Some machines have *quadruple precision*; in fact double precision on CRAY supercomputers is really quadruple precision. These distinctions concern the numbers that can be handled by the arithmetic part of the CPU. One can represent more precise numbers as strings of low-precision numbers and then write software that interprets these strings as high-precision numbers. This approach substantially slows down computation and is not discussed in this text. I know of no case related to the topics in this book where double precision on a personal computer is not adequate *if one is using the appropriate algorithms*.

The second important aspect of computer arithmetic are the operations. Inside the CPU is an arithmetic unit that performs at least the basic arithmetic operations of addition, subtraction, multiplication, and division and possibly also performs exponentiation, logarithmic, and trigonometric operations. In older machines, the more advanced operations were performed by software. Today's machines perform these

operations in the arithmetic unit at much greater speed. However, the only operations that are actually executed are the four basic operations; all other operations are approximated using methods discussed in chapter 6. On most computers addition is faster than multiplication which is a bit faster than division. Computing transcendental functions is generally slower; in particular, exponentiation is up to ten times slower than multiplication. The exact relative speeds of the basic mathematical operations vary across machines.

We must always keep in mind the limitations of the computer. In reality, all it can understand are integers, and it can do only simple arithmetic. Everything else is approximated.

2.2 Computer Processing and Algorithms

During a numerical computation, the computer is constantly taking numbers from input, swapping them between storage, internal registers, and the arithmetic units, and ultimately outputting them. There are several processing modes that computers can use. Since it is becoming increasingly important in serious computing to understand these, let us consider the various processing modes in use today.

The simplest operating mode for a computer is the *serial processing mode*. Strictly speaking, a serial processor is one that performs one operation at a time, executing one instruction only when all earlier instructions are finished. This is the mode we usually assume when we write programs.

Most modern computers are not strictly serial machines. It is common for a computer to have separate processors for fetching from or writing to memory, and for reading from or writing to an I/O device, where these devices perform their jobs at the same time that the arithmetic unit is working. In general, such a structure is called *multiprocessing*. Simple forms of multiprocessing are important even in desktop personal computers. Current processors even “look ahead,” making guesses about what the next steps will be and preparing for these future steps while still working on the current step. This preparation can greatly speed execution when the guesses are correct.

A particularly powerful form of multiprocessing is used in *vector processors*. Vector processors, such as those in CRAY supercomputers, have several arithmetic units that can independently and simultaneously perform arithmetic operations. The term vector processing is appropriate because vector processors are particularly good at performing vector and matrix operations. To see this, consider computing the inner product of the vectors (a, b) and (c, d) , which equals $ac + bd$. A vector processor is able to compute the products ac and bd nearly simultaneously, with the results being

reported to the CPU which then assigns an arithmetic unit to compute their sum. The long inner products that are common in linear operations can make good use of the potential power in vector processors.

However, while vector processing was the leading supercomputer design, it did not performed as well as hoped. First, many problems do not have a structure that is amenable to vector processing. Second, even when a problem is “vectorizable,” many programmers do not make the effort to write the code in the special fashion necessary to exploit the vector processor.

The idea of multiple simultaneous processes has been taken to its logical limit in *parallel processing* where more than one CPU is used simultaneously. In *massively parallel processing* we literally have hundreds of fully functional CPUs executing instructions at the same time. Parallel processing is thought to be the way (or at least part of the way) in which we will increase computing power; the human brain is a parallel processor with billions of CPUs. *Distributed computing* connects individual computers to work together to solve problems. Networks of desktop computers communicating via e-mail have been used to solve difficult problems. Since most computers spend most of their time turned off or waiting for people to do something, distributed computing has the potential of creating virtual supercomputers at low cost.

The key task in using parallel processing is organizing the communication among the individual CPUs. The simplest way to organize parallel processing and distributed computing is by a master-slave model. In this model there is one processor, the master, which manages all communication. The master issues commands to its slave processors, where the commands can be single arithmetic operations or complex programs. When a slave processor finishes one command, it tells the master processor the results and waits for the next command. The master processor keeps track of the results of all the slave processors, coordinating their operations and using their results appropriately.

The critical aspect of writing a program for parallel processing is the choice of synchronization. An algorithm is *synchronous* if the master gives out a set of commands to the slaves and waits for all the slaves to report before issuing the next set of commands. An algorithm is *asynchronous* if a slave can be given a new task independent of the progress the other slaves have made with their old tasks. Sometimes it is not known how much time a slave processor needs to perform its assignment. Synchronous algorithms can be held up by one slow processor, thereby being only as fast as the slowest subprocess. Asynchronous algorithms keep all the slave processors working all the time but may not take advantage of all information in the most efficient fashion because of their imperfect coordination. Few algorithms are purely synchronous or asynchronous, but these terms are useful in describing the

coordination problems present in an algorithm. The choice of algorithm will often depend on the extent to which the hardware and software can exploit multiprocessing possibilities.

2.3 Economics of Computation

Discussing efficient computational methods is natural for economists. Numerical analysis is the enterprise of devising algorithms that not only solve problems but do so in a fashion that economizes on scarce resources. In computational work the scarce resources are human programming time, computer runtime, and computer storage space. The basic objective of a numerical approach is to compute approximate solutions with minimal error in minimal computer time, using minimal computer storage and minimal programmer time. This multiple objective forces us to consider many trade-offs, just as any consumer must consider trade-offs among the various uses of his endowment. Numerical analysis and related applied mathematics methods provide us with a large variety of methods that comprises the *production possibility set*. When we attack any particular problem, we must be aware of the relative value of these objectives; that is, we must know our *indifference curves* in order to make an efficient choice of technique. One objective of this book is to describe a large variety of techniques so that readers can find combinations that are efficient for their problems and situations.

Say, as is sometimes true, we want to compute just a few examples of a model to illustrate the quantitative importance of some phenomenon. The greatest cost would likely be the programmer time than the computer runtime. If we further want an accurate result, we would choose an algorithm that is easy to implement and will reliably give an accurate answer. There would be little effort made to minimize computer time.

Say we want to solve several examples in order to find patterns that suggest theorems. In this case we do not need the highest accuracy, but we do care about runtime, since a faster program will produce more examples. Also we do not want to spend a large amount of programmer time on the computational project if the ultimate objective is to prove theorems.

If we rather want to demonstrate a general proposition by purely numerical means, then we will need a large number of accurately computed examples. Limitations on available computer time will likely bind, so it is desirable to have methods that maximize execution speed, even if it is at the cost of extra programmer time. Accuracy is an important factor in this case because we do not want the results to depend on errors, which may be systematic instead of random.

The basic lesson is that the choice of algorithm depends on the goal of the computation and the costs of inputs. For any set of objectives and costs, it is useful to have a variety of methods available. Essentially there is a production possibility frontier that expresses the trade-offs among programmer time, space, accuracy, and runtime. Such a production set approach is useful in thinking about algorithm evaluation. Since the marginal rate of substitution across those objectives can change depending on available resources and opportunity costs, the availability of a variety of methods allows an analyst to find one that suits each situation.

2.4 Efficient Polynomial Evaluation

The importance of speed in calculation is what separates numerical mathematics from pure mathematics. We present an example of a simple computation where some insight allows us to economize on computational costs relative to the “natural” algorithm.

Consider the computational cost of evaluating the polynomial,

$$\sum_{k=0}^n a_k x^k. \quad (2.4.1)$$

We want to know how many additions, multiplications, and exponentiations are needed to evaluate the typical polynomial. The obvious direct method for evaluating a polynomial is to compute the various powers of x , x^2 , x^3 , and so on, then multiply each a_k by x^k , and finally add the terms. This procedure (we call this direct method 1) results in $n - 1$ exponentiations, n multiplications, and n additions. Such a method is obviously expensive because of the exponentiations. Since the powers are all integers, we could replace the expensive exponentiations with multiplications; to do so, we compute x^2 by computing $x x$, then compute $x^3 = (x^2) x$, and so on, replacing the $n - 1$ exponentiations with $n - 1$ multiplications. This is some improvement, yielding direct method 2 and using $2n - 1$ multiplications and n additions.

We can do even better by being clever. The most efficient way to evaluate polynomials is by *Horner's method*. The idea is illustrated by the following identity for a generic third-degree polynomial:

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + x(a_1 + x(a_2 + x \cdot a_3)). \quad (2.4.2)$$

This factorization makes the most use out of each multiplication, bringing us down to three multiplications and three additions for the cubic polynomials.

Table 2.1
Polynomial evaluation costs

	Additions	Multiplications	Exponentiations
Direct method 1	n	n	$n - 1$
Direct method 2	n	$2n - 1$	0
Horner's method	n	n	0

Horner's method can be simply implemented for an arbitrary polynomial. First, we need to determine how a polynomial is represented in the computer. To do this, we define a one-dimensional array $A(\cdot)$ that stores the a_k coefficients. More precisely let $A(k+1) = a_k$ for $k = 0, 1, \dots, N$; the +1 shift is needed because some languages do not permit $A(0)$. Second, we compute the polynomial in accordance with the factorization in (2.4.2). Algorithm 2.1 is a Fortran program (and similar to corresponding BASIC, Gauss, and Matlab programs) which implements Horner's method for computing (2.4.1) at X , and Table 2.1 presents the relative operation counts of three methods for evaluating a degree n polynomial.

Algorithm 2.1 Horner's Method

```
SUM=A(N+1)
DO I=N,1,-1
    SUM=A(I)+SUM*X
ENDDO
```

This example shows us how to solve an important computational problem. It also illustrates the lesson that slight reformulations of a problem, reformulations of no consequence from a theoretical perspective, can result in a dramatic speedup. Even for the simplest problem, ingenuity has a large return.

2.5 Efficient Computation of Derivatives

The computation of derivatives is an important feature of many algorithms. In many problems, particularly in optimization and nonlinear equations, most of the computing time is devoted to obtaining first-order derivatives, such as gradients and Jacobians, and second-order derivatives, such as Hessians. Since derivative computations play an important role in so many problems, we will take up the topic here to teach some important numerical formulas, and investigate the economies of scale which arise in analytical calculations of derivatives.

Finite Differences

The most common ways to compute derivatives are *finite difference* methods. Specifically, if $f : R \rightarrow R$, then one way to approximate $f'(x)$ is the *one-sided finite difference* formula

$$f'(x) \doteq \frac{f(x+h) - f(x)}{h}, \quad (2.5.1)$$

where $h = \max\{\varepsilon x, \varepsilon\}$ is the step size and ε is chosen appropriately,¹ usually on the order of 10^{-6} . The relation between ε and h is motivated by two contrary factors; first, we want h to be small relative to x , but second, we want h to stay away from zero to keep the division and differencing in (2.5.1) well-behaved. More generally, if $f : R^n \rightarrow R$, then the one-sided formula for $\partial f / \partial x_i$ is

$$\frac{\partial f}{\partial x_i} \doteq \frac{f(x_1, \dots, x_i + h_i, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h_i} \quad (2.5.2)$$

with $h_i = \max\{\varepsilon x_i, \varepsilon\}$. Note that the marginal cost of computing a derivative of $f(x, y, z)$ at (x_0, y_0, z_0) equals one evaluation of f , since it is assumed that one computes $f(x_0, y_0, z_0)$ no matter how the derivative is computed. Hence the marginal cost of a finite difference derivative equals its average cost. The problem of computing the Jacobian of a multivariate function $f : R^n \rightarrow R^m$ reduces to the first-order derivative formula (2.5.2), since each element in the Jacobian of f is the first derivative of one of the component functions of f . Elements of the Hessian of $f : R^n \rightarrow R$ are of two types. Cross partials are approximated by

$$\begin{aligned} \frac{\partial^2 f}{\partial x_i \partial x_j} &\doteq \frac{1}{h_j} \left(\frac{f(\dots, x_i + h_i, \dots, x_j + h_j, \dots) - f(\dots, x_i, \dots, x_j + h_j, \dots)}{h_i} \right. \\ &\quad \left. - \frac{f(\dots, x_i + h_i, \dots, x_j, \dots) - f(\dots, x_i, \dots, x_j, \dots)}{h_i} \right), \end{aligned} \quad (2.5.3)$$

and the second partials are approximated by

$$\frac{\partial^2 f}{\partial x_i^2} \doteq \frac{f(\dots, x_i + h_i, \dots) - 2f(\dots, x_i, \dots) + f(\dots, x_i - h_i, \dots)}{h_i^2}. \quad (2.5.4)$$

1. The proper choice of ε is discussed in chapter 7.

Analytic Derivatives

Finite difference methods require multiple evaluations of f and thus can produce excessively large errors. For most functions that arise in economics, we can use algebra and calculus to compute derivatives faster and with more accuracy.

We illustrate this with a function familiar to economists. Suppose that we want to compute both the value of $f(x, y, z) = (x^\alpha + y^\alpha + z^\alpha)^\gamma$ and its gradient $\nabla f \equiv (f_x, f_y, f_z)$. One way is to just analytically derive the derivatives and evaluate them separately. This is typically called *symbolic differentiation*. In this case we would evaluate the three functions $y\alpha x^{\alpha-1}(x^\alpha + y^\alpha + z^\alpha)^{\gamma-1}$, $y\alpha y^{\alpha-1}(x^\alpha + y^\alpha + z^\alpha)^{\gamma-1}$, and $y\alpha z^{\alpha-1}(x^\alpha + y^\alpha + z^\alpha)^{\gamma-1}$, each of which is more costly to compute than the original function. The advantage of symbolic differentiation is the greater accuracy.

If we instead compute the analytic gradient efficiently, the cost will be much lower. The key insight is that by using the chain rule of differentiation, the analytic computation of gradients and Hessians can take advantage of algebraic relations among individual terms. We now compute ∇f using automatic differentiation. First, we compute the original function, $f(x, y, z) = (x^\alpha + y^\alpha + z^\alpha)^\gamma$. Note that this computation produces values for the individual terms x^α , y^α , z^α , $x^\alpha + y^\alpha + z^\alpha$, as well as $(x^\alpha + y^\alpha + z^\alpha)^\gamma$. As we compute $f(x, y, z)$, we store these values for later use. With these values in hand, the computation of $f_x = (x^\alpha + y^\alpha + z^\alpha)^{\gamma-1}y\alpha x^{\alpha-1}$, needs only 2 divisions and 3 multiplications. This is because x^α , $x^\alpha + y^\alpha + z^\alpha$, and $(x^\alpha + y^\alpha + z^\alpha)^\gamma$ are known from the $f(x, y, z)$ computation, and $f_x = (x^\alpha + y^\alpha + z^\alpha)^\gamma/(x^\alpha + y^\alpha + z^\alpha)) \cdot \gamma \cdot \alpha \cdot x^\alpha/x$ just involves 3 extra multiplications and 2 extra divisions. Note also how we have used division to compute the necessary exponentiations. In general, if one knows f , f' , and f^a , then $(f^a)' = a * f^a * f'/f$. This has the extra advantage of using division to compute an exponentiation, a replacement producing considerable time savings on many computers.

When we move to the other derivatives, we are able to realize even more economies of scale. Since $(x^\alpha + y^\alpha + z^\alpha)^{\gamma-1}y\alpha$ is a common factor among the partial derivatives, we only need one more multiplication and one more division to compute f_y , and similarly for f_z . Hence ∇f can be computed at a marginal cost of 4 divisions and 5 multiplications. In contrast, the finite difference method uses 12 exponentiations, 3 divisions, and 9 addition/subtractions. The savings increase as we move to higher dimensions, since the marginal cost of computing a derivative is one multiplication and one division. The study of methods to exploit these relations and create efficient differentiation code is called *automatic differentiation*. Algorithm 2.2 is Fortran code which efficiently computes f and ∇f .

Algorithm 2.2 Program to Compute Gradient of $(x^a + y^a + z^a)^\gamma$

```
XALP=X**ALPHA; YALP=Y**ALPHA; ZALP=Z **ALPHA
SUM=XALP+YALP+ZALP
F=SUM**GAMMA
COM=GAMMA*ALPHA*F/SUM
FX=COM*XALP/X; FY=COM*YALP/Y; FZ=COM*ZALP/Z
```

This is just one example of how careful attention to the form of a function can improve the efficiency of derivative computation. Another example would be the exploitation of separability in any $f(x, y, z)$ of the form $f_1(x) + f_2(y) + f_3(z)$. The savings are even greater when we consider computing the Hessian, which can make use of the computations already performed for the gradient as well as the many algebraic relations among the second derivatives. It is also clear that the savings relative to finite difference methods increase as we examine larger problems.

Arguing against using symbolic and automatic differentiation are the considerable algebraic costs. Many functions are difficult to differentiate. Finite difference methods avoid any errors that may arise from human differentiation of the objective. Even if one could reliably compute derivatives, much of the savings of automatic differentiation comes from recognizing common terms across derivatives, another process that challenges human algebraic abilities. Fortunately we can now take out much of the human error. Derivative computations can be done by symbolic software; in fact much of that software was created to do just this sort of computation. Some symbolic software, such as Maple and Macsyma, can form Fortran and C code which computes the function, its gradient, and Hessian and, furthermore, recognizes common terms. This makes it easy to exploit the economies of computation which are available through the analytical computation of gradients and Hessians.

The extensive literature on automatic differentiation formalizes these ideas and their application to real problems. While these methods are potentially very useful, we will not develop them in this text. Kalaba et al. (1983) and Tesfatsion (1992) are early examples of these methods. These ideas have been systematized in the automatic differentiation literature, which is reviewed in Griewank and Corliss (1991) and Berz et al. (1996). Currently the software for implementing these ideas is not as simple as desired; fortunately there is much work on this front and we should soon have seamless software integrating symbolic and numerical methods. Until then the readers should be aware that this approach is available, and that they can implement these ideas themselves if necessary.

Despite the advantages of analytic derivatives, we will stick with finite differences for the exercises in this text. Finite differences are also adequate for most problems in

economics, and they are easier to implement. It is also advisable to use finite difference methods in the early stages of a project; one does not want to work hard to compute all the derivatives with a Cobb-Douglas production function and then decide to use the constant elasticity of substitution (CES) specification. In general, analytic methods are better considered only when needed for accuracy reasons, or as a final stage of speeding up an otherwise complete program.

2.6 Direct versus Iterative Methods

Methods for solving numerical problems are divided into two basic types. *Direct methods* are algorithms which, in the absence of round-off error, give the exact answer in a predetermined finite number of steps. For example, the solution of $ax = b$ can be computed directly as $x = b/a$. The precision of the answer is as great as the precision with which the machine expresses a and b , and the algorithm consists of one division. In general, direct methods take a fixed amount of time and produce answers of fixed precision. Unfortunately, many problems fail to have direct methods of solution. For example, a fundamental theorem in algebra says that there is no direct method for computing the roots of a polynomial of degree 5 or greater. Even when they exist, direct methods may require too much space or time to be practical.

When direct methods are absent or too time-consuming, we turn to *iterative methods*. These methods have the form $x^{k+1} = g^{k+1}(x^k, x^{k-1}, \dots) \in R^n$. Let x^* denote the desired solution to the underlying problem. The basic questions for an iterative method are whether the sequence x^k converges to x^* , and if convergent how fast it converges. Convergence may depend on the initial guess, x^0 . No matter how fast an algorithm converges or how precise the calculations, we will almost never reach x^* . We must terminate the sequence at some finite point.

Iterative schemes are flexible in that we can control the quality of the result. We may need only a rough approximation obtained in a few iterations and little time, or we may want the greater accuracy obtained by using more iterates. Iterative procedures will generally ask the user for the desired accuracy; no such flexibility is available with direct methods.

2.7 Error: The Central Problem of Numerical Mathematics

The unfortunate fact about computer mathematics is that it is usually approximate. We saw above how the computer representation of numbers necessarily produces

approximation errors. To keep the final error of a computation to a minimum, we need to understand how errors may arise and how they propagate.

Sources of Error

Most errors arise from two basic aspects of numerical operations. First, a computer must round off the results of many arithmetic results; for example, even $1/10$ is not represented exactly on a binary computer. Second, algorithms that implement infinite processes must be stopped before the true solution is reached.

The first source, *rounding*, arises from the fact that the only thing computers can do correctly is integer arithmetic. When we want to represent a number other than an integer or an integral multiple of a (possibly negative) power of two, we must round it. Rounding replaces a number with its nearest machine representable number. This source of error is inherent in the machine representation of real numbers. Increasing the number of bits used to represent a number is the only way to reduce rounding errors.

The second source of error is called *mathematical truncation*. Many mathematical objects and procedures are defined as infinite sums or, more generally, the limit of an infinite process, such as an iterative algorithm. To approximate the limit of an infinite process, we must end it at some finite point. For example, the exponential function is defined as

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}, \quad (2.7.1)$$

but on some computers it is at best $\sum_{n=0}^N x^n/n!$ for some finite N . Mathematical truncation errors are inherent in the desire to compute infinite limits in finite time. One object of numerical analysis is to develop algorithms that have small truncation errors when dealing with infinitistic objects.

Not only do rounding and truncation create errors, they can also turn a continuous function into a discontinuous function. For example, rounding turns the identity function, $f(x) = x$, into a step function. In the case of the exponential function approximated by $\sum_{n=0}^N x^n/n!$, a further discontinuity arises because the choice of N usually differs for different x , and changes in x that cause N to change will generally cause a discontinuity in the computed value of e^x . Many computations involve both kinds of errors. For example, if we want to compute $e^{1/3}$, we must first replace $\frac{1}{3}$ with a computer representable rational number, and then we must use an algorithm which approximates (2.7.1).

Error Propagation

Once errors arise in a calculation, they can interact to reduce the accuracy of the final result even further. Let us take a simple example of this problem to see how to solve it.

Consider the quadratic equation $x^2 - 26x + 1 = 0$. The quadratic formula tells us that $x^* = 13 - \sqrt{168} = .0385186\cdots$ is a solution. Suppose that we want to compute this number with a five-digit machine. On such a machine, rounding makes us accept 12.961 as $\sqrt{168}$. If we proceed in the way indicated by the quadratic formula, the result is

$$x^* = 13 - \sqrt{168} \doteq 13.000 - 12.961 = 0.039 \equiv \hat{x}_1.$$

The relative error of the approximation \hat{x}_1 , $|x^* - \hat{x}_1|/x^*$, is more than 1 percent. The problem here is clear; by subtracting two numbers of similar magnitude, the initial significant digits are zeroed out, and we are left with few significant digits.

A better approach is to rewrite the quadratic formula and produce a second approximation, \hat{x}_2 :

$$13 - \sqrt{168} = \frac{1}{13 + \sqrt{168}} \doteq \frac{1}{25.961} \doteq 0.038519 \equiv \hat{x}_2,$$

which is correct to five significant digits and a relative error of 10^{-5} . Here we have replaced a subtraction between similar quantities with an addition of the same numbers and obtained a much better answer.

The problem in this case arises from the subtraction of two numbers. In general, the following rules help to reduce the propagation of errors when several additions and subtractions are necessary: First, avoid unnecessary subtractions of numbers of similar magnitude, and second, when adding a long list of numbers, first add the small numbers and then add the result to the larger numbers.

This is just one example of how to deal with error propagation. Again we see that considerations and distinctions that are irrelevant for pure mathematics can be important in numerical procedures.

2.8 Making Infinite Sequences Finite

The desired result of any iterative procedure is the limit of an infinite sequence, not any particular member of the sequence. However, we must choose some element of a sequence as an approximation of its limit, since we do not have an infinite amount of time. This problem arises frequently in numerical methods because most algorithms

are iterative. It is, in some sense, an insurmountable problem; since the limit of a sequence does not depend on any initial segment of a sequence, there cannot be any valid general rule based on a finite number of elements for finding the limit. To deal with this problem, and minimize the errors that arise from truncating an infinite sequence, we must use the properties of infinite sequences to develop useful heuristics for approximating their limits.

Rates of Convergence

A key property of a convergent sequence is the rate it converges to its limit. Suppose that the sequence $x^k \in R^n$ satisfies $\lim_{k \rightarrow \infty} x^k = x^*$. We say that x^k converges at rate q to x^* if

$$\lim_{k \rightarrow \infty} \frac{\|x^{k+1} - x^*\|}{\|x^k - x^*\|^q} < \infty. \quad (2.8.1)$$

If (2.8.1) is true for $q = 2$, we say that x^k converges quadratically. If

$$\lim_{k \rightarrow \infty} \frac{\|x^{k+1} - x^*\|}{\|x^k - x^*\|} \leq \beta < 1, \quad (2.8.2)$$

we say that x^k converges linearly; we also say that it converges linearly at rate β . If $\beta = 0$, x^k is said to converge superlinearly. Clearly, convergence at rate $q > 1$ implies superlinear and linear convergence.

Stopping Rules

Since we can only examine an initial portion of a sequence, we need good rules for truncating an infinite sequence. Ideally we will want to stop the sequence only when we are close to the limit. Unfortunately, there is no foolproof, general method of accomplishing this. That is inherent in the nature of a sequence: the limit depends on only the asymptotic behavior of the sequence, whereas any particular point on a sequence illustrates only the initial behavior of the sequence. For example, consider the scalar sequence

$$x_k = \sum_{j=1}^k \frac{1}{j}. \quad (2.8.3)$$

The limit of x_k is infinite, but any particular x_k is finite; in fact the sequence x_k goes to infinity so slowly that even $x_{1000} = 7.485\dots$, giving little hint that the limit is infinite.

We must rely on heuristic methods, *stopping rules*, to determine when to end a sequence. In general, one part of any stopping rule is to stop when the sequence is not “changing much.” Formally this is often expressed as $|x_k - x_{k+1}|/|x_k| \leq \varepsilon$ for some small ε . As long as x_k is large relative to ε , this says that we stop when the change in the sequence $|x_{k+1} - x_k|$ is small relative to $|x_k|$. This kind of rule may never allow us to stop if x_k converges to zero. To account for this, we stop the sequence if it appears that the changes are small or if the limit appears to be close to zero. The rule that combines these considerations is

$$\text{Stop and accept } x_{k+1} \text{ if } \frac{|x_k - x_{k+1}|}{1 + |x_k|} \leq \varepsilon. \quad (2.8.4)$$

Therefore criterion (2.8.4) says to stop if changes are small relative to $1 + |x_k|$. The kind of stopping rule in (2.8.4) is not very good but is about as good as one can do in general. For example, it would fail in (2.8.3), since for any ε the test (2.8.4) will cause us to end (2.8.3) at some finite point whereas the true limit is infinite. For example, $\varepsilon = 0.001$ would end (2.8.3) at $k = 9330$ where $x_k = 9.71827$.

Fortunately we often have information about the sequence that will help us. In particular, if we know the rate of convergence, we can develop a more reliable stopping rule. If we know that a sequence that converges quadratically, a simple rule such as (2.8.4) will probably be adequate if $\varepsilon \ll 1$.

If we know that a sequence is linearly convergent at rate $\beta < 1$, we need a more careful rule than (2.8.4). Knowing β allows us to estimate $\|x^* - x^k\|$. In some cases we can strengthen (2.8.2) to the inequality

$$\|x^{k+1} - x^*\| \leq \beta \|x^k - x^*\| \quad (2.8.5)$$

for all k , not just for sufficiently large k . In this case we know that $\|x^k - x^*\| \leq \|x^k - x^{k+1}\|/(1 - \beta)$. Therefore, if we use the rule

$$\text{Stop and accept } x_{k+1} \text{ if } \|x^k - x^{k+1}\| \leq \varepsilon(1 - \beta), \quad (2.8.6)$$

the error in accepting x^{k+1} is bounded by $\|x^k - x^*\| \leq \varepsilon$.

For problems where we know (2.8.5), the rule (2.8.6) provides us with a completely reliable stopping rule. For other problems we can use this approach by assuming that we have reached a point where the sequence’s terms obey the asymptotic behavior of the sequence and construct a useful heuristic. Suppose that we know that x^k converges linearly and want to conclude that x^n is within ε of x^* . We first compute $\|x^{n-l} - x^n\|/\|x^{n-l-1} - x^n\|$ for $l = 1, 2, \dots$ to estimate the linear rate of convergence; we let the maximum of these ratios be $\hat{\beta}$. If $\hat{\beta}$ is the true convergence rate and

$\|x^n - x^*\| \leq \varepsilon$, then we should also have $\|x^n - x^{n-1}\| \leq (1 - \hat{\beta})\varepsilon$; if this is not true, we should not stop at x^n . More precisely, for small ε and convergence rate β , $\|x^n - x^{n-1}\| \leq \varepsilon(1 - \beta)$ does imply that $\|x^n - x^*\| \leq \varepsilon$; our heuristic turns this into a general stopping rule when we substitute β with an estimate $\hat{\beta}$. Since most useful algorithms are at least linearly convergent, this approach is generally applicable. However, it is important to make some attempt to determine the rate of convergence and estimate the error. To see that the rule in (2.8.6) is better than (2.8.4), apply it to (2.8.3). If we were to apply (2.8.6) to (2.8.3), we would never conclude that the sequence (2.8.3) converges.

Unfortunately, we can never be sure that the part of the sequence we are looking at really displays the limiting behavior of the sequence. For this reason we should never accept an iterate as the solution to a problem solely because of some convergence test. We should always go back to the original problem and ask how well our estimate of a sequence's limit satisfies the original problem.

Changes in the initial point x^0 or other parametric aspects of the problem will create discontinuities in the final result, just as rounding produces discontinuities. The looser the stopping rule, the more pronounced is the discontinuity. This discontinuity is important because we often approximate continuous functions with truncations of infinite sequences and create discontinuities that can affect numerical performance.

The problems listed in the previous paragraphs may lead the reader to conclude that we should never try to judge when a sequence is near its limit. Fortunately, these heuristic rules do quite well for many problems. As long as we use them carefully and remain mindful of their limitations, they are useful guides.

2.9 Methods of Approximation

There are two basic approaches for generating approximate solutions to problems. The following rough discussion makes some imprecise distinctions between them.

The first approach includes standard numerical analysis methods. A numerical method takes a problem (a set of nonlinear equations, a differential equation, etc.) and computes a candidate solution (a vector or a function) that nearly solves the problem. Numerical methods produce solutions of an arbitrarily large number of examples. From this collection of examples can be observed patterns from which we can infer general results. While this set of approximate solutions may be informative, they will not have the compelling logical force of theorems.

Symbolic methods constitute a second approach to approximating solutions of problems. *Perturbation methods* are the best example of these methods, and they rely on Taylor's theorem, implicit function theorems, and related results. Perturbation

methods take a problem that has a known solution for some parameter choice and then derive solutions for similar problems with nearby parameter choices. The results often produce excellent approximations of high-order accuracy. Comparative statics and linearizing around a steady state are perturbation procedures familiar to economists, but they are only the simplest such methods. We will apply a wide variety of perturbation, and related *bifurcation* and *asymptotic* methods, to several economic problems. In many cases perturbation methods can be used to prove theorems for an open set of problems near the initial known case. In this way a theory can be obtained for a nontrivial collection of cases of a model. Perturbation methods can also be used to generate solutions that compete with standard numerical procedures.

Since symbolic methods often proceed by algebraic manipulations of equations, they can be implemented on the computer using symbolic algebra manipulation software, such as Macsyma, Reduce, Maple, and Mathematica. Perturbation methods sometimes combine both algebraic manipulation with standard numerical methods. Another class of symbolic methods is explored in Adomian (1986) and several papers referenced there.

The mathematics literature distinguishes between these methods. In fact, generally, perturbation methods are not discussed in numerical analysis texts. We include both approaches in this book because both are useful in economics, and there is growing interest in merging numerical and symbolic methods to create hybrid methods. We will see an example of the possible synergies between numerical and symbolic methods in chapter 15.

2.10 Evaluating the Error in the Final Result

When we have completed a computation using either numerical or symbolic methods, we want to know how good our answer is. There are two ways to approach this problem. We would really like to know the *magnitude* of the error of the final approximation, but this is often not possible. Alternatively, we take the approximation we have computed and quantify the *importance* of the error to determine whether the approximation is acceptable. The second approach is more generally available.

Error Bounds

When possible, we would always like to put a bound on the actual error, that is, the difference between the true answer, say x^* , and our approximation, say \hat{x} . This is the focus of most mathematical analysis of errors of numerical methods. In some cases we can compute a bound on the size of the error, $\|x^* - \hat{x}\|$. This approach is called

forward error analysis. In the special cases where such bounds can be computed, they can be quite useful; this is the case, for example, in dynamic programming.

Unfortunately, it is usually difficult to determine $\|x^* - \hat{x}\|$ with useful precision. The error bounds that we do have tend to be very conservative, producing, at best, information about the order of magnitude of the error. Many of the error bounds that are available need information about the true solution, which is not available, and must thus be approximated.

The forward approach to error analysis has some general value. For almost any general type of problem, there are special cases where we can determine the true solution with high precision either by using another algorithm that produces an error bound or by deriving a closed-form solution. To gain information about an algorithm's ability to solve the general case, we test it out on the special cases where we can then determine the error. However, error bounds are generally not available for most problems.

Error Evaluation: Compute and Verify

Since we generally cannot bound the error of an answer, we must develop another way to check it. The second, and much more applicable, concept of error evaluation is to measure the extent to which the result of some computation violates some condition satisfied by the true solution. Consider, for example, finding the solution to $f(x) = 0$ for some function f . A numerical solution, \hat{x} , will generally not satisfy $f(\hat{x}) = 0$ exactly. The value $f(\hat{x})$ is a measure of the deviation of $f(x)$ from the target value 0. This deviation may, if properly formulated, provide an economically meaningful measure of the importance of the error implied by the accepting \hat{x} as an approximation of x . I will refer to this procedure as *compute and verify*, since we first *compute* an approximation and then *verify* that it is an *acceptable* approximation according to some economically meaningful criteria. In general, these computation and verification steps are two distinct procedures. Also the verification step involves some additional computation, but this extra cost is generally worth the effort.

To illustrate these ideas, consider the problem $f(x) = x^2 - 2 = 0$. A three-digit machine would produce the answer 1.41. The compute and verify method takes the approximation $\hat{x} = 1.41$ and computes (on the three-digit machine) $f(1.41) = -0.01$. While this computation tells us that 1.41 is not the true answer, the value -0.01 for $f(1.41)$ may help us decide if 1.41 is an acceptable approximation for the zero of $f(x)$.

The value $f(\hat{x})$ can be a useful index of acceptability in our economic problems, *but only if it is formulated correctly*. Suppose that we want to compute the equilibrium price of a single market, where $E(p) = D(p) - S(p)$ is an excess demand

function defined in terms of the demand and supply functions. Suppose that our numerical algorithm produces an approximation \hat{p} such that $E(\hat{p}) = 10.0$. Is that an acceptable error? That depends on $D(\hat{p})$ and $S(\hat{p})$. If $D(\hat{p}) = 10^5$, then the approximation \hat{p} implies an excess demand which equals only one ten-thousandth of the volume of trade. In this case most would consider this error economically unimportant and conclude that our computed answer was fine given all the other noise in our analysis and data. However, if $D(\hat{p}) = 2$, we would not want to accept any \hat{p} that implies $E(\hat{p}) > 2$.

While this example is trivial, it does illustrate our general approach. In general, we first use a numerical or approximation procedure to *compute* a candidate solution \hat{x} to $f(x) = 0$. We then *verify* that \hat{x} is an acceptable solution by showing that $g(\hat{x})$ is small where g is a function (perhaps f itself) or a list of functions that have the same zeros as f . In our excess demand example we would solve $E(p) = 0$ but then compute $g(\hat{p}) \equiv S(\hat{p})/D(\hat{p}) - 1$ to check \hat{p} .

It is fortunate for economists that in many problems the numerical error of a calculation can be evaluated by some verification calculation. One natural approach is to choose g so that $g(\hat{x})$ can be interpreted as measures of optimization errors on the part of agents, or as “leakage” between demand and supply implied by the approximation \hat{x} . If our approximation \hat{x} implies that the agents are making “nearly” optimal decisions and that the difference between demand and supply is “trivial” according to some measure, then we have verified that the approximation \hat{x} is an economically reasonable approximation to the economic problem behind the mathematical equation $f(x) = 0$. We argue that \hat{x} is as plausible a prediction of real-world economic behavior as the true zero of $f(x)$ because we know that real economic agents are not perfect optimizers and that market frictions will keep demand and supply from being exactly equal. Once we have agreed as to what “trivial” and “nearly” mean quantitatively, these interpretations help us evaluate the acceptability of the numerical solutions.

This is one way in which economics differs from physics and the other hard sciences. God may or may not throw dice, but the premise of physics is that he does not make mathematical errors, and therefore there is only one right answer to any physical problem. In contrast, economists do not assign such perfection to the “invisible hand,” economists’ *deus ex machina*, nor to the individuals who participate in a market. These differences affect how economists interpret the errors arising in computations.

In general, the attitude expressed here is that any model is at best an approximation of reality, and a good approximation of the model’s solutions may be as useful an answer as the exact solution. The advantage of numerical methods is that

we can analyze more realistic models, reducing the approximation errors arising from using an excessively simple model. The cost of using numerical methods is the inevitable numerical error, but the gain from model realism is often well worth the cost.

A related approach to error analysis, called *backward error analysis*, is to ask if there are similar problems for which the approximate solution would be an exact solution. For example, to see if $x = 1.41$ is an acceptable solution to $x^2 - 2 = 0$, we ask if there are similar equations for which $x = 1.41$ is the true solution. One such equation is $x^2 - 1.9881 = 0$. If we thought that the equation $x^2 - 1.9881 = 0$ was as good a model of our economic problem as the equation $x^2 - 2 = 0$, then we would accept $x = 1.41$ as the solution to our problem. In general, if we have an approximate solution, \hat{x} , to $f(x) = 0$, and there is another problem $\hat{f}(x) = 0$ for which $\hat{f}(\hat{x}) = 0$ is exactly true, and the differences between f and \hat{f} are economically insignificant, then we accept \hat{x} as an approximate solution.

Both the compute and verify approach and the backward approach to error analysis have some drawbacks; in particular, they both allow for multiple acceptable, “verifiable,” solutions. However, that is already a feature of numerical methods. The existence of multiple acceptable equilibria makes it more difficult for us to make precise statements, such as comparative statics, concerning the nature of equilibrium. However, we could usually run some diagnostics to estimate the size of the set of acceptable solutions. One such diagnostic would be random sampling of x near \hat{x} to see how many other nearby points also satisfy the acceptance criterion.

No matter what approach is taken, it is important that error analysis be implemented to the extent feasible. Many authors make no effort to discuss the methods, if any, they use to check the error in their computational results. While that may be acceptable for analyses applying well-known and previously validated methods in familiar problems, it is not acceptable when introducing a novel method or applying old methods to a substantially different problem. Since easy and cheap diagnostics are generally available, there is no good reason for not performing them. In our discussion of numerical methods, we will frequently discuss checks that can be used. Only by making economically relevant checks on our approximate numerical solutions can computational economists and their work attain and maintain credibility.

2.11 Computational Complexity

Numerical analysts spend much effort on evaluating the computational cost of algorithms, making formal the intuitive economic ideas of the previous section. This work has lead to a literature on optimal algorithms.

This literature focuses on the asymptotic relation between accuracy and computational effort. Specifically, we let ε denote the error and N the computational effort (measured in number of arithmetic operations, number of iterations, or some other appropriate measure of effort). The literature focuses on the limit behavior of $N(\varepsilon)$, the function expressing the computational effort necessary to reduce the error to ε or less, or on the limit behavior of its inverse, $\varepsilon(N)$. For example, if an iterative method converges linearly at rate β and N is the number of iterations, then $\varepsilon(N) \sim \beta^N$ and $N(\varepsilon) \sim (\log \varepsilon)(\log \beta)^{-1}$.

While the results from this literature are interesting, we must be aware of their limitations and understand why, in practice, we often ignore the implications of the asymptotically optimal algorithm theory. If a scheme obeys the convergence rule

$$\lim_{\varepsilon \rightarrow 0} \frac{N(\varepsilon)}{\varepsilon^{-p}} = a < \infty$$

then the number of operations necessary to bring the error down to ε is $a\varepsilon^{-p}$ asymptotically. When discussing the asymptotic rankings of algorithms, only the coefficient p matters, since any differences in p across algorithms will dominate any differences in a . Therefore the asymptotically optimal algorithm is the one with the smallest p .

However, we will not always choose the algorithm with the smallest p . This asymptotic analysis assumes that we have access to infinite-precision arithmetic, whereas we have to live with far less. Suppose that we have one convergent scheme using $a\varepsilon^{-p}$ operations and another asymptotically requiring $b\varepsilon^{-q}$ operations, where $q > p > 0$. Asymptotically the $a\varepsilon^{-p}$ scheme is more efficient, independent of the values of a and b , but this asymptotic superiority may hold only for very small ε . Two such algorithms use equal effort for $\varepsilon^* = (b/a)^{1/(q-p)}$, which is less than one if $a > b$ and $q > p$. Therefore the asymptotically superior method is inferior if our accuracy target is less than ε^* . For example, if $q = 2$, $p = 1$, $b = 1$, and $a = 1,000$, then the critical ε is 0.001; that is, if we need to and can make the error to be less than 0.001, we choose the asymptotically superior algorithm, but otherwise we should choose the “nonoptimal” algorithm.

The asymptotically superior scheme is not always best when we consider the desired level of accuracy and the limits of computer arithmetic. Since our target level of accuracy will vary from problem to problem, it is useful to maintain a collection of algorithms that allows us to make an efficient choice relative to our target level of accuracy.

2.12 Further Reading and Summary

In this chapter we have laid out the basic themes of numerical methods. First, computers do only approximate mathematics, and almost every mathematical object is imperfectly represented on a computer. Second, the errors that arise from these considerations must be controlled to keep them from growing during a calculation. Third, time is the scarce factor here; differences that are unimportant in pure mathematics often take on great importance in numerical mathematics. And fourth, a computation produces only an approximation to the true solution and must be evaluated to see if it is adequate for our purposes.

The topics discussed here are presented in far greater detail in many numerical analysis texts. Acton (1996) examines the nature of error in numerical analysis. See Chaitin-Chatelin and Frayssé (1996) for an extensive analysis of forward and backward error analysis and for strategies to limit error. Paskov (1995) develops a theory of optimal stopping criteria. Traub and Wozniakowski (1980) and Traub et al. (1988) discuss a formal treatment of algorithmic complexity.

Automatic differentiation is a topic of increasing interest. Rall (1981) is one of the original treatments of the problem, and the Griewank and Corliss (1991) and Berz et al. (1996) books are useful compilations of recent work. There are many links to automatic differentiation papers and software at http://www.mcs.anl.gov/autodiff/AD_Tools/.

Exercises

The expressions in exercises 1–5 are taken from Kulisch and Miranker (1986), and they have terms added, subtracted, and multiplied in a particular order. Experiment with equivalent but different orderings. Also compare single-precision and double-precision results, and do these exercises on various computers and software.

1. Compute $(1682xy^4 + 3x^3 + 29xy^2 - 2x^5 + 832)/107751$ for $x = 192119201$ and $y = 35675640$.
2. Compute $8118x^4 - 11482x^3 + x^2 + 5741x - 2030$ for $x = 0.707107$. Use both Horner's method and the straightforward approach.
3. Solve the linear equations $64919121x - 159018721y = 1$ and $41869520.5x - 102558961y = 0$
4. Let

$$f(x) = \frac{4970x - 4923}{4970x^2 - 9799x + 4830}.$$

Use a finite difference approach from section 2.5 to compute $f''(1)$ for $h = 10^{-4}$, 10^{-5} , and 10^{-8} .

5. Compute $83521y^8 + 578x^2y^4 - 2x^4 + 2x^6 - x^8$ for $x = 9478657$ and $y = 2298912$.

6. Write programs to analytically compute the gradients and Hessian of $(\sum_{i=1}^n x_i^5)^2$ for $n = 3, 4, \dots, 10$. Compare running times and accuracy to finite difference methods with $\varepsilon = 10^{-2}, 10^{-3}, 10^{-4}$, and 10^{-5} .
7. Write programs to determine the relative speeds of addition, multiplication, division, exponentiation, and the sine function on your computer.
8. Write a program that takes coefficients a_0, \dots, a_n , and compute the polynomial $\sum_0^n a_i x^i$ by Horner's method. Repeat the exercise for computing $\sum_{i=0}^n \sum_{j=0}^n a_{ij} x^i y^j$ and $\sum_{i=0}^n \sum_{j=0}^n \sum_{l=0}^n a_{ijl} x^i y^j z^l$ using the multivariate generalization (which you must figure out) of Horner's method. Compare the efficiency of Horner's method relative to direct methods 1 and 2.
9. Use the program in exercise 8 to evaluate the polynomial

$$f(x, y, z) = \sum_{i=0}^{10} \sum_{j=0}^{10} \sum_{l=0}^{10} x^i y^j z^l$$

for $x, y, z \in \{0, \pm 0.1, \pm 0.2, \pm 0.3, \pm 0.4, \pm 0.5\}$, and put the results in a three-dimensional table. Write a program that allows you to compare the cost of evaluating $f(x, y, z)$ for the values on this grid, and the cost of looking up a typical value.

10. Write a program that determines your machine ε .
11. Apply the stopping rules (2.8.4) and (2.8.6) to the following sequences: (a) $x_k = \sum_{n=1}^k 3^n / n!$; (b) $x_k = \sum_{n=1}^k n^{-2}$; (c) $x_k = \sum_{n=1}^k n^{-1.001}$; (d) $x_k = \sum_{n=1}^k n^{-0.5}$. Use $\varepsilon = 10^{-2}, 10^{-3}$, and 10^{-4} . For each ε , determine the number of terms computed before the stopping rule makes a choice and compute the error of the answer.

III BASICS FROM NUMERICAL ANALYSIS ON R^n

3 Linear Equations and Iterative Methods

The most basic task in numerical analysis is solving the linear system of equations

$$Ax = b,$$

where $b \in R^n$ and $A \in R^{n \times n}$. This chapter examines techniques for solving linear equations for three reasons. First, they are important problems in themselves. Second, linear equations are almost the only problem we know how to solve directly. Solution methods for linear problems are basic building blocks for much of numerical analysis, since nonlinear problems are often solved by decomposing them into a sequence of linear problems. Understanding how linear problems are solved and the associated difficulties is important for much of numerical analysis. Third, many of the ideas used in this chapter to solve linear problems are applicable to general problems. In particular, introduced are the basic iterative methods. The presentation of iterative schemes for linear equations will give readers the intuition needed to guide them in developing nonlinear iterative methods.

Linear problems are divided into two basic types depending on the entries of A . We say that A is *dense* if $a_{ij} \neq 0$ for most i, j . On the other hand, we say that A is *sparse* if $a_{ij} = 0$ for most i, j . This terminology is obviously imprecise. In practice, however, matrices that arise in numerical problems tend to either be clearly dense or clearly sparse.

This chapter first describes the basic direct methods for solving linear equations. Then it discusses condition numbers for linear systems and the error bounds they imply. The concept of conditioning is one of the most important ones in numerical analysis and appears in almost all numerical problems. The chapter includes the general principles of iterative methods for solving systems of equations and how these iterative methods are fine-tuned. It ends with discussions of sparse matrix methods, applications to Markov chain theory, and solving overidentified systems.

3.1 Gaussian Elimination, *LU* Decomposition

Gaussian elimination is a common direct method for solving a general linear system. We will first consider the solution for a simple class of problems, and then we will see how to reduce the general problem to the simple case.

The simple case is that of *triangular* matrices. A is *lower triangular* if all nonzero elements lie on or below the diagonal; that is, A has the form

$$A = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}.$$

Upper triangular matrices have all nonzero entries on or above the diagonal. A is a *triangular matrix* if it is either upper or lower triangular. A *diagonal* matrix has nonzero elements only on the diagonal. Some important facts to remember are that a triangular matrix is nonsingular if and only if all the diagonal elements are nonzero, and that lower (upper) triangular matrices are closed under multiplication and inversion.

Linear systems in which A is triangular can be solved by *back-substitution*. Suppose that A is lower triangular and nonsingular. Since all nondiagonal elements in the first row of A are zero, the first row's contribution to the system $Ax = b$ reduces to $a_{11}x_1 = b_1$, which has the solution $x_1 = b_1/a_{11}$. With this solution for x_1 in hand, we next solve for x_2 . Row 2 implies the equation $a_{22}x_2 + a_{21}x_1 = b_2$, in which only x_2 is not known. Proceeding down the matrix, we can solve for each component of x in sequence. More formally, back-substitution for a lower triangular matrix is the following procedure:

$$x_1 = \frac{b_1}{a_{11}}, \quad (3.1.1)$$

$$x_k = \frac{b_k - \sum_{j=1}^{k-1} a_{kj} x_j}{a_{kk}}, \quad k = 2, 3, \dots, n, \quad (3.1.2)$$

which is always well-defined for nonsingular, lower triangular matrices. If A is upper triangular, we can similarly solve $Ax = b$ beginning with $x_n = b_n/a_{nn}$.

To measure the speed of this solution procedure, we make an operation count. There are n divisions, $n(n - 1)/2$ multiplications, and $n(n - 1)/2$ additions/subtractions. Ignoring additions/subtractions and dropping terms of order less than n^2 , we have an operation count of $n^2/2$. Therefore solving a triangular system can be done in quadratic time.

We will now use the special method for triangular matrices as a basis for solving general nonsingular matrices. To solve $Ax = b$ for general A , we first factor A into the product of two triangular matrices, $A = LU$ where L is lower triangular and U is upper triangular. This is called a *LU decomposition* of A . We then replace the problem $Ax = b$ with the equivalent problem $LUX = b$, which in turn reduces to two triangular systems, $Lz = b$ and $UX = z$. Therefore, to find x , we first solve for z in $Lz = b$ and then solve for x in $UX = z$.

Gaussian elimination produces such an *LU* decomposition for any nonsingular A , proceeding row by row, transforming A into an upper triangular matrix by means of a sequence of lower triangular transformations. The first step focuses on getting the first column into upper triangular form, replacing a_{il} with a zero for $i = 2, \dots, n$.

Suppose that $a_{11} \neq 0$. If we define $l_{i1}^1 = a_{i1}/a_{11}$, $i = 2, \dots, n$, and $a_{ij}^2 = a_{ij} - l_{i1}^1 a_{1j}$, $j = 1, \dots, n$, then a_{i1}^2 is zero for $i = 2, \dots, n$. Let $A^{(1)} = A$. If we define a new matrix $A^{(2)}$ to have the same first row as A , $A_{ij}^{(2)} = a_{ij}^2$ for $i, j = 2, \dots, n$, and $A_{i1}^{(2)} = 0$, $i = 2, \dots, n$, then

$$\left[I - \begin{pmatrix} 0 & 0 & \cdots & 0 \\ l_{21}^1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1}^1 & 0 & \cdots & 0 \end{pmatrix} \right] A = \begin{pmatrix} a_{11}^1 & a_{12}^1 & \cdots & a_{1n}^1 \\ 0 & a_{22}^2 & \cdots & a_{2n}^2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^2 & \cdots & a_{nn}^2 \end{pmatrix} \equiv A^{(2)}.$$

Note that we have premultiplied A by a lower triangular matrix to get $A^{(2)}$. Proceeding column by column in similar fashion, we can construct a series of lower triangular matrices that replaces the elements below the diagonal with zeros. If $a_{kk}^k \neq 0$, we define

$$l_{ij}^k = \begin{cases} \frac{a_{ik}^k}{a_{kk}^k}, & j = k, i = k+1, \dots, n, \\ 0, & \text{otherwise,} \end{cases} \quad (3.1.3)$$

$$a_{ij}^{k+1} = \begin{cases} a_{ij}^k - l_{ik}^k a_{kj}^k, & i = k+1, \dots, n, j = k, \dots, n, \\ a_{ij}^k, & \text{otherwise,} \end{cases} \quad (3.1.4)$$

then we have defined a sequence of matrices such that

$$A^{(k+1)} = \underbrace{\left[I - \begin{pmatrix} 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & l_{k+1,k}^k & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & l_{n,k}^k & \cdots & 0 \end{pmatrix}\right]}_{L^{(k)}} A^{(k)}.$$

The result is that $A^{(n)}$ is upper triangular, where the factorization

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

implies that $A = LU$ where $L \equiv (L^{(n-1)} \cdots L^{(1)})^{-1}$ is also lower triangular. Algorithm 3.1 summarizes Gaussian elimination.

Algorithm 3.1 Gaussian Elimination

Objective: Solve $Ax = b$.

Step 1. Compute the LU decomposition of A .

Step 2. Solve $Lz = b$ for z by back-substitution.

Step 3. Solve $Ux = z$ for x by back-substitution.

There are two difficulties with Gaussian elimination. First is the possibility that a_{kk}^k is zero, making (3.1.3) ill-defined. However, as long as A is nonsingular, some element of $A^{(k)}$ below a_{kk}^k will be nonzero and a rearrangement of rows will bring a nonzero element to the k th diagonal position, allowing us to proceed; this is called *pivoting*. Even if a_{kk}^k is not zero, a small value will magnify any numerical error from an earlier step. Therefore a good pivoting scheme will use a rearrangement that minimizes this potential problem. See Golub and van Loan (1983) for details about pivoting. Since good pivoting schemes are complex, readers should not write their own LU codes but use the refined codes, such as the programs in LAPACK, which are available. Anderson et al. (1992) describes this package. Second, round-off error can accumulate because of the mixture of additions and subtractions that occur in Gaussian elimination.

To measure the speed of this procedure, we again perform an operation count. The factorization step involves roughly $n^3/3$ multiplications and divisions. Solving the two triangular systems implicit in $LUx = b$ uses a total of n^2 multiplications and divisions. The cost of solving a linear problem depends on the context. Since the factorization cost is borne *once* for any matrix A , if you want to solve the linear problem with m different choices of b , the total cost is $n^3/3 + mn^2$. Therefore the fixed cost is cubic in the matrix dimension n , but the marginal cost is quadratic.

3.2 Alternative Methods

LU decomposition is the most common approach to solving linear equations, but there are other useful methods. This section discusses some of the alternatives and indicates their advantages.

QR factorization

We say that A is *orthogonal* if $A^\top A$ is a diagonal matrix. A *QR factorization* for an arbitrary real nonsingular square matrix A is $A = QR$ where Q is orthogonal and R is upper triangular. Once in this form, we solve $Ax = b$ by noting that $Ax = b$ iff $Q^\top Ax = Q^\top b$ iff $Q^\top QRx = Q^\top b$ iff $DRx = Q^\top b$ where $D = Q^\top Q$ is a diagonal

matrix. Therefore DR is upper triangular, and x can be computed by applying back-substitution to $DRx = Q^T b$. The main task in this procedure is computing Q and R . See Golub and van Loan for the details; in particular, they show that this can be done for arbitrary matrices $A \in R^{n \times m}$.

Cholesky Factorization

The LU and QR decompositions can be applied to any nonsingular matrix. Alternative factorizations are available for special matrices. *Cholesky factorization* can be used for symmetric positive definite matrices. Cholesky factorization is used frequently in optimization problems where symmetric positive-definite matrices arise naturally.

A Cholesky decomposition of A has the form $A = LL^T$ where L is a lower triangular matrix. L is a Cholesky factor, or “square root” of A . This is just a special case of LU decomposition since L^T is upper triangular; hence, after computing the Cholesky decomposition of A , one proceeds as in the LU decomposition procedure to solve a system $Ax = b$.

Clearly, A is symmetric positive definite if it has a Cholesky decomposition. One can show by construction that if A is a symmetric positive definite matrix, then A has a Cholesky decomposition. The equation defining a_{11} in $A = LL^T$ implies that $a_{11} = l_{11}^2$. Hence $l_{11} = \sqrt{a_{11}}$; to pin down any element of L , we will always take positive square roots. In general, the first column of $A = LL^T$ implies that $a_{i1} = l_{i1}l_{11}$, $i = 2, \dots, n$, yielding

$$l_{i1} = \frac{a_{i1}}{l_{11}}, \quad i = 2, \dots, n. \quad (3.2.1)$$

When we move to the second column of A , we first find that $A = LL^T$ implies the equation $a_{12} = l_{11}l_{21}$ which fixes l_{21} . Since l_{11} and l_{21} have been fixed, and $l_{11}l_{21} = a_{21}$, this demands that $a_{12} = a_{21}$, which is true when A is symmetric.

The second condition implied by the second column of $A = LL^T$ is $a_{22} = l_{21}^2 + l_{22}^2$, which implies that $l_{22} = \sqrt{a_{22} - l_{21}^2}$. The difficulty here is that $a_{22} - l_{21}^2$ must be positive if l_{22} is to be real. Fortunately that is true given our solution for l_{21} and the assumption that A is positive definite. The other elements of the second column in $A = LL^T$ imply that $a_{i2} = l_{i1}l_{21} + l_{i2}l_{22}$, $i = 3, \dots, n$, which implies, since l_{i1} is known, $i = 1, \dots, n$, that

$$l_{i2} = \frac{a_{i2} - l_{i1}l_{21}}{l_{22}}, \quad i = 3, \dots, n. \quad (3.2.2)$$

Proceeding sequentially in column j , we arrive at the conditions

$$\begin{aligned} a_{jj} &= \sum_{k=1}^j l_{jk}^2, \\ a_{ij} &= \sum_{k=1}^j l_{ik}l_{jk}, \quad i = j+1, \dots, n. \end{aligned} \tag{3.2.3}$$

If A is symmetric positive definite, we can use these conditions sequentially to solve for the elements of L and arrive at the Cholesky decomposition of A .

The advantages of the Cholesky decomposition is that it involves only $n^3/6$ multiplications and n square roots, which is about half the cost of Gaussian elimination for large n . It is also more stable than LU decomposition, particularly since there is no need for pivots. Therefore, if A is known to be symmetric positive definite, Cholesky decomposition is preferred to LU decomposition.

An important application of Cholesky decomposition occurs in probability theory. If $Y \sim N(\mu, \Sigma)$ is a multivariate normal random variable with mean μ and variance-covariance matrix Σ , then Σ is positive definite. If $\Sigma = \Omega\Omega^\top$ is a Cholesky decomposition of Σ , then $Y = \mu + \Omega X$ where $X \sim N(0, I)$ is a vector of i.i.d. unit normals. This tells us that any multivariate normal is really a linear sum of i.i.d. unit normals, a very useful transformation.

Cramer's Rule

Cramer's rule solves for x in $Ax = b$ by applying a direct formula to the elements of A and b . We first need to compute $\det(A)$. The (i,j) cofactor of the a_{ij} element of a $n \times n$ matrix A is the $(n-1) \times (n-1)$ matrix C formed from A by eliminating row i and column j of A . The determinant is computed recursively by

$$\det(A) = \begin{cases} A, & A \text{ a scalar,} \\ \sum_i (-1)^{i+1} a_{ij} \det(C_{ij}), & C_{ij} \text{ the } (i,j) \text{ cofactor of } A. \end{cases} \tag{3.2.4}$$

Let A_k be equal to A except that its k 'th column is b , and let D_k be the determinant of A_k . Then Cramer's rule says that the solution to $Ax = b$ is

$$x_k = \frac{D_k}{D}, \quad k = 1, \dots, n.$$

Cramer's method is very slow, having an operation count of $\mathcal{O}(n!)$. In symbolic computations, however, Cramer's method is used since it is a closed-form expression for the solution x .

3.3 Banded Sparse Matrix Methods

Decomposition methods can be used for any matrix, but for large matrices they will consume much time and space. In many applications we find ourselves working with matrices that have mostly zero entries. When we have a sparse matrix, we can exploit its special structure to construct fast algorithms that use relatively little space. In this section we introduce some special examples of sparse matrices.

Near-Diagonal Methods

The most trivial example of a sparse matrix is a diagonal matrix, D . If we know that a matrix is diagonal, we need only to store the n diagonal elements in a vector, d , where $d_i = D_{ii}$. This reduces storage needs from n^2 to n . Such matrices are denoted $\text{diag}\{d_1, d_2, \dots, d_n\}$. The inverse of such a matrix is the diagonal matrix formed by inverting the diagonal elements; that is, $D^{-1} = \text{diag}\{d_i^{-1}\}$. Solving a diagonal system of equations is therefore trivial, with $Dx = b$ having the solution $x = D^{-1}b$.

It is also easy to solve sparse matrices where all the nonzero elements are on or close to the diagonal. Consider the case of a *tridiagonal* matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & \cdots & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & a_{n-2,n-3} & a_{n-2,n-2} & a_{n-2,n-1} & 0 \\ 0 & \cdots & \cdots & 0 & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & \cdots & \cdots & 0 & 0 & a_{n,n-1} & a_{nn} \end{pmatrix}.$$

We will solve the equation $Ax = b$ in a direct fashion. First, note that if $a_{j,j+1} = 0$ for some j , then A decomposes into the block form

$$A = \begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix},$$

where A_{11} is the $j \times j$ upper left corner of A and A_{22} is the $(n-j) \times (n-j)$ lower right corner, and solving $Ax = b$ reduces to solving smaller tridiagonal systems. Therefore we assume that $a_{j,j+1} \neq 0$ for $j = 1, \dots, n$.

Consider the variable x_1 . The first equation in $Ax = b$, corresponding to the first row in A , tells us that $x_2 = (b_1 - a_{11}x_1)/a_{12}$; if $c_2 = b_1/a_{12}$ and $d_2 = -a_{11}/a_{12}$, then

$$x_2 = c_2 + d_2 x_1 \tag{3.3.1}$$

expresses the solution for x_2 in terms of x_1 . The second equation in $Ax = b$ involves x_1 , x_2 , and x_3 , but (3.3.1) expresses x_2 in terms of x_1 . Substituting (3.3.1) into $a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$ implies that $x_3 = (b_2 - a_{21}x_1 - a_{22}x_2)/a_{23}$; this expresses of x_3 in terms of x_1 , a relation we denote $x_3 = c_3 + d_3x_1$. Moving through rows 3 through $n - 1$, we can sequentially solve for x_k , $k = 4, \dots, n$, in terms of x_1 ; let $x_k = c_k + d_kx_1$ denote these solutions. As we move through these equations, we compute and store the c_k and d_k coefficients. We next substitute the $x_n = c_n + d_nx_1$ and $x_{n-1} = c_{n-1} + d_{n-1}x_1$ expressions into the n th equation, corresponding to the last row of A , finding

$$\begin{aligned} b_n &= a_{n,n-1}x_{n-1} + a_{nn}x_n \\ &= a_{n,n-1}(c_{n-1} + d_{n-1}x_1) + a_{nn}(c_n + d_nx_1), \end{aligned}$$

which is a single equation in the single unknown x_1 . From this we can compute the solution for x_1 and then compute $x_k = c_k + d_kx_1$ to get the solutions for x_k for $k = 2, \dots, n$.

These ideas can be adapted to solve sparse matrices with particular *banding* properties. A matrix is said to have *bandwidth* $2l + 1$ if all nonzero elements are within l rows of the diagonal. Tridiagonal matrices have bandwidth 3; to solve them, the method above reduces the system to one equation in one unknown, x_1 . In general, a matrix with bandwidth $2l + 1$ can be reduced to a system of l equations in x_1, \dots, x_l , in the same fashion as used above for tridiagonal matrices.

Banded matrices can also be stored efficiently. An $n \times n$ matrix, A , with bandwidth $2l + 1$ can be stored in a smaller, $n \times (2l + 1)$, matrix, B . Specifically, $B_{ij} = A_{i,j-l+i-1}$ when $1 \leq j - l + i - 1 \leq n$, and $1 \leq j \leq 2l + 1$, and zero otherwise. This can greatly economize on storage if n is large and l small.

3.4 General Sparse Matrix Methods

Many sparse matrices do not have a special structure such as banding. In this section we present basic ideas for creating and using general sparse matrices.

Storage Scheme for General Sparse Matrices

There are two different ways to approach sparse problems. One way is to find a particular pattern for the nonzero elements and exploit it to develop efficient storage methods and write algorithms specific to that pattern. Banded matrices are one such pattern. Sometimes the nonzero elements in a sparse matrix do not fall into a useful pattern; this forces the use of more general approaches. In this section we will discuss a simple way to store, construct, and use an arbitrary sparse matrix.

We first need to have a way to store all the information in a sparse matrix in a reasonable amount of space. Suppose that we know that $A \in R^{n \times n}$ contains at most N nonzero elements. If a matrix is sparse, we only need to store information about the location and value of nonzero entries. To that end, construct the arrays $B \in R^{N \times 5}$, $ROW \in R^n$, and $COL \in R^n$. Each row of B contains the necessary information about one of the nonzero elements of A . In particular, row l in B contains information about some nonzero a_{ij} . The first three entries in row l give location and value; specifically if $B(l, 1) = i$, $B(l, 2) = j$, then $B(l, 3) = a_{ij}$. If $a_{ij} \neq 0$, then, for some l , the first and second entries of row l of B are i and j , and the third entry is the value a_{ij} .

If we know a_{ij} , we often need to know the next nonzero entries in row i and column j of A . Given that row l in B represents the information about element a_{ij} , the fourth entry of row l in B tells one which row of B contains the next nonzero entry in row i of A ; that is, if $i = B(l, 1)$, and $l^R = B(l, 4)$, then $a_{i,k} = 0$ for $j < k < B(l^R, 2)$. Finally, the fifth entry tells us which row of B contains the next nonzero entry in column j of A ; that is, if $j = B(l, 2)$, and $l^C = B(l, 5)$, then $a_{k,j} = 0$ for $i < k < B(l^C, 1)$. We interpret nonsense entries (something not between 1 and N ; we use zero for this purpose) in the fourth (fifth) entry of a row in B as telling us that we have reached the end of that row (column). To help us get started, ROW_i tells us what row in B contains the first nonzero element in row i of A ; if there is no such entry, let $ROW_i = 0$. Similarly COL_j tells us what row in B represents the first nonzero element in column j of A ; if there is no such entry, let $COL_j = 0$. Figure 3.1 displays row l of B and related rows.

Essentially B , ROW , and COL give us a map for tracing through the nonzero elements of A . For example, if we want to know the value of $a_{i,j}$, ROW_i would tell us

⋮					
row l of B :	i	j	a_{ij}	l^R	l^C
⋮					
row l^R of B :	i	j'	$a_{j'}$	l^{RR}	l^{RC}
⋮					
row l^C of B :	i'	j	$a_{i'j}$	l^{CR}	l^{CC}
⋮					

Figure 3.1
Sparse matrix storage

the column that contained the first nonzero element in row i . From that point we start tracing through row i and continue until we find a_{ij} represented in B , or find that we have passed the (i, j) element, and conclude that $a_{ij} = 0$. Algorithm 3.2 describes how to find the value of a_{ij} .

Algorithm 3.2 Sparse Matrix Lookup

Objective: Given i, j , find the value of a_{ij} in a sparse matrix stored in the general form in $B \in R^{N \times 5}$.

Step 1. $l = ROW_i$; if $l = 0$, then $a_{ij} = 0$, and STOP.

Step 2. $j' = B(l, 2)$

Step 3. If $j' < j$, then $l = B(l, 5)$ and go to step 2;

elseif $j' = j$ $a_{ij} = B(l, 3)$ and STOP;

else $a_{ij} = 0$ and STOP;

endif

Creating a Sparse Markov Transition Matrix

Now that we know how to store a sparse matrix, we need to know how to construct one. Suppose that we have a method for determining the elements of a π_{ij} Markov transition matrix Π with a large number of states. This generally takes the form of a function or subroutine which takes (i, j) as input. If Π is sparse, we will want to put it into sparse form use it to compute its stationary distribution. We now present a method for constructing a sparse matrix representation for Π .

Specifically, if there are N nonzero elements of Π we need to form a $N \times 5$ matrix B of the form displayed in figure 3.1. Actually, to compute the stationary distribution, we only need to do left multiplications, $x\Pi$; hence we have no need for column four of B nor the vector ROW . We will construct the remaining elements of B and the vector COL in the following fashion:

First, we initialize COL to be a vector of -1 ; a -1 value for COL_j will indicate that we have yet to find a nonzero element in column j . We will also use another vector $LAST$. $LAST_j$ will indicate the row in B that contains the last nonzero entry we have found in column j ; $LAST$ will also be initialized to -1 . The variable l will tell us the next row in B to be filled; l is initialized to 1 .

We will traverse Π one row at a time. That is, we begin with row one and find the first column, say j , that contains a nonzero element. This will be our first entry in B ; hence $B(1, 1) = 1$, $B(1, 2) = j$, $B(1, 3) = \pi_{1,j}$, $COL_j = 1$, and $LAST_j = 1$. We increment l , making it 2 , and then proceed to the next nonzero element in row one of Π .

In general, if the most recent element we have found lies in row i of Π and is coded in row l of B , then we proceed as indicated in steps 1 through 6 of algorithm 3.3:

Algorithm 3.3 Constructing a Sparse Markov Matrix

Objective: Given a procedure to compute \prod_{ij} for any i and j , construct a sparse matrix that represents \prod_{ij} .

Initialization. Set elements in COL , $LAST \in R^n$ to be -1 . Let j be the column containing the first nonzero element in row $i = 1$. Set $B(1, 1) = 1$, $B(1, 2) = j$, and $B(1, 3) = \pi_{i,j}$, $COL_j = 1$, and $LAST_j = 1$. Set $l = 2$.

Step 1. Find the next nonzero element in row i , and go to step 2. If there are no more nonzero elements in row i , then increment i , and go to step 1; if we have no more rows, go to step 6.

Step 2. Let j equal to the column of the new nonzero element of row i in Π . Set $B(l, 1) = i$, $B(l, 2) = j$, and $B(l, 3) = \pi_{i,j}$.

Step 3. If $COL_j = -1$, then set $COL_j = i$

Step 4. Set $l = l + 1$.

Step 5. Set $B(LAST_j, 5) = l$ and $LAST_j = l$, and go to step 1.

Step 6. Finishing: For $j = 1, \dots, n$, set $B(LAST_j, 5) = -1$, indicating that the end of a column has been reached. STOP.

This procedure is straightforward. Step 2 deposits the basic information about π_{ij} in row l of B . Step 3 determines if π_{ij} is the first nonzero element in its column and records that fact in COL if true. Step 4 tells us that we move to the next row in B by incrementing l . Step 5 finds, through the $LAST$ vector, the row in B which contains the element in column j immediately above π_{ij} and records there the fact that π_{ij} is the next nonzero element below it. Step 5 also records the fact that row l now contains the most recently found nonzero element of column j . After we have deposited the critical information in row l of B and the appropriate entry in $LAST$, we go back to start again with the next nonzero element of Π . Step 6 finishes up by filling out the empty column 5 entries in B .

Operations with Sparse Matrices

With our sparse matrix representation, we can easily multiply a row vector, x , by A to produce the row vector, $y = xA$. A similar procedure computes Ax for a column vector x . To compute y_j , we first let $l_1 = COL_j$; then $B(l_1, 1)$ is the row in A that contains the first nonzero element in column j of A . Hence we begin with the assignment $y_j = x_{B(l_1, 1)} B(l_1, 3)$. Next let $l_2 = B(l_1, 5)$, since $B(l_1, 5)$ is the row of B

containing information about the next nonzero element of A below the current one. If $l_2 = 0$, we know that there are no more nonzero elements, and we are finished. If $l_2 \neq 0$, we update our computation of y_j with $y_j \leftarrow y_j + x_{B(l_2,1)} B(l_2, 3)$. In general, we proceed with the iteration,

$$l_k = B(l_{k-1}, 5)$$

if $l_k = 0$, STOP,

else $y_j \leftarrow y_j + x_{B(l_k,1)} B(l_k, 3)$

for $k = 2, 3, \dots$, continuing until we hit some k where $B(l_k, 5) = 0$, which indicates that we have exhausted column j of A . At this point we conclude that y_j has been computed, and we move onto column $j + 1$ and the computation of y_{j+1} .

We have shown how to represent a sparse matrix, and how to multiply a matrix and a vector. One can also multiply sparse matrices, but this is a more complex operation. Multiplication of sparse matrices leads to *fill-in problems*, that is, the result of the operation is less sparse than the inputs. Fill-in problems are severe for LU and QR decompositions and matrix inversion. Sparse methods are most useful when a single sparse matrix is used repeatedly in matrix-vector operations.

3.5 Error Analysis

After solving $Ax = b$, we will want to know how sensitive our solution is to errors. Since linear systems arise in nonlinear problems, the error analysis for linear systems is the basis for error analysis in other problems. Before proceeding, we need to review some basic concepts in matrix analysis.

Matrix Analysis

In our work below, we will need a measure of the “size” of a matrix. We can base a concept of matrix size on any notion of vector length. If $\|\cdot\|$ is a norm on R^n , then the induced norm of A is

$$\|A\| \equiv \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \max_{\|x\|=1} \|Ax\|$$

The following theorem ties together the notions of matrix norm and spectral radius:

THEOREM 3.5.1 For any norm $\|\cdot\|$, $\rho(A) \leq \|A\|$.

Error Bounds

We now turn to the analysis of errors that arise in solving $Ax = b$. Specifically, suppose that b is perturbed by errors, such as rounding errors, but there are no errors in A nor any that arise in solving $Ax = b$. Let r be the error in b and \tilde{x} be the solution of the perturbed system; therefore $A\tilde{x} = b + r$. If we define the error $e \equiv \tilde{x} - x$, then by linearity of A , $e = A^{-1}r$.

Suppose that we have chosen a vector norm, $\|\cdot\|$. The sensitivity of the solution of $Ax = b$ to errors will be measured by the *condition number*, and it is defined to be the ratio

$$\frac{\|e\|}{\|x\|} \div \frac{\|r\|}{\|b\|},$$

which is the percentage error in x relative to the percentage error in b . The smallest possible condition number for a linear system of equations is 1. If $A = aI$, then $x = b/a$, and any percentage error in b leads to an identical percentage error in x . This is the best possible case.

The condition number is essentially the elasticity of the solution to a problem with respect to the data, and it can be computed for any problem once an appropriate norm has been chosen. A small condition number is desirable because it indicates that the problem is less sensitive to rounding errors in inputs. While we will here only analyze the condition number of linear problems, it is clear from the definition that the concept can be applied to any problem in general.

Note that $Ae = r$ implies that $\|A\| \|e\| \geq \|r\|$ and that $e = A^{-1}r$ implies that $\|A^{-1}\| \|r\| \geq \|e\|$. Together with $Ax = b$ this implies that

$$\frac{\|r\|}{\|A\| \|A^{-1}\| \|b\|} \leq \frac{\|e\|}{\|x\|} \leq \frac{\|A^{-1}\| \|A\| \|r\|}{\|b\|}.$$

If we define the *condition number* of A to be

$$\text{cond}(A) \equiv \|A\| \|A^{-1}\|,$$

then we have upper and lower bounds

$$\frac{1}{\text{cond}(A)} \frac{\|r\|}{\|b\|} \leq \frac{\|e\|}{\|x\|} \leq \frac{\|r\|}{\|b\|} \text{cond}(A). \quad (3.5.1)$$

These constructions indicate that the condition number is a useful measure of a matrix being nearly singular; it appears to be the only way to do so. It is clear that the

determinant is not a measure of singularity. Even though a zero determinant is a definition of being singular, having a small determinant is not a useful measure of being nearly singular. For example, if $A = \varepsilon I_n$, then $\text{cond}(A) = 1$ and A is a well-behaved matrix, but $\det(A) = \varepsilon^n$ can be arbitrarily small.

What constitutes “small” and “large” condition numbers depends on the machine. Since we are concerned with the number of significant decimal digits, a condition number of 100 indicates that an input error of 10^{-10} results in an output error of up to 10^{-8} , that is, a possible loss of two significant decimal digits. The rough rule of thumb is that as $\text{cond}(A)$ increases by a factor of 10, you lose one significant digit in the solution of $Ax = b$. Normal computers carry about 12 to 16 significant decimal digits. On those machines the condition number is “small” if its base 10 logarithm is about two or three. Any condition number over 10^{10} is considered “large” and unacceptable. Some machines (e.g., double precision on a Cray and quadruple precision on a VAX) carry almost 30 significant decimal digits. On those machines one would not start to worry about poor conditioning unless the condition number exceeds 10^{15} .

Even though the error bound expressions above are natural, they depend on the norm used. It would be more convenient to have norm-independent error bounds. Since $\|A\| \geq \rho(A)$, $\text{cond}(A) \geq \rho(A)\rho(A^{-1})$. A property of eigenvalues is that $\lambda \in \sigma(A)$ if and only if $\lambda^{-1} \in \sigma(A^{-1})$. Hence, for any norm,

$$\text{cond}(A) \geq \frac{\max_{\lambda \in \sigma(A)} |\lambda|}{\min_{\lambda \in \sigma(A)} |\lambda|} \equiv \text{cond}_*(A),$$

where $\text{cond}_*(A)$ is the *spectral condition number* of A .

We would like to replace $\text{cond}(A)$ in (3.5.1) with $\text{cond}_*(A)$, since the result would be useful bounds on the errors. Unfortunately, we cannot make that substitution on the right-hand side of (3.5.1) because $\text{cond}_*(A)$ is only a lower bound for $\text{cond}(A)$. However, there are norms that have condition functions arbitrarily close to $\text{cond}_*(A)$; hence we can find norms that nearly allow us to make this substitution.

Therefore we use $\text{cond}_*(A)$ as a norm-independent lower bound to the true condition number, making $\|r\|/\text{cond}_*(A)\|b\|$ a lower bound of the maximum error. Numerical procedures typically use $\text{cond}_*(A)$ as an estimate of the condition number. While this substitution results in some error, we are generally not concerned because we really want only a rough approximation of the condition number corresponding to the norm of interest. Another commonly used condition number is the L^∞ condition number, $\text{cond}_\infty(A) \equiv \|A\|_\infty \|A^{-1}\|_\infty$, where $\|A\|_\infty \equiv \max_{i,j} |a_{i,j}|$.

Table 3.1
Condition numbers of Hilbert matrices

n	3	4	5	6	7	8	9	10
cond.(H_n)	55	6,000	4.8(5)	1.5(7)	4.8(8)	1.5(10)	4.9(11)	1.6(13)
cond. $_{\infty}$ (H_n)	192	6,480	1.8(5)	4.4(6)	1.3(8)	4.2(9)	1.2(11)	3.5(12)

To illustrate the possible values of the spectral condition number, we now consider a simple example. Consider the Hilbert matrix:

$$H_n \equiv \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{n} & \cdots & \cdots & \cdots & \frac{1}{2n-1} \end{pmatrix}.$$

Table 3.1 displays the spectral and L^{∞} condition numbers for some small Hilbert matrices.

Note the similar magnitudes of $\text{cond}_*(H_n)$ and $\text{cond}_{\infty}(H_n)$. Even though the norms underlying these condition numbers are substantially different, the condition numbers are “close” in terms of their orders of magnitude, which is really all that matters for our purposes. The condition numbers for these Hilbert matrices rise rapidly as n increases. Anyone using a machine with 12 significant digits cannot be confident about any digit in a solution of $H_n x = b$ if n exceeds 9. This is not a rare example. While in some sense “most” matrices have low to moderate condition numbers, many intuitive procedures, such as least squares approximation procedures, lead to poorly conditioned problems with disastrously high condition numbers.

The error bounds we computed were approximate upper bounds, but that was under the assumption that we could exactly solve the perturbed system. Any solution method will introduce further round-off error. This can be particularly true with Gaussian elimination where subtraction plays a critical role in the LU decomposition. A complete analysis of the errors for any particular solution technique is beyond the scope of this text, but it suffices to say that conditioning considerations apply to all methods and provide us with a useful guide for estimating errors. In the case of Gaussian elimination, $\text{cond}_*(A)$ is still a good summary index for the total error in solving $Ax = b$ including round-off errors in the LU and back-substitution steps.

We should finish with one word on terminology. We have defined the condition number in the usual fashion, but the base ten logarithm of this condition number is also referred to as the “condition number” of a matrix. This is sensible because the base ten logarithm indicates the loss in decimal precision. One can usually tell from the context which definition is being used because the numbers differ by orders of magnitude. In this text we will use the phrase “log condition number” to refer to the logarithmic version.

III-conditioned Problems and Preconditioning

The condition number is essentially the elasticity of the solution to a problem with respect to the data. Furthermore it can be computed for any problem, linear or nonlinear, once an appropriate norm has been chosen. In general, the conditioning of the equation $f(x) = 0$ is the condition number of the Jacobian of f at a zero of $f(x) = 0$. A problem is called *well-conditioned* if this elasticity is small and *poorly conditioned* if it is large. The condition number indicates how sensitive the result of an algorithm is to changes in the data and is a property of the mathematical problem, not just a numerical difficulty. We are concerned about it in numerical analysis because errors in the early steps of an algorithm affect the input to later steps. If the later steps are poorly conditioned problems, then the early errors are magnified.

In general, if a problem is poorly conditioned, the hope is that a (possibly nonlinear) transformation will convert it into an equivalent problem which is better conditioned. In the linear case we would consider some simple linear transformations when trying to solve a poorly conditioned system. Suppose that $Ax = b$ is poorly conditioned. *Preconditioning* chooses another nonsingular matrix D and solves the equivalent system $DAx = Db$. If D is chosen well, the new system is better conditioned. See Golub and van Loan (1983) for some examples.

3.6 Iterative Methods

Decomposition methods for linear equations are direct methods of solution. Direct methods can be very costly for large systems, since the time requirements are order n^3 and the space requirements are order n^2 . If A is dense, there is no way to avoid a space requirement of order n^2 . Even if A is sparse, decomposition methods may not be practical because the *LU* and *QR* factors will generally not be sparse.

Fortunately iterative methods are available that economize on space and often provide good answers in reasonable time. In this section we discuss some basic iterative ideas. Studying iteration methods is valuable also because the basic ideas generalize naturally to nonlinear problems.

Fixed-Point Iteration

The simplest iterative approach is *fixed-point iteration* where we rewrite the problem as a fixed-point problem and repeatedly iterate the fixed-point mapping. For the problem $Ax = b$, we define $G(x) \equiv Ax - b + x$ and compute the sequence

$$x^{k+1} = G(x^k) = (A + I)x^k - b. \quad (3.6.1)$$

Clearly x is a fixed point of $G(x)$ if and only if x solves $Ax = b$. Unfortunately, (3.6.1) will converge only if all the eigenvalues of $A + I$ have modulus less than one. Since this is rare, direct application of fixed-point iteration is seldom used.

Gauss-Jacobi Algorithm

One of the simplest iterative methods for solving linear equations is the *Gauss-Jacobi* method. It begins with the observation that each equation is a single linear equation, a type of equation wherein we can solve for one variable in terms of the others. Consider the equation from the first row of $Ax = b$:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1.$$

We can solve for x_1 in terms of (x_2, \dots, x_n) if $a_{11} \neq 0$, yielding

$$x_1 = a_{11}^{-1}(b_1 - a_{12}x_2 - \cdots - a_{1n}x_n).$$

In general, if $a_{ii} \neq 0$, we can use the i th row of A to solve for x_i , finding

$$x_i = \frac{1}{a_{ii}} \left\{ b_i - \sum_{j \neq i} a_{ij} x_j \right\}.$$

Initially this appears to be just a different way to express the linear system. We now turn this into an iterative process. We start with a guess for x and use the single-equation solutions to compute a new guess for x . This can be compactly expressed (assuming $a_{ii} \neq 0$) as the iteration

$$x_i^{k+1} = \frac{1}{a_{ii}} \left\{ b_i - \sum_{j \neq i} a_{ij} x_j^k \right\}, \quad i = 1, \dots, n. \quad (3.6.2)$$

The result is the Gauss-Jacobi method, and it reduces the problem of solving for n unknowns simultaneously in n equations to that of repeatedly solving n equations with one unknown. This only defines a sequence of guesses; the hope is that (3.6.2) will converge to the true solution, a problem to which we return below.

Gauss-Seidel Algorithm

In the Gauss-Jacobi method we use a new guess for x_i , x_i^{k+1} , only after we have computed the entire vector of new values, x^{k+1} . This delay in using the new information may not be sensible. Suppose that x^* is the solution to $Ax = b$. If, as we hope, x_i^{k+1} is a better estimate of x_i^* than x_i^k , using x_i^{k+1} to compute x_{i+1}^{k+1} in (3.6.2) would seem to be better than using x_i^k . The intuitive argument is that new information about x_i^* should be used as soon as possible.

The basic idea of the *Gauss-Seidel method* is to use a new approximation of x_i^* as soon as it is available. More specifically, if our current guess for x^* is x^k , then we compute the next guess for x_1 as in (3.6.2),

$$x_1^{k+1} = a_{11}^{-1}(b_1 - a_{12}x_2^k - \cdots - a_{1n}x_n^k),$$

but use this new guess for x_1^* immediately when computing the other new components of x^{k+1} . In particular, our equation for x_2^{k+1} derived from the second row of A becomes

$$x_2^{k+1} = a_{22}^{-1}(b_2 - a_{21}x_1^{k+1} - a_{23}x_3^k - \cdots - a_{2n}x_n^k).$$

In general, to solve $Ax = b$ by the Gauss-Seidel method, define the sequence $\{x^k\}_{k=1}^\infty$ by the iteration

$$x_i^{k+1} = \frac{1}{a_{ii}} \left\{ b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^n a_{ij} x_j^k \right\}, \quad i = 1, \dots, n. \quad (3.6.3)$$

Here we clearly see how the new components of x^{k+1} are used immediately after they are computed, since x_i^{k+1} is used to compute x_j^{k+1} for $j > i$.

Note that in the Gauss-Seidel case, the order in which we solve for the successive components of x matters, whereas the order did not matter for Gauss-Jacobi. This gives the Gauss-Seidel method more flexibility. In particular, if one ordering does not converge, then we could try a different ordering.

Block Gaussian Algorithms

The Gauss-Jacobi and Gauss-Seidel iterative algorithms break a large problem into a large sequence of easy univariate problems. The *block iteration* idea is to break down the large problem into a sequence of smaller, but multivariate, problems. We choose the subproblems so that they are easy to solve exactly but then proceed as in the Gauss-Jacobi or Gauss-Seidel algorithms.

Suppose that we decompose $x \in R^{nm}$ into a list of n subvectors

$$x \equiv \begin{pmatrix} x^1 \\ \vdots \\ x^n \end{pmatrix},$$

where each x^i is in R^m . Decompose b similarly. Next decompose A :

$$A \equiv \begin{pmatrix} A^{1,1} & \cdots & A^{1,n} \\ \vdots & \ddots & \vdots \\ A^{n,1} & \cdots & A^{n,n} \end{pmatrix},$$

where each $A^{k,j} \in R^{m \times m}$. To solve $Ax = b$, the *block Gauss-Jacobi* method executes the iteration

$$A^{i,i}x^{i,k+1} = b^i - \sum_{j \neq i} A^{i,j}x^{j,k}, \quad i = 1, \dots, n, \quad (3.6.4)$$

where $x^{i,k}$ is the i th piece of the k th iterate of x . We express each step in (3.6.4) as a linear equation to emphasize the fact that each $x^{i,k}$ is found by solving a linear equation, whereas the alternative form $x^{i,k+1} = (A^{i,i})^{-1}(b^i - \sum_{j \neq i} A^{i,j}x^{j,k})$ might be interpreted as requiring the inversion of each $A^{i,i}$ matrix. Instead, one should think in terms of solving the separate equations in (3.6.4) by some linear equation solution method, such as *LU*, *QR*, or Cholesky decomposition. Note that any such decomposition used for the first iterate, $k = 1$, can be reused in each later iteration. Therefore the marginal cost of each iteration k in (3.6.4) is quite small if one can store the decompositions of the $A^{i,i}$ blocks.

The block Gauss-Jacobi method does not update any component of x until one has all of the new components. The *block Gauss-Seidel* method uses new components immediately, executing the iteration

$$A^{i,i}x^{i,k+1} = b^i - \sum_{j=1}^{i-1} A^{i,j}x^{i,k+1} - \sum_{j=i+1}^n A^{i,j}x^{i,k}, \quad i = 1, \dots, n. \quad (3.6.5)$$

The iteration in (3.6.5) combines the advantages of block constructions and immediate replacement.

Tatonnement and Iterative Schemes

We now give an example of both Gauss-Jacobi and Gauss-Seidel iteration in a familiar economic context. Suppose that we have the inverse demand equation

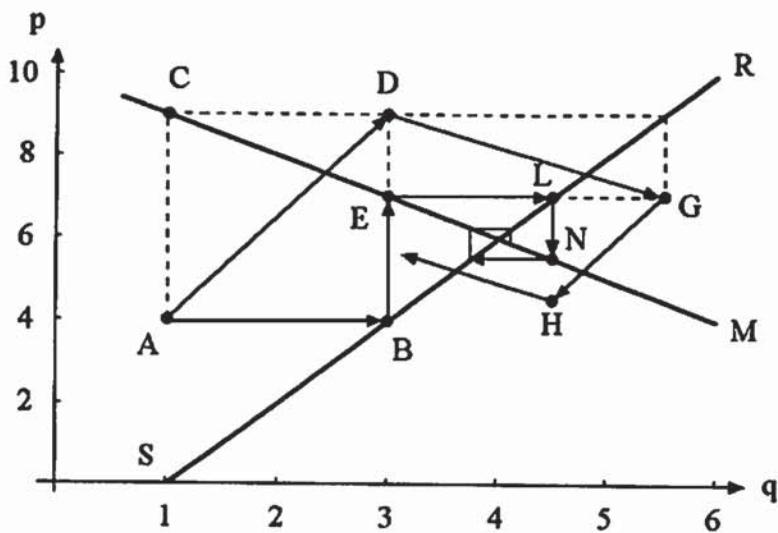


Figure 3.2
Gauss-Jacobi and Gauss-Seidel iteration

$p = 10 - q$ and the supply curve $q = p/2 + 1$, where p is the price and q is quantity.
Equilibrium is where supply equals demand and solves the linear system

$$p + q = 10, \quad (3.6.6a)$$

$$p - 2q = -2. \quad (3.6.6b)$$

In figure 3.2 the demand curve is the line CM and is represented by (3.6.6a), and the supply curve is the line SR represented by (3.6.6b).

In Gauss-Jacobi we make an initial guess $p = 4$ and $q = 1$ represented by point A in figure 3.2. To get a new guess, we solve the demand equation for p holding q fixed by moving up to C on the demand equation from A . Similarly we move to the right from A to the B on the supply equation to solve for q holding p fixed. The new guess encompassing these calculations is point D . This process corresponds to the following auctioneer process: An auctioneer starts with $(p_0, q_0) = (4, 1)$. He then asks demanders what the price will be if supply is q_0 ; their response, $p_1 = 9$, is the new price. He also asks suppliers how much they will produce if price is p_0 ; the producers' response is the new quantity, $q_1 = 3$. The general iteration is

$$\begin{aligned} q_{n+1} &= 1 + \frac{1}{2}p_n, \\ p_{n+1} &= 10 - q_n. \end{aligned} \quad (3.6.7)$$

Repeating this procedure from point D , we next move to G , then H , and so on. This iteration process converges, but slowly as indicated by the list of Gauss-Jacobi

Table 3.2
Gaussian methods for (3.6.6)

Iteration <i>n</i>	Gauss-Jacobi		Gauss-Seidel	
	<i>p_n</i>	<i>q_n</i>	<i>p_n</i>	<i>q_n</i>
0	4	1	4	1
1	9	3	7	3
2	7	5.5	5.5	4.5
3	4.5	4.5	6.25	3.75
4	5.5	3.25	5.875	4.125
5	6.75	3.75	6.0625	3.9375
7	5.625	4.125	6.0156	3.9844
10	6.0625	4.0938	5.9980	4.0020
15	5.9766	4.0078	6.0001	3.9999
20	5.9980	3.9971	6.0000	4.0000

iterates in table 3.2. Figure 3.2 also indicates that it spirals into the final equilibrium at $p = 6$ and $q = 4$.

On the other hand, Gauss-Seidel corresponds to a more conventional economic dynamic story. Figure 3.2 also displays the Gauss-Seidel iteration. Start again from *A*. We first use the supply equation to get a new guess for q , q_1 at *B*, but then use the new quantity, q_1 , at *B* to get a new guess for price, p_1 , from the demand equation. That new point is *E*. This corresponds to an auctioneer starting from $A = (p_0, q_0) = (4, 1)$, learning from suppliers that supply will be $q_1 = 3$ if price is $p_0 = 4$; and then learning from demanders that the price will be $p_1 = 7$ if quantity is $q_1 = 3$. This iteration continues to *L*, then *N*, and so on. This is often referred to as the hog cycle—firms expect p_0 , produce q_1 , which causes prices to rise to p_1 , causing production to be q_2 , and so on. The general iteration is

$$\begin{aligned} q_{n+1} &= 1 + \frac{1}{2}p_n, \\ p_{n+1} &= 10 - q_{n+1}. \end{aligned} \tag{3.6.8}$$

The Gauss-Seidel iterates are listed in table 3.2. Note that the Gauss-Seidel method converges more rapidly but that both methods do converge.

3.7 Operator Splitting Approach

While the direct application of fixed-point iteration is not generally useful, it does lead to a powerful approach. The general strategy involves transforming the given

problem into *another* problem with the *same* solution where fixed-point iteration *does* work. We next examine how to generate iterative schemes abstractly. In doing so, we will see that the Gauss-Jacobi and Gauss-Seidel methods are both applications of this generalized fixed-point iteration method.

Suppose that we want to solve $Ax = b$. We first express A as the difference of two operators

$$A = N - P, \quad (3.7.1)$$

thereby “splitting” A into two components. Note that $Ax = b$ if and only if $Nx = b + Px$. Next define the iteration

$$Nx^{m+1} = b + Px^m, \quad (3.7.2)$$

which, if N is invertible, can also be written

$$x^{m+1} = N^{-1}(b + Px^m). \quad (3.7.3)$$

If we choose N so that $Nx^{m+1} = b + Px^m$ is quickly solved, then computing the iterates will also be easy. This simplicity condition on N is critical for any iteration scheme, since there is little point to using an iterative scheme if each iteration itself is a difficult problem.

Iterative schemes fit into this framework. The Gauss-Jacobi method corresponds to the splitting

$$N = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}, \quad P = -\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}.$$

Since N is a diagonal matrix, solving (3.7.2) for x^{m+1} is easy. The Gauss-Seidel method corresponds to the splitting

$$N = \begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}, \quad P = -\begin{pmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & 0 & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 \end{pmatrix}.$$

In this case, solving for x^{m+1} in (3.7.2) is quickly accomplished by back-substitution.

It is clear, however, that these are not the only possible splittings of A . The key requirement is that solving (3.7.2) is easy.

General Iterative Scheme

Notice that our discussion of operator splitting did not really depend on linearity. The application of this approach to nonlinear problems can be expressed in a common abstract fashion. Algorithm 3.4 outlines a general iterative scheme for solving a general operator equation $Ax = 0$. Note that all we need is a split into a “nice” N , which leads to a convergent scheme.

Algorithm 3.4 General Iterative Schema for $Ax = b$

- Step 1.* Find an N with an easily computed N^{-1} , and split the operator $A \equiv N - P$.
- Step 2.* Construct the iterative scheme $x^{m+1} = N^{-1}(b + Px^m)$.
- Step 3.* Find acceleration scheme to ensure and/or speed up convergence.
- Step 4.* Find adaptive scheme to learn acceleration parameters.

3.8 Convergence of Iterative Schemes

Iterative schemes are worthless unless the iterates converge to the solution. Suppose that we have split the linear operator $A = N - P$, where both N and P are linear, and that $Ax^* = b$. Then the linear iterative scheme $Nx^{m+1} = b + Px^m$ implies that the error $e^m \equiv x^* - x^m$ obeys the linear iteration $e^m = (N^{-1}P)^m e^0$. Therefore $e^m \rightarrow 0$ if and only if $(N^{-1}P)^m e^0 \rightarrow 0$. This holds for any e^0 if and only if $\rho(N^{-1}P) < 1$. The fact that the convergence problem reduces to computing $\rho(N^{-1}P)$ may not be comforting since spectral radii are not easy to compute. On the other hand, all we need for an iterative scheme is to find some splitting which is convergent.

In some important cases we can demonstrate convergence. Consider, for example, the case of a diagonal A . If we apply Gauss-Jacobi or Gauss-Seidel to a diagonal matrix, we will converge after only one iteration. This suggests that if a matrix is “close to being diagonal,” these methods will converge. The key concept is *diagonal dominance*. A is diagonally dominant if each diagonal element is greater than the sum of the magnitudes of the nondiagonal elements in its row, that is,

$$\sum_{j \neq i} |a_{ij}| < |a_{ii}|, \quad i = 1, \dots, n.$$

In the case of Gauss-Jacobi and Gauss-Seidel, diagonal dominance is a sufficient condition for convergence.

THEOREM 3.8.1 If A is diagonally dominant, both Gauss-Jacobi and Gauss-Seidel iteration schemes are convergent for all initial guesses.

Proof For Gauss-Jacobi, N is the matrix of diagonal elements of A , and P is the matrix of nondiagonal elements of A . The matrix $N^{-1}P$ takes the nondiagonal elements of A and divides each element in a row by the corresponding diagonal element. By the diagonal dominance of A , each element of $N^{-1}P$ is less than unity and the row sum of magnitudes in each row cannot exceed unity. Hence

$$\sum_j |(N^{-1}P)_{ij}| < 1, \quad i = 1, \dots, n,$$

which in turn implies that $(N^{-1}P)^m \rightarrow 0$ as $m \rightarrow \infty$. See Stoer and Bulirsch (1990) for a proof of the Gauss-Seidel case.

The relevance of diagonal dominance is economically intuitive. For example, if $p \in R^n$ is a vector of prices and $(Ap)_i$ is the excess demand for good i , then diagonal dominance implies that the sensitivity of the excess demand for good i to its own price exceeds the sum of its sensitivities to all other prices. This is also known as *gross substitutability*. There are also several other contexts in which diagonal dominance naturally arises, making Gauss-Seidel and Gauss-Jacobi methods useful. While theorem 3.8.1 applies strictly only to linear equations, it gives an indication of what is important for convergence in both linear and nonlinear applications of the ideas behind the Gauss-Jacobi and Gauss-Seidel methods.

A key feature to note about Gauss-Jacobi and Gauss-Seidel methods is that they are at best linearly convergent and that the rate of convergence is given by the spectral radius of $N^{-1}P$. While we cannot affect the linearity of convergence, we next investigate methods that increase the linear rate of convergence.

3.9 Acceleration and Stabilization Methods

Many iterative schemes diverge or converge slowly. We often resort to acceleration and stabilization schemes. Again we should remember that the procedures below can be tried in any context, linear, and nonlinear, which fits the notation of the operator splitting approach.

Extrapolation and Dampening

We next examine a simple scheme that can stabilize unstable iteration schemes and accelerate slow ones. Suppose that we want to solve the linear problem $Ax = b$. Define $G = I - A$. Consider the iteration

$$x^{k+1} = Gx^k + b. \tag{3.9.1}$$

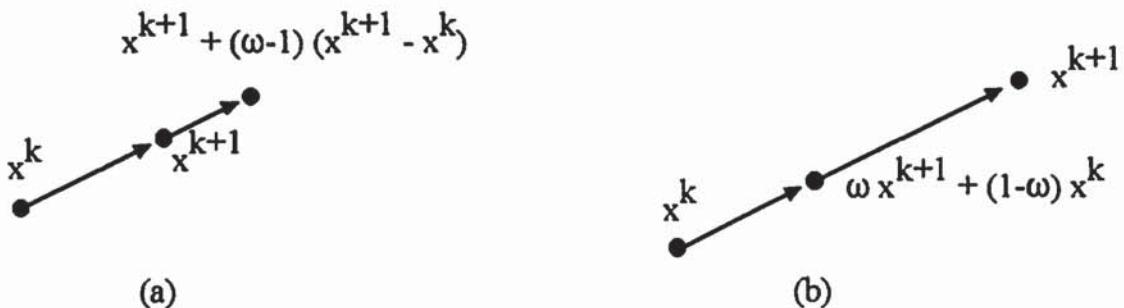


Figure 3.3
Extrapolation (a) and dampening (b)

The scheme (3.9.1) will converge to the solution of $Ax = b$ only if $\rho(G) < 1$, but if $\rho(G)$ is close to 1, convergence will be slow.

Consider next the iteration

$$\begin{aligned} x^{k+1} &= \omega Gx^k + \omega b + (1 - \omega)x^k \\ &\equiv G_{[\omega]}x^k + \omega b \end{aligned} \quad (3.9.2)$$

for scalar ω . When $\omega > 1$, the altered iterative scheme in (3.9.2) is called *extrapolation*. Figure 3.3a illustrates the effect of extrapolation when $\omega > 1$. The iteration in (3.9.1) will take us from x^k to $Gx^k + b$ traversing the vector $Gx^k + b - x^k$; extrapolation sets x^{k+1} equal to $Gx^k + b + (\omega - 1)(Gx^k + b - x^k)$, which stretches x^{k+1} beyond $Gx^k + b$. The idea is that if (3.9.1) converges, then $Gx^k + b - x^k$ is a good direction to move, and perhaps it would be even better to move to a point in this direction beyond $Gx^k + b$ and converge even faster.

When $\omega < 1$, (3.9.2) is called *dampening*. Figure 3.3b illustrates the effect of dampening. The iteration in (3.9.1) will take us from x^k to $Gx^k + b$ traversing the vector $Gx^k + b - x^k$; dampening sets x^{k+1} equal to $(1 - \omega)x^k + \omega(Gx^k + b)$, which is a convex combination of the initial point x^k and the iterate in (3.9.1). The idea here is that if (3.9.1) is unstable, it could be that the direction $Gx^k + b - x^k$ is a good one, but that the point $Gx^k + b$ overshoots the solution and results in oscillation and divergence. Setting $\omega < 1$ and using (3.9.2) results in a damped sequence and hopefully results in convergence.

These ideas are suggestive but will not always work. However, if all of the eigenvalues of G are real, we can surely improve upon (3.9.1) with (3.9.2) for some ω . The stability of $G_{[\omega]}$ depends on its spectrum and spectral radius. If G is nonsingular and has distinct eigenvalues,¹ there is a nonsingular matrix P such that $G = P^{-1}DP$

1. The Jordan canonical form can be used to show this for other matrices G .

where D is the diagonal matrix of eigenvalues of G , implying that $\omega G + (1 - \omega)I$ can be written as $P^{-1}(\omega D + (1 - \omega)I)P$ and that $\sigma(\omega G + (1 - \omega)I) = \omega\sigma(G) + 1 - \omega$. From this we see that in the definition of $G_{[\omega]}$, the scalar ω first stretches or shrinks the spectrum of G depending on the magnitude of ω , flips it around 0 if ω is negative, and finally shifts it by $1 - \omega$.

We want to choose ω so as to minimize $\rho(G_{[\omega]})$. When all eigenvalues of G are real, this reduces to

$$\min_{\omega} \max_{\lambda \in \sigma(\omega G + (1 - \omega)I)} |\lambda|. \quad (3.9.3)$$

If m is the minimum element of $\sigma(G)$ and M the maximum, then the optimal ω will force the minimal and maximal eigenvalues of $G_{[\omega]}$ to be centered around zero and equal in magnitude; hence $-(\omega m + (1 - \omega)) = \omega M + 1 - \omega$, which implies that the solution to (3.9.3) is

$$\omega^* = \frac{2}{2 - m - M} \quad (3.9.4)$$

and that the new spectral radius is

$$\rho(G_{[\omega^*]}) = \left| \frac{M - m}{2 - M - m} \right|. \quad (3.9.5)$$

Note that if $M < 1$, then $\rho(G_{[\omega^*]}) < 1$, no matter what m is. Therefore, as long as no eigenvalues exceed 1 algebraically, we can find an ω^* that produces a stable iteration. Even if G converges, that is, $-1 < m < M < 1$, we can accelerate convergence by shifting the spectrum so that it is symmetric about 0. Only when $M > 1$ and $m < -1$, that is, when G has both kinds of unstable roots, does (3.9.5) fail.

Successive Overrelaxation

Associated with Gauss-Seidel method is an extrapolation procedure called *successive overrelaxation* (SOR) method. For scalar ω , define

$$x_i^{k+1} = \omega \left(\frac{1}{a_{ii}} \right) \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^n a_{ij} x_j^k \right] + (1 - \omega) x_i^k. \quad (3.9.6)$$

This makes the i th component of the $k+1$ iterate a linear combination, parameterized by ω , of the Gauss-Seidel value and the k th iterate.

Note that if the matrix A is decomposed into its diagonal, D , elements below the diagonal, L , and elements above the diagonal, U , then $A = D + L + U$ and SOR has the convenient matrix representation

$$(D + \omega L) x^{k+1} = ((1 - \omega) D - \omega U) x^k + \omega b.$$

If we define $M_\omega \equiv D + \omega L$ and $N_\omega \equiv (1 - \omega) D - \omega U$, then

$$x^{k+1} = M_\omega^{-1} N_\omega x^k + \omega M_\omega^{-1} b. \quad (3.9.7)$$

To enhance convergence, the best choice for ω is the solution to $\min_\omega \rho(M_\omega^{-1} N_\omega)$. The benefit from acceleration can be large, since the optimal ω , ω^* , can greatly increase the convergence rate. The problem is that ω^* is costly to compute. To address that problem, algorithms have been developed that initially set ω equal to some safe value and then adaptively adjust ω , using information about A generated as the iterations proceed. These *adaptive SOR* methods test several values for ω and try to move toward the best choice. See Hageman and Young (1981) for details.

We have discussed the ideas of stabilization and acceleration only for linear Gauss-Jacobi and Gauss-Seidel schemes. These ideas can also be used to stabilize and accelerate nonlinear methods, as indicated in algorithm 3.4.

Stabilizing an Unstable “Hog Cycle”

Figure 3.2 displayed a stable application of Gauss-Seidel to a tatonnement process. A different example is illustrated in figure 3.4. Here the inverse demand equation is $p = 21 - 3q$, line BD in figure 3.4, and the supply curve is $q = p/2 - 3$, line EC in

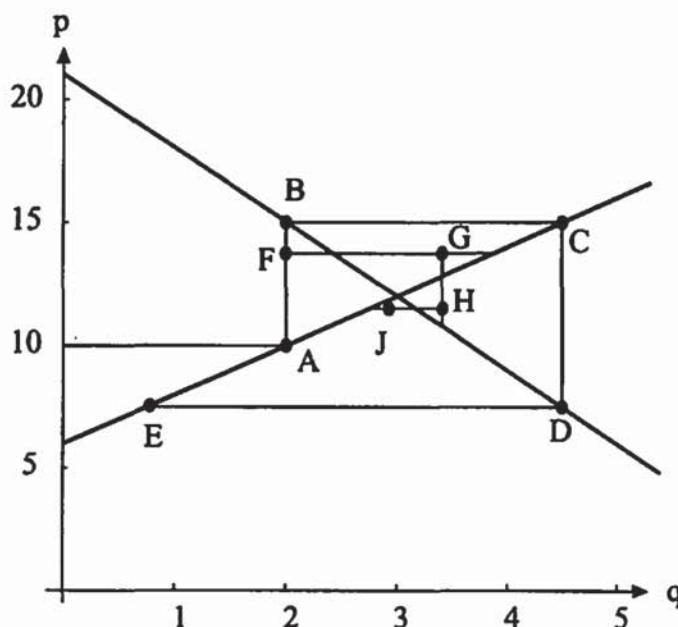


Figure 3.4
Dampening an unstable hog cycle

figure 3.4. The equilibrium system is

$$\begin{aligned} p + 3q &= 21, \\ p - 2q &= 6. \end{aligned} \tag{3.9.8}$$

The system (3.9.8) is not diagonally dominant, indicating that convergence may be problematic. In fact figure 3.4 shows that Gauss-Seidel applied to (3.9.8) produces the iterative scheme

$$p_{n+1} = 21 - 3q_n, \tag{3.9.9a}$$

$$q_{n+1} = \frac{1}{2}p_{n+1} - 3, \tag{3.9.9b}$$

which is unstable. If we begin (3.9.9) at point A , where $(p, q) = (10, 2)$, the first iteration moves us first to B then C , and the second iteration moves us to D and then E . Since E is farther from the equilibrium than A , the iteration is diverging.

We will apply (3.9.6) to stabilize (3.9.9). If we choose $\omega = 0.75$, we arrive at the iteration

$$p_{n+1} = 0.75(21 - 3q_n) + 0.25p_n, \tag{3.9.10a}$$

$$q_{n+1} = 0.75(p_{n+1} - 3) + 0.25q_n. \tag{3.9.10b}$$

The difference between (3.9.9) and (3.9.10) is displayed in figure 3.4. In (3.9.9a) we move from the point A on the supply curve up to the point B on the demand curve, whereas in (3.9.10a) we move only 75 percent of the way, stopping at F . Equation (3.9.10b) then moves us from F to a point G that is 75 percent of the way to the supply curve. The impact of $\omega = 0.75$ is to dampen any change we make. The second iteration of (3.9.10) sends us to J , a point much closer to the equilibrium than our starting point of A . This tells us that the damped Gauss-Seidel process will converge to the equilibrium.

Accelerating Convergence in a Game

Figure 3.4 displayed an application of extrapolation which stabilized an unstable simple iteration. A different example is illustrated in figure 3.5. Here we have the reaction curves of two price-setting duopolists, where firm 2's reaction curve is $p_2 = 2 + 0.80p_1 \equiv R_2(p_1)$ and firm one's reaction curve is $p_1 = 1 + 0.75p_2 \equiv R_1(p_2)$. The equilibrium system is

$$\begin{aligned} p_1 - 0.75p_2 &= 1, \\ -0.80p_1 + p_2 &= 2. \end{aligned} \tag{3.9.11}$$

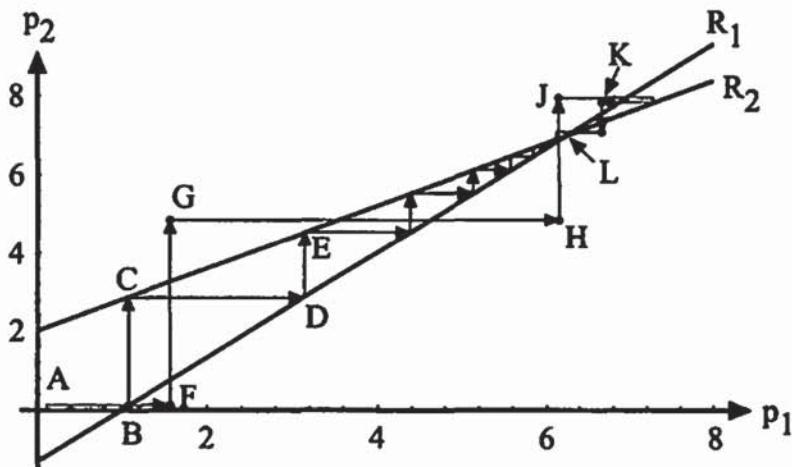


Figure 3.5
Accelerating a Nash equilibrium computation

The system (3.9.11) is diagonally dominant, indicating that both Gauss-Seidel and Gauss-Jacobi will converge. In fact figure 3.5 shows the iterates of Gauss-Seidel applied to (3.9.11), which is the iterative scheme

$$p_1^{n+1} = 1 + 0.75p_2^n, \quad (3.9.12a)$$

$$p_2^{n+1} = 2 + 0.80p_1^{n+1}. \quad (3.9.12b)$$

If we begin (3.9.12) at point *A*, where $(p_1, p_2) = (0.1, 0.1)$, the first iteration moves us first to *B* then *C*, and the second iteration moves us to *D* and then *E*, clearly converging to the equilibrium at $(6.25, 7.0)$.

We will apply (3.9.6) to accelerate (3.9.12). If we choose $\omega = 1.5$, we arrive at the iteration

$$p_1^{n+1} = 1.5(1 + 0.75p_2^n) - 0.5p_1^n, \quad (3.9.13a)$$

$$p_2^{n+1} = 1.5(2 + 0.80p_1^{n+1}) - 0.5p_2^n. \quad (3.9.13b)$$

The difference between (3.9.12) and (3.9.13) is displayed in figure 3.5. In (3.9.12a) we move from the point *A* right to the point *B* on player one's reaction curve, whereas in (3.9.13a) we move 50 percent further, continuing on to *F*. Equation (3.9.13b) then moves us from *F* to a point *G* which is 50 percent beyond player two's reaction curve. The impact of $\omega = 1.5$ is to exaggerate any change we make. The second iteration of (3.9.13) sends us to *J*, a point much closer to the equilibrium than point *E* which is the outcome of two unaccelerated Gauss-Seidel iterations.

However, as we approach the solution, the extrapolated iteration overshoots the solution and must work its way back to the solution. This is illustrated by the fact that from J the accelerated algorithm moves to K which is northeast of the solution. From K the algorithm moves to L , moving back toward the solution. This overshooting occurs because of the exaggeration inherent in $\omega > 1$. Such an extrapolation gets us in the neighborhood of the true solution more quickly, but it may lead to extraneous oscillation. Even with this overshooting, L is a better approximation to the true solution than the unaccelerated Gauss-Seidel iterate after four iterations.

Gauss-Jacobi and Gauss-Seidel schemes and their accelerations are at best linearly convergent; this comment remains true also for the accelerated methods. Acceleration changes the linear rate of convergence but does not change linear convergence to, say, quadratic of convergence. We have not explicitly discussed stopping rules for these iterative schemes. One can implement the rules discussed in Section 2.8. The key fact is that these schemes, unaccelerated and accelerated, are linearly convergent, and therefore we should use rules like (2.8.6) with estimated rates of convergence.

3.10 Calculating A^{-1}

Computing the inverse of a matrix is a costly problem and is therefore avoided whenever possible. For most purposes in numerical methods, we only need $A^{-1}b$ for a few choices of b , in which case it is better to compute $x = A^{-1}b$ by solving the linear system $Ax = b$. This is an important rule: Whenever you see $x = A^{-1}b$, think $Ax = b$ and use a linear equation solution method.

We will sometimes want to compute A^{-1} . This will be true if we need to compute $A^{-1}b$ for more than n choices of b . Repeated use of the LU method is an efficient way to compute A^{-1} . If e^i is the i th column vector of the identity matrix, then the solution to $Ax = e^i$ is the i th column of A^{-1} . Therefore to compute A^{-1} , one factors A into its LU decomposition which is then used to solve n linear equations, a total of $\mathcal{O}(n^3)$ operations.

Decomposition methods will compute A^{-1} to the precision of the machine. Sometimes we are satisfied with a less precise estimate of A^{-1} . In that case there are some iterative schemes that can be used. It is also possible that an iterative scheme converges rapidly and actually dominates LU decomposition. One iterative scheme to calculate matrix inverses is given by Neuman's lemma and follows the idea of operator splitting and iteration.

LEMMA 3.10.1 (Neuman) Let B be a $n \times n$ matrix. If $\rho(B) < 1$, then $(I - B)^{-1}$ exists, and

$$(I - B)^{-1} = \sum_{k=0}^{\infty} B^k.$$

Proof Since $\rho(B) < 1$, all of the eigenvalues of B , λ_i , differ from unity, and the eigenvalues of $I - B$, $1 - \lambda_i$, must be nonzero. Therefore $I - B$ is nonsingular. The identity

$$(I - B)(I + B + B^2 + \cdots + B^k) = I - B^{k+1}$$

implies that

$$I + B + \cdots + B^k = (I - B)^{-1} - (I - B)^{-1}B^{k+1}.$$

Since the modulus of each of the eigenvalues of B is less than unity, B^{k+1} goes to zero as k increases, proving the lemma. ■

Neuman's lemma will be important in understanding some dynamic programming methods. Furthermore it is really much more general. The only property of B that we used in the proof was that it was a linear operator all of whose eigenvalues were less than unity in modulus. If the spectral radius is small, then the convergence is rapid, and good approximations to A^{-1} can be quickly computed.

3.11 Computing Ergodic Distributions

A common step in the numerical analysis of stochastic economic models is the computation of an ergodic distribution and its moments. In this section we discuss methods for approximating ergodic distributions.

Finite Markov Chains

Let $\Pi = (\pi_{ij})$ be an n -state Markov transition matrix. If x is the ergodic distribution,² then $x\Pi = x$, which can also be expressed as the linear system $x(\Pi - I) = 0$. However, this equation is a homogeneous linear equation; therefore, if x is a solution, so is αx for any $\alpha \in R$. The ergodic distribution is fixed by imposing the additional

2. In this section x will be a row vector in order to remain consistent with the notation from probability theory. We then have a problem of the form $xA = b$. This creates no essential problem because it can be transformed into the equivalent problem $A^T x^T = b^T$.

condition $\sum x_i = 1$ which makes x a probability measure. Therefore a direct way to compute the ergodic distribution is to replace the last column of $\Pi - I$ with a vector of ones and solve the resulting linear system:

$$x \begin{pmatrix} \pi_{11} - 1 & \pi_{12} & \cdots & \pi_{1,n-1} & 1 \\ \pi_{21} & \pi_{22} - 1 & \cdots & \pi_{2,n-1} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \pi_{n-1,1} & \pi_{n-1,2} & \cdots & \pi_{n-1,n-1} - 1 & 1 \\ \pi_{n,1} & \pi_{n,2} & \cdots & \pi_{n,n-1} & 1 \end{pmatrix} = (0, \dots, 0, 1). \quad (3.12.1)$$

We see here that the altered last column forces $\sum x_i = 1$, while the other columns enforce the ergodicity conditions.

An alternative procedure is suggested by a basic theorem of Markov chains:

THEOREM 3.11.1 If Π is a Markov transition matrix, then $x = \lim_{k \rightarrow \infty} x^0 \Pi^k$ is an ergodic distribution for any $x^0 \in R^n$. If Π is *irreducible*, that is, for each pair of states (i, j) there is positive probability of going from state i to state j in a finite number of transitions, then x is the unique ergodic distribution.

This suggests that one way to compute an ergodic distribution is to compute the sequence $x^{k+1} = x^k \Pi$ for a large k . The choice between the direct method in (3.12.1), and this iterative method depends on the number of states and how many iterations are necessary for a good approximation. The direct linear equation approach takes $n^3/2$ multiplications. If we start with x^0 , it takes mn^2 operations to compute $x^0 \Pi^m$. Therefore, if the number of iterations necessary to approximate the ergodic distribution is small relative to the number of states, one should iteratively calculate $x^0 \Pi^m$; otherwise, use the direct linear equation approach in (3.12.1).

The following example illustrates the principles. Suppose that an individual earns 0, 1, or 2 per period with equal probability. Suppose that he consumes a quarter of his wealth each period. We will approximate the problem with a Markov chain that takes on 12 wealth values, 1 through 12, inclusive. The transition law is approximated by assuming that consumption is rounded to the nearest integer. To compute the ergodic distribution of wealth, we choose an initial vector, v^1 , and compute the sequence $v^{k+1} = v^k \Pi$. The columns in table 3.3 indicates the probability of being in state i at periods $k = 4, 8, 16, 32$ if initial wealth is 1, an initial condition represented by the probability vector v with $v_1 = 1$ and all other components equaling zero. Note the rapid convergence. In only 16 iterations we have computed a six-digit approximation of the ergodic distribution.

Table 3.3
Successive distributions of a Markov process

State	$k = 4$	$k = 8$	$k = 16$	$k = 32$
1	0.003906	0	0	0
2	0.082031	0.010117	0.008929	0.008929
3	0.140625	0.029119	0.026786	0.026786
4	0.199219	0.074844	0.071429	0.071429
5	0.242187	0.181837	0.178571	0.178571
6	0.167969	0.215218	0.214286	0.214286
7	0.109375	0.212991	0.214286	0.214286
8	0.050781	0.175156	0.178571	0.178571
9	0.003906	0.068163	0.071429	0.071429
10	0	0.024665	0.026786	0.026786
11	0	0.007890	0.008929	0.008929
12	0	0	0	0

Discrete Approximations of Continuous Chains

We will often want to analyze Markov processes on continuous spaces. One way to do that is to discretize a continuous chain. Suppose that a continuous process on a compact interval, $[a, b]$, is described by $F(x|y)$, that is, the probability of jumping to a point in $[a, x]$ from y equals $F(x|y)$.

We will construct an N -state approximation to the continuous process and compute the approximation's ergodic distribution. To do this, divide $[a, b]$ into N equal-sized intervals, with interval i being

$$I_i = [a + (i - 1)h, a + ih], \quad i = 1, \dots, N,$$

where $h = (b - a)/N$. To construct the transition matrix, Π , we could focus on the midpoint of each interval and define

$$\pi_{i,j} = F(a + jh|a + (i - \frac{1}{2})h) - F(a + (j - 1)h|a + (i - \frac{1}{2})h). \quad (3.12.2)$$

In words, the probability of going to state j from state i in the discrete approximation equals the probability of going to interval j from the center of interval i in the continuous chain.

Alternatively, we could focus on the “average” probability; this approximation is defined by

$$\pi_{i,j} = (b - a)^{-1} \int_{I_i} \int_{I_j} dF(x|y) dx dy.$$

Here the probability of going to state j from state i in the approximation equals the probability of going to interval j from the “average” point in interval i in the continuous chain.

3.12 Overidentified Systems

We began the chapter discussing the problem $Ax = b$ where A is square. Sometimes we will want to solve the system $Ax = b$ where A has more rows than columns, implying more linear equations than unknowns; this is an *overidentified* problem. Since there may be no solution to overidentified problems, we need first to redefine the concept of solution.

One common way is to define the solution to be the x that minimizes the Euclidean norm of the residual, $(Ax - b)^\top (Ax - b)$; this is called the *least squares solution*. This approach replaces a system of equations with an optimization problem and is equivalent if A is square. The least squares problem has a unique solution $(A^\top A)^{-1} A^\top b$ if $(A^\top A)^{-1}$ is nonsingular.

3.13 Software

There is much software available for linear algebra. The standard library of programs performing the basic linear operations is called BLAS (basic linear algebra subroutines). Double precision versions are often called DBLAS. LAPACK is a library combining linear algebra and eigenvalue routines. It can be downloaded from netlib, but your local mainframe or network is the most appropriate source, since it may have versions tuned to the hardware. To do the exercises in future chapters, one needs programs for matrix and vector arithmetic, matrix inversion, LU and Cholesky decomposition, computing determinants, solving linear systems, and computing eigenvalues and eigenvectors of general matrices. There are several ways to represent sparse matrices. The scheme described in the text is a simple and direct method, but not the most efficient approach. Some of the more popular approaches are the Yale Sparse Matrix Package and PCGPACK.

Software like Matlab and Gauss are very good at doing linear algebra and are easy to use. Matlab offers easy-to-use sparse matrix capabilities. If one is doing only linear algebra, these are as good as Fortran or C and easier to use. However, such languages are not as competitive if one also needs to do nonlinear procedures that cannot be vectorized.

3.14 Summary and Further Reading

This chapter has discussed basic solution methods for linear equations and introduced several basic ideas which are used in many numerical analysis. The basic linear solution methods, such as *LU* and Cholesky decomposition, are used in many numerical procedures. The performance of linear and nonlinear numerical procedures depends on the condition number of the critical matrices; therefore the conditioning of a matrix is one of the most important concepts in numerical mathematics.

We introduced iteration ideas and applied them to linear problems. The key features are the construction of iterative schemes applying Gauss-Jacobi or Gauss-Seidel substitution schemes, and developing stabilization and acceleration ideas. Young (1971) and Hageman and Young (1981) discuss iterative procedures for linear equations. Many of these ideas are intuitive for economists, since they correspond to basic ideas in economic dynamics, such as hog cycles and strategic stability. The iteration ideas introduced here for linear problems are applicable to nonlinear problems as well.

There are several excellent monographs on numerical linear algebra, including Golub and van Loan (1983). Most basic numerical analysis texts contain extensive formal developments of the topics covered in this chapter; in particular, see Stoer and Burlisch (1980).

Exercises

1. Write programs for *LU* decomposition (without pivoting) and Cholesky decomposition. Using these programs, compute the following:

- a. The *LU* decomposition of

$$A = \begin{pmatrix} 54 & 14 & -11 & 2 \\ 14 & 50 & -4 & 29 \\ -11 & -4 & 55 & 22 \\ 2 & 29 & 22 & 95 \end{pmatrix}.$$

- b. The *LU* decomposition of

$$B = \begin{pmatrix} 54 & -9 & 18 & 9 \\ 6 & 23 & -2 & 9 \\ -12 & 14 & 34 & -3 \\ -6 & 13 & 20 & 49 \end{pmatrix}.$$

- c. The Cholesky decomposition of A .

2. Solve $Ax = b$ for

$$A = \begin{pmatrix} 54 & 14 & -11 & 2 \\ 14 & 50 & -4 & 29 \\ -11 & -4 & 55 & 22 \\ 2 & 29 & 22 & 95 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix},$$

by (a) LU decomposition, (b) Cholesky decomposition, (c) Gauss-Jacobi, (d) Gauss-Seidel, (e) successive overrelaxation, (f) block Gauss-Jacobi (two blocks of 2×2 systems), (g) block Gauss-Seidel (two blocks of 2×2 systems), and (h) Cramer's rule.

3. Define A_n to be the $n \times n$ matrix

$$A_n = \begin{pmatrix} 2 & 1 & 0 & \cdots & 0 \\ 1 & 2 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & 2 \end{pmatrix}.$$

The unique solution to $A_n x = 0$ is $x = 0_n$.

- a. Solve $A_n x = 0$ for $n = 5, 10, 20$, and 50 using Gauss-Jacobi, Gauss-Seidel, and successive overrelaxation with $\omega = (0.1)l$, $l = 1, \dots, 20$, and initial guess $x_i^0 = 1$, $i = 1, \dots, n$.
 - b. How many iterations are required for each method to get the error (defined to be the maximum error among the n components of x) under 0.01 ? under 0.001 ? under 0.0001 ?
 - c. Plot the spectral radius of the $M_\omega^{-1}N_\omega$ matrix in successive overrelaxation (see 3.9.7) as a function of ω .
 - d. Which value of ω was best? How much better was the best ω relative to $\omega = 1$?
4. Assume that $x \in R$ follows the process $x_{t+1} = \rho x_t + \varepsilon_{t+1}$ where $\rho < 1$ and that the ε_t are i.i.d. and uniformly distributed on $[-1, 1]$. Write a n -state approximation for the Markov transition matrix for this process, $n = 5, 10, 25, 100$, and compute the ergodic distribution for $\rho = 0.8, 0.5, 0.2$. Compare the two methods discussed in the text for discretizing a continuous-state Markov chain.
5. Use sparse matrix methods to model the ergodic distribution of wealth where an individual's wealth fluctuates because of i.i.d. wage shocks and expenditures. Assume that there are N possible wealth states and that π is the transition matrix where

$$\pi_{ij} = \text{Prob}\{\text{tomorrow's wealth is } w_j \mid \text{current wealth is } w_i\}$$

Assume there are $N = 1,001$ wealth states. Assume that in each state there is a small probability, 0.01 , of losing all wealth and that wealth stays constant or goes up or down by 10 states. Specifically, if $i < 10$, then $\pi_{i,0} = 0.34$, and $\pi_{i,i} = \pi_{i,i+10} = 0.33$; if $10 \leq i \leq 990$, wealth is equally likely to move to w_{i-10} , w_i , and w_{i+10} ; and if $i > 990$, then $\pi_{i,i-10} = \pi_{i,i} = 0.33$, and $\pi_{i,N} = 0.33$. In this case each row of π has three or four nonzero elements.

- a. Compute the ergodic distribution of wealth.
- b. Compute the mean, variance, and skewness of the ergodic distribution.

c. Repeat this exercise for the $N = 10^2$, 10^3 , and 10^4 discretizations of the continuous process $w_{t+1} = 1.03w_t - 0.001w_t^2 + \varepsilon_t$ where the $\varepsilon \sim U[0, 1]$.

6. Suppose that demand for good i is

$$d_i(p) = a_i \sum_{j \neq i} p_j - b_i p_i + c_i, \quad i = 1, \dots, n,$$

where $a_i > b_i > 0$, and that supply is

$$s_i(p) = A_i + B_i p_i, \quad i = 1, \dots, n,$$

where $B_i > 0$. Write a program to solve for equilibrium with sensible choices of a_i, b_i, A_i , and B_i . That means read in the parameters, check that they are consistent with commonsense economics (downward sloping demand curves, increasing supply curves, and concave utility), and output the equilibrium prices and outputs.

7. Suppose that three Cournot duopolists have the reaction curves

$$q_1 = 5 - 0.5q_2 - 0.3q_3,$$

$$q_2 = 7 - 0.6q_1 - 0.1q_3,$$

$$q_3 = 4 - 0.2q_1 - 0.4q_2.$$

Compute equilibrium using fixed-point iteration and extrapolation. What is the optimal extrapolation factor?

8. Let A be a matrix of zeros except that $a_{ii} = 2$ for odd i , $a_{ii} = 3$ for even i , and $a_{i,i-1} = a_{i,i+1} = 1$ for $i = 1, 2, \dots, 1000$. Let $b_i = 1001 - i$, $i = 1, 2, \dots, 1000$. Solve $Ax = b$.

4 Optimization

The chief behavioral assumption in economics is that agents optimize. Economists assume that firms minimize costs and maximize profits, consumers maximize utility, players maximize payoffs, and social planners maximize social welfare. Econometricians also use optimization methods in least squares, method of moments, and maximum likelihood estimation procedures. These considerations make optimization methods a central numerical technique in economic analysis.

The most general optimization problem minimizes¹ an objective function subject to equality and inequality constraints:

$$\begin{aligned} \min_x f(x) \\ \text{s.t. } g(x) = 0, \\ h(x) \leq 0, \end{aligned}$$

where $f: R^n \rightarrow R$ is the *objective function*, $g: R^n \rightarrow R^m$ is the vector of m *equality constraints*, and $h: R^n \rightarrow R^l$ is the vector of l *inequality constraints*. Examples of such problems include relatively simple problems such as the maximization of consumer utility subject to a linear budget constraint, as well as the more complex problems from principal-agent theory. We generally assume that f , g , and h are continuous, although that will not always be necessary.

All numerical optimization methods search through the space of feasible choices, generating a sequence of guesses that should converge to the true solution. Methods differ in the kind of information they use. The simplest methods compute the objective and constraint functions at several points and pick the feasible point yielding the largest value. These comparison methods need no derivative information. A second class of methods uses the gradients of the objective and constraints in their search for better points. A third class of methods uses both gradient and curvature information about the objective and constraints.

This chapter will examine algorithms of all types, since they have different strengths and all have uses in economic problems. Comparison methods require no smoothness conditions on the objective or constraints, but they are generally slow to converge. Comparison methods are suitable for problems with kinks and some kinds of discontinuities. Gradient methods require differentiability of the objective and constraints. They also use little space, an important consideration when the number of variables is large. Methods that use curvature information require even more

1. We examine minimization problems, since maximizing f is the same as minimizing $-f$. Also most optimization software is geared for minimization, so we need to get used to transforming maximization problems into minimization problems.

smoothness of the objective and constraints, but they may converge more rapidly to the solution. However, curvature-based methods are generally confined to problems with only a moderate number of variables because of the high cost of computing and storing Hessians.

The chapter first examines one-dimensional unconstrained optimization problems, then multidimensional unconstrained optimization problems, and finally general multidimensional constrained problems. Nonlinear least squares problems are common in economics and have special algorithms which are discussed here. The chapter finishes with applications of optimization to the computation of Nash equilibria, portfolio problems, dynamic life-cycle consumption profiles, and allocation problems affected by incentive compatibility constraints.

4.1 One-Dimensional Minimization

The simplest optimization problem is the scalar unconstrained problem:

$$\min_{x \in R} f(x),$$

where $f: R \rightarrow R$. We first focus on one-dimensional problems for two reasons. First, they illustrate basic techniques in a clear fashion. Second, one-dimensional methods are particularly important since many multivariate methods reduce to solving a sequence of one-dimensional problems. Efficient and reliable methods for one-dimensional problems are essential ingredients of efficient multivariate algorithms.

Bracketing Method

The simplest one-dimensional optimization scheme is a comparison method called the *bracketing algorithm*. Suppose that we have found points a , b , and c such that

$$\begin{aligned} a < b < c, \\ f(a), f(c) > f(b). \end{aligned} \tag{4.1.1}$$

Figure 4.1 illustrates such a case, and algorithm 4.1 outlines the bracketing method. These conditions imply that a local minimum exists somewhere in $[a, c]$; that is, we have bracketed a minimum. We now need to find a smaller interval that brackets some minimum in $[a, c]$. To do this, we pick a fourth point $d \in (a, c)$. Suppose that $d \in (a, b]$. Then, if $f(d) > f(b)$, we know that there is some minimum in $[d, c]$. If $f(d) < f(b)$, as in figure 4.1, then there is some minimum in $[a, b]$. If $d \in [b, c)$, we similarly would find a smaller interval that brackets a minimum. In any case we have

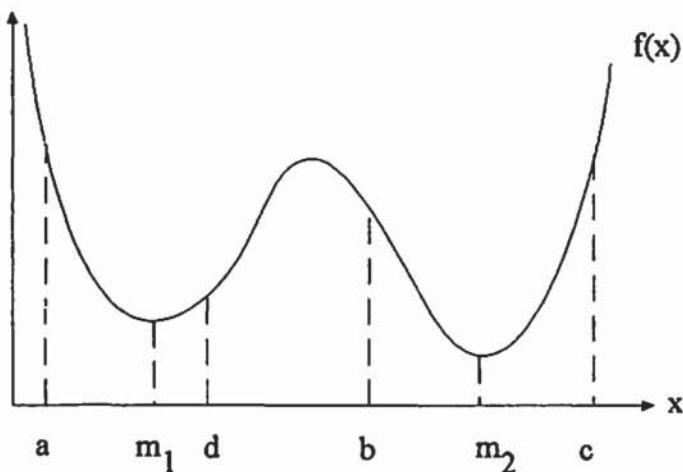


Figure 4.1
Bracketing method

bracketed a minima with a smaller interval. With the new triple, we can repeat this step again, and bracket a minimum with an even smaller interval.

Algorithm 4.1 Bracketing Algorithm

Initialization. Find $a < b < c$ such that $f(a), f(c) > f(b)$. Choose a stopping criterion ε .

Step 1. Choose d : If $b - a < c - b$, then set $d = (b + c)/2$; otherwise, $d = (a + b)/2$.

Step 2. Compute $f(d)$.

Step 3. Choose new (a, b, c) triple: If $d < b$ and $f(d) > f(b)$, then replace (a, b, c) with (d, b, c) . If $d < b$ and $f(d) < f(b)$, then replace (a, b, c) with (a, d, b) . If $d > b$ and $f(d) < f(b)$, then replace (a, b, c) with (b, d, c) . Otherwise, replace (a, b, c) with (a, b, d) .

Step 4. Stopping criterion: If $c - a < \varepsilon$, STOP. Otherwise, go to step 1.

Before we begin a comparison procedure, we must find points a , b , and c that satisfy the conditions in (4.1.1). This is generally done by picking some point x_0 , constant $\alpha > 1$, and step Δ , computing $f(x_0), f(x_0 \pm \alpha\Delta), f(x_0 \pm \alpha^2\Delta), \dots$, until we get a triple satisfying (4.1.1).

The stopping criterion for the comparison method is quite simple. Since the bracketing interval is (a, c) at each iteration, we just compare $c - a$ to some maximum permissible value ε . We want to choose ε so that bracketing the minimum in (a, c) is “practically” the same as knowing the true minimum, but given the slow speed of the method, ε should not be smaller than necessary.

The comparison method will be slow, but it makes few requirements of the objective function. In particular, it will work for any continuous, bounded function on a finite interval, no matter how jagged it may be. One must keep in mind, however, that the only guarantee is that the comparison method will find a local minimum. In figure 4.1 the comparison method starting with $f(x)$ at a , b , and c will converge to the local minimum at m_1 instead of the global minimum at m_2 . This problem of converging to local extrema instead of global extrema will arise with any optimization method. If we know that the problem has only one solution, say, because the objective is convex, then this problem will not arise.

Newton's Method

For C^2 functions $f(x)$, Newton's method is often used. The idea behind Newton's method is to start at a point a and find the quadratic polynomial, $p(x)$, that approximates $f(x)$ at a to the second degree, implying that

$$p(x) \equiv f(a) + f'(a)(x - a) + \frac{f''(a)}{2}(x - a)^2.$$

If $f''(a) > 0$, then p is convex. We next approximately minimize f by finding the point x_m which minimizes $p(x)$. The minimizing value is $x_m = a - f'(a)/f''(a)$. If $p(x)$ is a good global approximation to $f(x)$, then x_m is close to the minimum of $f(x)$. In any case the hope is that x_m is closer than a to the minimum. Furthermore we hope that if we repeat this step, we will get even closer.

These considerations motivate *Newton's method*:²

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}. \quad (4.1.2)$$

Note that Newton's method does not care if f is concave or convex. Newton's method is really trying to find critical points, that is, solutions to $f'(x) = 0$. If Newton's method converges to x^* , we must check $f''(x^*)$ to determine if x^* is a local minimum or a local maximum.

The problems with Newton's method are several. First, it may not converge. Second, $f''(x)$ may be difficult to calculate. The good news is that Newton's method converges rapidly if we have a good initial guess.

THEOREM 4.1.1 Suppose that $f(x)$ is minimized at x^* , C^3 in a neighborhood of x^* , and that $f''(x^*) \neq 0$. Then there is some $\varepsilon > 0$ such that if $|x_0 - x^*| < \varepsilon$, then the x_n

2. Actually a more proper name is the Newton-Raphson-Simpson method, but this text uses the term Newton's method. See Ypma (1995) and Kollerstrom (1992) for intellectual histories of Newton's method.

sequence defined in (4.1.2) converges quadratically to x^* ; in particular,

$$\lim_{n \rightarrow \infty} \frac{|x_{n+1} - x^*|}{|x_n - x^*|^2} = \frac{1}{2} \left| \frac{f'''(x^*)}{f''(x^*)} \right| \quad (4.1.3)$$

is the quadratic rate of convergence.

Proof Since f is C^3 , $f'(x^*) = 0 \neq f''(x^*)$ and f' is C^2 . Taylor's theorem says that

$$f'(x) = f'(x_n) + f''(x_n)(x - x_n) + \frac{1}{2}f'''(\eta_n(x))(x - x_n)^2 \quad (4.1.4)$$

for some function $\eta_n(x) \in [x_n, x] \cup [x, x_n]$. Given (4.1.2) and $f'(x^*) = 0$, the Taylor expansion (4.1.4) at $x = x^*$ implies that

$$|x_{n+1} - x^*| = \frac{1}{2} \left| \frac{f'''(\eta_n(x^*)) (x^* - x_n)^2}{f''(x_n)} \right|. \quad (4.1.5)$$

For any $\beta \in (0, 1)$, if $|x_m - x^*|$ is sufficiently small, (4.1.5) shows that $|x_{n+1} - x^*| < \beta|x_n - x^*|$ for $n > m$, which implies $x_n \rightarrow x^*$. Furthermore, along a convergent sequence, (4.1.5) implies (4.1.3), which is finite if $f''(x^*) \neq 0$. ■

This rapid quadratic convergence makes Newton's method particularly powerful. The weaknesses are also significant but often manageable in practice, as we will see later.

Stopping Rule

Even if Newton's method would converge to the true minimum, that is only an asymptotic property. We still need to decide when to stop the iteration. We will want to stop the iteration when $x_k - x^*$ is “unimportant.” Even after choosing an acceptable error, ε , it is unclear how to be sure that some x_k is within ε of x^* . Unlike the comparison method the iterates of Newton's method do not bracket a minimum.

In the case of a one-dimensional problem, we could switch to a comparison method once we think Newton's method is close to a solution. That is, once $|x_k - x_{k+1}| < \varepsilon$, then we could begin the comparison method with some interval $[a, b]$ containing $[x_k, x_{k+1}]$ satisfying conditions (4.1.1), thereby bracketing a minimum.

Since $f'(x^*) = 0$, we are inclined to stop whenever $f'(x_k)$ is small; however, the curvature $f''(x^*)$ indicates the wisdom of this. If $f''(x^*)$ is close to zero, there are many points where $f'(x)$ is nearly zero, some not near x^* ; hence it is not sufficient to stop when $f'(x_k)$ is nearly zero. Therefore the standard stopping condition requires that $f'(x_k)$ is nearly zero, and that the last two iterates are very close.

When we combine the Newton iteration (4.1.2) with these stopping tests, we arrive at Newton's method:

Algorithm 4.2 Newton's Method in R^1

Initialization. Choose initial guess x_0 and stopping parameters $\delta, \varepsilon > 0$:

Step 1. $x_{k+1} = x_k - f'(x_k)/f''(x_k)$.

Step 2. If $|x_k - x_{k+1}| < \varepsilon(1 + |x_k|)$ and $|f'(x_k)| < \delta$, STOP and report success; else go to step 1.

Since Newton's method is quadratically convergent, the stopping rule in step 2 will produce a solution that has an error of order ε unless f is nearly flat and/or has rapidly changing curvature in a large neighborhood of the solution. We can even test for these problems by computing f'' at some points near the candidate solution.

Consumer Budget Example

To illustrate these three methods, we will consider a utility maximization problem. Suppose that a consumer has \$1 to spend on goods x and y , the price of x is \$2, the price of y is \$3, and his utility function is $x^{1/2} + 2y^{1/2}$. Let θ be the amount spent on x with the remainder, $1 - \theta$, spent on y . The consumer choice problem then reduces to

$$\max_{\theta} \left(\frac{\theta}{2}\right)^{1/2} + 2\left(\frac{1-\theta}{3}\right)^{1/2} \quad (4.1.6)$$

which has the solution $\theta^* = 3/11 = 0.272727\dots$

To apply the comparison method, we need to bracket the optimal θ . From the form of the utility function, it is clear that the optimal θ lies between 0 and 1. Therefore we let $a = 0$, $b = 0.5$, and $c = 1.0$ initially, and generate a sequence of (a, b, c) triples, where at each iterate b is the minimal value of θ thus far tested. The new minimum points form the sequence

0.5, 0.5, 0.25, 0.25, 0.25, 0.25, 0.281, 0.281, 0.266, 0.2737.

The apparent rate of convergence is slow as predicted.

We next examine Newton's method. If $\theta_0 = \frac{1}{2}$ is the initial guess, then the Newton iteration applied to (4.1.6) yields the sequence

0.2595917942, 0.2724249335, 0.2727271048, 0.2727272727,

which we see is rapidly converging to the solution. The first iterate has only one correct digit, the second has three, the third has six, and the fourth ten (at least). The convergence is clearly quadratic.

We see here that each method will converge to the true solution. However, the Newton method is far faster once we get close.

Importance of Scaling

We will use the consumer example to illustrate what can happen if we use excessively small or large numbers; the resulting difficulties are called *scaling* problems. We repeat the consumer problem except this time we use the utility function $-e^{-x} - e^{-y}$ and with income \$10,000. In terms of θ , the objective becomes $-e^{-10,000\theta/2} - e^{-10,000(1-\theta)/3}$. On most machines this problem leads to severe difficulties and cannot be solved. The problem is that for any value of θ either $e^{-10,000\theta/2}$ or $e^{-10,000(1-\theta)/3}$ (or both) will cause underflow, leading to the computer stopping or, even worse, continuing by assigning a value of zero to the term.

What is to be done in cases such as this? In this case we must use a different approach. The first-order conditions imply the budget constraint $2x + 3y = 10000$ and the marginal rate of substitution condition $y - x = \log(\frac{2}{3})$, a pair of equations that is easy to solve numerically. In general, we must choose units and/or formulate the problem to minimize the likelihood of overflow and underflow.

Global Optimization

Our optimization methods are designed to find a local minimum of a function; there is no assurance that they will find the global minimum. If one starts a Newton iteration from a point sufficiently close to the global minimum, then theorem 4.1.1 says that Newton's method will converge to the global minimum, but any of these methods may stop at a local minimum different from the global minimum.

There are no good ways to find the global optimum. If we know that a function is convex (concave), then any local minimum (maximum) is also the unique global minimum (maximum). Outside of such special cases, it is difficult to be sure that one has a global minimum. These difficulties arise frequently in economics. Likelihood functions, for example, often have multiple local maxima, making life difficult for econometricians using maximum likelihood estimation. In practice, one deals with this problem by restarting the optimization method with several different initial guesses, taking the best of the resulting local optima as the global optimum. It is easy to be lazy and try just, say, ten restarts. Experience indicates that one should do as many restarts as is feasible. In any case, global optimization problems are difficult outside of special cases, and one must develop careful heuristics to reduce the chance of gross error.

4.2 Multidimensional Optimization: Comparison Methods

Most interesting optimization problems in economics involves several variables. The generic unconstrained problem is

$$\min_x f(x) \quad (4.2.1)$$

for $f: R^n \rightarrow R^n$. For convenience, we assume that f is continuous although some of the methods we discuss will work more generally. To solve (4.2.1), we must develop methods more sophisticated than the one-dimensional case. This section discusses the most basic methods applicable to all problems.

Grid Search

The most primitive procedure to minimize a function is to specify a grid of points, say, between 100 and 1000 points, evaluate $f(x)$ at these points, and pick the minimum. We are often impatient and want to immediately use more sophisticated methods, but many practical considerations argue for first doing a grid search.

Even though we would seldom stop at this point, the results of this preliminary search will often be very useful. For example, if we are trying to maximize a likelihood function, the result of these calculations may indicate the general curvature of the likelihood function. If the grid search indicates that the function is flat over a wide range, there is little reason to proceed with more sophisticated methods. If the grid search indicates that there are multiple local optima, then we will need to work hard to find the global optimum. Also, if the Hessian at the best point on the grid is poorly conditioned, then it is unlikely that the more sophisticated methods discussed below can be effective. The grid search should produce good initial guesses for better methods.

If we know that the objective is smooth with well-conditioned curvature and unique local optimum, this preliminary step is less useful. However, outside of such special cases, we need the information provided by a grid search. Even for well-behaved cases, a grid search will at least provide a test of whether we have correctly coded the objective function. Despite the natural inclination to try supposedly better methods first, it is always a good idea to begin with a grid search for any problem.

Polytope Methods

We first examine *polytope* methods, which take a simple but robust approach to multivariate optimization. Polytope methods are multidimensional comparison methods. They require no smoothness conditions on the objective. They are easy to program and are reliable, but slow.

The general idea is to think of someone going down a hill. The method first constructs a simplex in R^n , which we will denote by its set of vertices, $\{x^1, \dots, x^{n+1}\}$. The simplex should have full dimension, that is, the $n + 1$ vertices should not lie on a

$(n - 1)$ -dimensional hyperplane. One simple initial choice for this set is $(0, 0, \dots, 0)$ plus all n points of the form $(0, \dots, 0, 1, 0, \dots, 0)$. In each iteration we throw out a vertex with a high objective value and replace it with one of lower value, converging ultimately to a minimum.

Specifically, we begin by numbering the simplex's vertices so that

$$f(x^i) \geq f(x^{i+1}), \quad i = 1, \dots, n. \quad (4.2.2)$$

We want to replace the current simplex with another resting on lower points. Since f is greatest at x^1 , we try to find some point x to replace x^1 in the simplex. Since f is large at x^1 , there is a good chance that f declines along the ray from x^1 to the opposing face. Hence our first guess, y^1 , will be the reflection of x^1 through the opposing face. If $f(y^1) < f(x^1)$, then we have succeeded in finding a suitable replacement for x^1 , whereupon we drop x^1 from our simplex, add y^1 , reorder according to (4.2.2), and again try to find a replacement for the vertex at which f is greatest.

If $f(y^1) \geq f(x^1)$, we then need to check other points. The simple polytope method turns to x^2 and attempts to replace x^2 with its reflection, y^2 . We continue this until we find some x^i and its reflection y^i such that $f(y^i) < f(x^i)$. If we succeed in finding such a x^i , we make the replacement, renumber the vertices so that (4.2.2) holds, and begin again trying to replace one of the vertices with its reflection. Sometimes we may not find any such reflection. This will happen when the simplex is too large for the reflection process to work. In this case we shrink (say by half) the simplex toward x^{n+1} , the vertex where f is smallest. We renumber the resulting simplex, and begin again the replacement process.

Algorithm 4.3 Polytope Algorithm

Initialization. Choose the stopping rule parameter ε . Choose an initial simplex $\{x^1, x^2, \dots, x^{n+1}\}$.

Step 1. Reorder the simplex vertices so that $f(x^i) \geq f(x^{i+1})$, $i = 1, \dots, n$.

Step 2. Find the least i such that $f(x^i) > f(y^i)$ where y^i is the reflection of x^i . If such an i exists, set $x^i = y^i$, and go to step 1. Otherwise, go to step 3.

Step 3. Stopping rule: If the width of the current simplex is less than ε , STOP. Otherwise, go to step 4.

Step 4. Shrink simplex: For $i = 1, 2, \dots, n$ set $x^i = \frac{1}{2}(x^i + x^{n+1})$, and go to step 1.

Figure 4.2 gives a typical example of such a simplex in R^2 , and illustrates a typical iteration. Suppose that the initial simplex is ABC with $f(A) > f(B) > f(C)$. In this

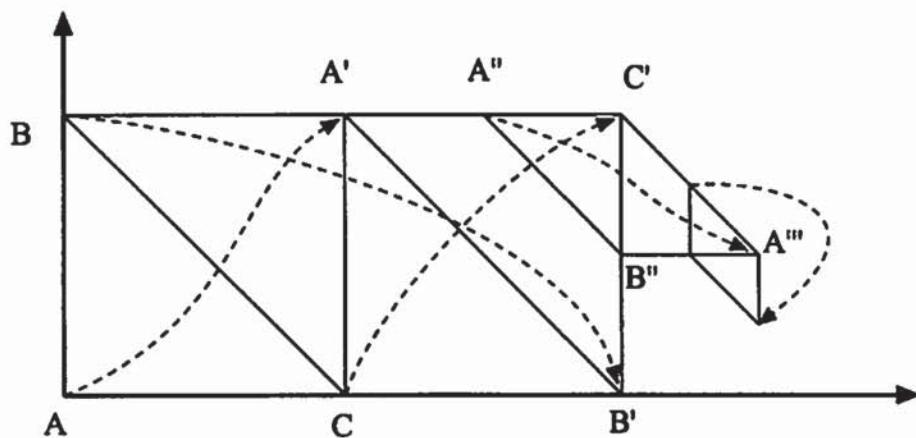


Figure 4.2
Polytope method

example, let $f(A') < f(A)$ where A' is the reflection of A with respect to the simplex ABC ; then the first step replaces A with A' to produce the simplex $BA'C$. Suppose that simple comparisons of points with their reflections next replaces B with its reflection relative to $BA'C$, B' , to produce $A'B'C$, and then replaces C with C' to arrive at $A'B'C'$. These simple comparisons will generally produce some progress.

However, simple reflection comparisons will eventually come to a halt. Suppose that there is no suitable reflection for $A'B'C'$ but that the lowest value is at C' . The polytope algorithm then replaces $A'B'C'$ with $A''B''C'$ by shrinking A' and B' toward C' . We then attempt the reflection step with $A''B''C'$, arriving at, say $A'''B'''C'$, and continue from there with simple reflection comparisons. We continue to alternate between reflection replacements and shrinking steps.

Polytope methods will find a minimum of a continuous function because they have an important property: They always move downhill by a nontrivial amount. They also can handle discontinuous objectives. However, they are slow. For smooth objectives, far faster methods are available.

If we let the polytope algorithm proceed forever, the simplex would converge to a local minima for any continuous function $f(x)$. In real life we must derive a criterion that causes the iterations to end in finite time and produce a good approximation to a minimum. We stop when we come to a point where all reflections produce higher values of f and the simplex is smaller than some critical size. However, we must realize that there is no guarantee that a minimum is inside the final simplex, or even near it. This is a major difference between one-dimensional and multidimensional problems; we can bracket minima in one dimension but not in many dimensions.

4.3 Newton's Method for Multivariate Problems

While comparison algorithms for (4.2.1) are generally useful because they rely on little information about the objective, they do not take advantage of extra information when it is available. Objective functions, such as utility functions and profit functions, are often smooth with many derivatives. We can construct superior algorithms by taking advantage of derivatives when they are available. We denote

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \dots, \frac{\partial f}{\partial x_n}(x) \right), \quad H(x) = \left(\frac{\partial^2 f}{\partial x_i \partial x_j}(x) \right)_{i,j=1}^n$$

to be the gradient and Hessian of f at x . Note that x is a column vector and that the gradient is a row vector.

Newton's method is a simple approach that often yields rapid solutions to C^2 problems. Consider the special case of quadratic minimization. If $f(x) = \frac{1}{2}x^\top Ax + b^\top x$ where A is symmetric positive definite, the minimum is at $x^* = -A^{-1}b$. If A is not positive definite, x^* is still a critical point of $f(x)$.

If f is convex but not quadratic, we generally cannot directly solve for its minimum. However, we can replace f locally with a quadratic approximation and solve for the minimum of the approximation. In Newton's method we examine a sequence of points, x^k , where at each iteration we replace $f(x)$ with a quadratic approximation of f based at x^k and choose x^{k+1} to be the critical point of the local approximation. At x^k that quadratic approximation is $f(x) \doteq f(x^k) + \nabla f(x^k)(x - x^k) + \frac{1}{2}(x - x^k)^\top H(x^k)(x - x^k)$. If f is convex, then $H(x^k)$ is positive definite, and the approximation has a minimum at $x^k - H(x^k)^{-1}\nabla(f(x^k))^\top$. Newton's method is the iterative scheme

$$x^{k+1} = x^k - H(x^k)^{-1}(\nabla f(x^k))^\top. \quad (4.3.1)$$

While the minimization logic does not apply, we use (4.3.1) even when $H(x^k)$ is not positive definite. This is a weak point of Newton's method, but it is addressed in the refinements below.

The performance of the multivariate Newton method is similar to the one-dimensional version, as stated in theorem 4.3.1.

THEOREM 4.3.1 Suppose that $f(x)$ is C^3 , minimized at x^* , and that $H(x^*)$ is non-singular. Then there is some $\varepsilon > 0$ such that if $\|x^0 - x^*\| < \varepsilon$, then the sequence defined in (4.3.1) converges quadratically to x^* .

Proof See Luenberger (1984).

The multivariate Newton method can be expensive because computing and storing the Hessian, $H(x)$, takes $\mathcal{O}(n^3)$ time to compute and occupies $\mathcal{O}(n^2)$ space. In practice, the inversion step in (4.3.1) should never be executed. The critical fact is that the Newton step $s^k \equiv x^{k+1} - x^k$ is the solution to the linear system $H(x^k)s^k = -(\nabla f(x^k))^\top$. Therefore the actual procedure is to compute $H(x^k)$, solve the linear problem $H(x^k)s^k = -(\nabla f(x^k))^\top$ for s^k , and set $x^{k+1} = x^k + s^k$.

Theorem 4.3.1 states that convergence is quadratic for initial guesses sufficiently close to a local minimum. The multivariate Newton method is similar to the one-dimensional version also in terms of its problems. In particular, the multivariate Newton method is not globally convergent. The convergence problem is not surprising since Newton's method only uses the first-order optimization conditions. In particular, there is no assurance that the successive Hessians are positive definite; Newton's method could be generating iterates along which f is increasing. Even when the Hessians are positive definite, there is no assurance that the sequence x^k is moving downhill. Future sections will introduce methods that surely go downhill and get us to a minimum more reliably.

Stopping Rules

Even when Newton's method converges, we need a way to stop the process when it is close to a minimum. In the multivariate case we cannot resort to any bracketing method. We therefore must make a decision based on x^k , $f(x^k)$, $\nabla f(x^k)$, and $H(x^k)$. As in the one-dimension case, we typically use a twofold convergence test. The first thing is to determine if the sequence has converged. The convergence test is of the form

$$\|x^{k+1} - x^k\| < \varepsilon(1 + \|x^k\|). \quad (4.3.2)$$

Second, if the convergence test is satisfied, we determine if the latest iterate satisfies the first-order condition. The optimality test focuses on whether $\nabla f(x^k)$ is zero and takes the form

$$\|\nabla f(x^k)\| \leq \delta(1 + |f(x^k)|). \quad (4.3.3)$$

Again we add the 1 to $|f(x^k)|$ to deal with the case of $f(x^k) \approx 0$. If both (4.3.2) and (4.3.3) are satisfied, we stop and report success. Sometimes we may satisfy the convergence test (4.3.2) but not optimality, (4.3.3). This says that the sequence is changing slowly but not yet near an optimum, indicating that the quadratic appro-

ximations are not good locally. If this happens, we can restart with a smaller ε or change to another method.

The choices of ε and δ are tolerances that users can specify in most minimization software. The primary consideration in choosing δ is the accuracy with which one computes ∇f . One should never ask for tolerances smaller than the machine ε , and generally should use tolerances on the order of the square root of the machine ε . Also larger tolerances should be used if $f(x)$ is computed with error greater than the machine ε ; generally, if f is computed with accuracy ε_f , then the ε and δ used in the stopping rules should be between $\varepsilon_f^{1/4}$ and $\varepsilon_f^{1/2}$. However, the best choices are problem dependent, and users should try several alternatives to learn the trade-off between the higher speed of large tolerances and the accuracy of smaller tolerances. Good software packages give users guidelines concerning these tolerances. Algorithm 4.4 summarizes Newton's method for R^n .

Algorithm 4.4 Newton's Method in R^n

Initialization. Choose initial guess x^0 and stopping parameters δ and $\varepsilon > 0$.

Step 1. Compute the Hessian, $H(x^k)$, and gradient, $\nabla f(x^k)$, and solve $H(x^k)s^k = -(\nabla f(x^k))^T$ for the step s^k

Step 2. $x^{k+1} = x^k + s^k$.

Step 3. If $\|x^k - x^{k+1}\| < \varepsilon(1 + \|x^k\|)$, go to step 4; else go to step 1.

Step 4. If $\|\nabla f(x^k)\| < \delta(1 + |f(x^k)|)$, STOP and report success; else STOP and report convergence to nonoptimal point.

The key pieces of Newton's method are the Hessian and gradient computations. Since step 1 involves solving a linear system of equations, Newton's method inherits all the problems of linear systems. In particular, Newton's method will work well only if the Hessian is well-conditioned. Poorly conditioned Hessians will lead to large errors when computing s^k in step 1. If practical, one should estimate the condition number of $H(x)$ at the purported solution before accepting it. Of course we should also check that $H(x)$ satisfies second-order conditions at the purported solution.

Monopoly Example

To illustrate multivariate optimization algorithms, we use a simple monopoly problem. Suppose that a firm produces two products, Y and Z . Suppose that the demand for Y and Z is derived from the utility function

$$U(Y, Z) = (Y^\alpha + Z^\alpha)^{\eta/\alpha} + M$$

$$\equiv u(Y, Z) + M,$$

where M is the dollar expenditure on other goods. We assume that $\alpha = 0.98$ and $\eta = 0.85$. If the firm produces quantities Y and Z of the goods, their prices will be u_Y and u_Z . Hence the monopoly problem is

$$\max_{Y, Z} \Pi(Y, Z) \equiv Yu_Y(Y, Z) + Zu_Z(Y, Z) - C_Y(Y) - C_Z(Z), \quad (4.3.4)$$

where $C_Y(Y)$ and $C_Z(Z)$ are the production costs of Y and Z . We assume that $C_Y(Y) = 0.62Y$ and $C_Z(Z) = 0.60Z$.

We immediately see a problem that can arise with any optimization method. In its search for the optimum, an algorithm may attempt to evaluate the profit function, (4.3.4), at a point where Y or Z is negative. Such attempts would cause most programs to crash, since fractional powers of negative numbers are generally complex. We need to make some change to avoid this problem. It is possible to perform all arithmetic as complex numbers, but that is generally impractical. The approach we take here is to restate the problem in terms of $y \equiv \ln Y$ and $z \equiv \ln Z$. When we make this substitution into (4.3.4), we find that the resulting problem is

$$\max_{y, z} \pi(y, z), \quad (4.3.5)$$

where $\pi(y, z) \equiv \Pi(e^y, e^z)$; now Π is never evaluated at a negative argument. The system (4.3.5) will serve as our test problem, and has the solution $y^* = -0.58$ and $z^* = 1.08$.

We now discuss how Newton's method does in solving our test monopoly problem, (4.3.5). We will let x denote the log output vector, (y, z) , and $f(x) = \pi(y, z)$ denote the objective. Table 4.1 displays the iterates for Newton's method. The second column contains $x^k - x^*$, the errors of the Newton iterates, beginning with $x^0 = (y_0, z_0) = (1.5, 2.0)$. Quadratic convergence is apparent. Column 1 contains the iteration counter, k . Column 2 contains the deviation of x^k from the optimal solution, x^* . Column 3 contains $f(x^k) - f(x^*)$, the "loss" from choosing x^k instead of x^* . Column 4 displays the steps taken by each iterate, $x^{k+1} - x^k$. Column 5 contains the coordinates of $\nabla f(x^k)/(1 + |f(x^k)|)$, the normed gradient used in step 4 of Newton's method.

The quadratic nature of convergence is demonstrated clearly in column 2 where the error at each iterate roughly equals the square of the error at the previous iterate once the error is less than 1. The properties of various convergence rules can be inferred. We see that if we only demanded that $\|\nabla f\| \leq 0.001$, we would stop at $k = 5$, but that by also requiring $\|x^k - x^{k-1}\| \leq 0.01$, we continue to $k = 7$ and reduce the error in x^k from 0.2 to 0.007. Notice that at $k = 9$, the error and the gradients were reduced to magnitudes close to machine ε .

Table 4.1
Newton's method applied to (4.3.5)

k	$x^k - x^*$	$f(x^k) - f(x^*)$	$x^k - x^{k-1}$	$\nabla f(x^k)/(1 + f(x^k))$
0	(2.1, 0.92)	-0.4		(-0.43, -0.62)
1	(1.5, 0.37)	-0.08	(-0.61, -0.56)	(-0.13, -0.18)
2	(0.98, -0.0029)	-0.012	(-0.48, -0.37)	(-0.034, -0.04)
3	(0.61, -0.12)	-0.0019	(-0.36, -0.11)	(-0.0087, -0.0045)
4	(0.28, -0.041)	-0.0003	(-0.33, 0.075)	(-0.0028, -0.0022)
5	(0.075, -0.0093)	-1.9(-5)	(-0.2, 0.032)	(-6.1(-4), -6.5(-4))
6	(0.0069, -7.8(-4))	-1.6(-7)	(-0.068, 0.0085)	(-5.3(-5), -6.4(-5))
7	(6.3(-5), -7.0(-6))	-1.3(-11)	(-0.0068, 7.7(-4))	(-4.8(-7), -6.0(-7))
8	(5.4(-9), -6.0(-10))	-3.2(-16)	(-6.3(-5), 7.0(-6))	(-4.1(-11), -5.1(-11))
9	(-2.3(-15), 2.4(-15))	-3.2(-16)	(-5.4(-9), 6.0(-10))	(-4.0(-17), -3.2(-16))

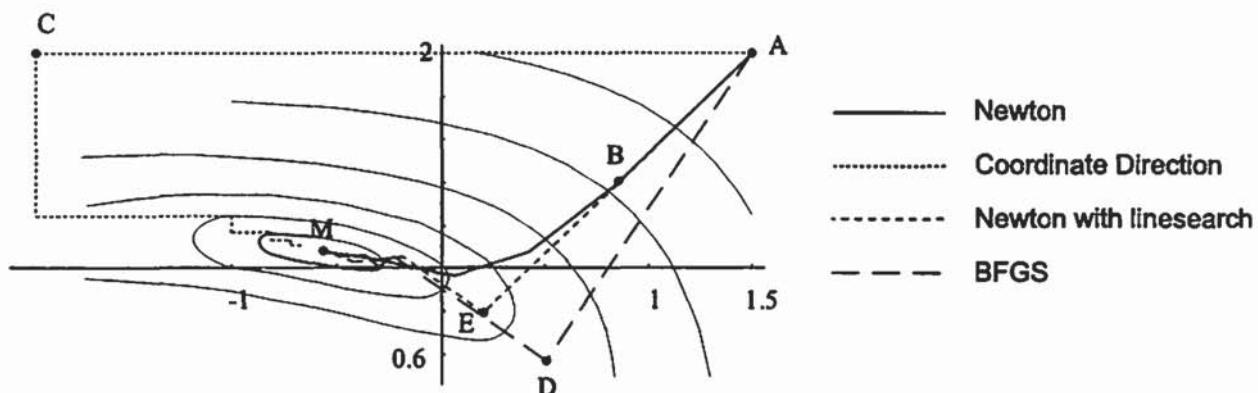


Figure 4.3
Various methods applied to (4.3.5)

Figure 4.3 displays the Newton iterates computed in table 4.1. The initial guess is at point A and the first iterate is B . After that the Newton iterates move along the solid line to the minimum at M .

Each step of Newton's method requires calculation of the Hessian, a step that is costly in large problem. One modification is to compute the Hessian only once at the initial point and use it to calculate each iteration; this is called the *modified Newton method*. The formal definition is to choose an initial guess, x^0 , compute $H(x^0)$, and compute the sequence $x^{k+1} = x^k - H(x^0)^{-1}(\nabla f(x^k))^T$. This will converge for sufficiently good initial guesses, but only linearly.

Although the modified Newton method may seem rather unappealing, there may be cases where it is useful if the Hessian is very difficult to compute. Also, when an

application of the Newton method appears to be converging, one may want to switch to the modified Newton method, since the Hessian will not change much during the final iterations of Newton's method.

Notes on Computing Gradients and Hessians

In many optimization problems much computing time is devoted to computing gradients and Hessians. We must choose between finite difference methods and analytic computations. We will usually want to use finite difference methods, since analytic computation of derivatives by human hand is at best a tedious, error-laden process for most people. Fortunately finite-difference methods are often adequate. At the early stages of a computational project, we want to stick with finite differences because it is very costly to recompute derivatives each time the model is modified. For most problems the option of moving to analytic gradients and Hessians is relevant only at the final stage if more accuracy or speed is needed.

In some problems we cannot compute the true objective and must replace it with numerical approximations. This is the case, for example, whenever the objective includes an integral that must be computed numerically. We then must decide the accuracy needed for the objective. We will generally want to use a good approximation of the objective. Since the objective is only an approximation, any derivative we compute is also only an approximation. We could just approximate the derivatives of the true objective with the derivatives of the approximation we use for the objective. However, the question of how well we approximate the gradients and Hessians is independent of the approximation of the objective. To save time (remember, most of the running time in optimization problems is spent on gradient and Hessian computations), we may want to use lower-quality approximations for gradients and Hessians but use high-quality approximations for the objective.

Carter (1993) shows that high accuracy in computing Hessians and gradients is often not critical. He took a group of test optimization problems and used standard methods to solve them, but after computing gradients and Hessians analytically, he added random errors. These errors ranged from 1 to 80 percent of the true derivative. He showed that the optimization algorithms generally performed well even when he used 20 percent errors, and often did well even with 50 percent errors.

All this indicates that finite difference approximations of derivatives are generally adequate. In this book every problem can be successfully solved using finite differences unless it is explicitly stated otherwise; this text's attitude is that programmers have better things to do than taking difficult derivatives. Furthermore, if analytical derivatives are necessary, then they should be done by the computer using the symbolic software if possible.

LOL

4.4 Direction Set Methods

The most useful methods in optimization of smooth functions are direction set methods. Direction set methods pick a series of directions to search, performing a sequence of one-dimensional searches in each direction. If f is C^1 the first-order conditions of (4.2.1) state that the gradient of f vanishes at a local minimum, that is,

$$\nabla f \cdot u = 0 \quad \forall u \in R^n. \quad (4.4.1)$$

Equation (4.4.1) holds for all $u \in R^n$ if $f \in C^2$ for n linearly independent vectors. The idea of direction set methods is to find a sequence of directions, and then, by repeated one-dimensional minimizations in those directions, to find a point where (4.4.1) holds. These methods reduce multidimensional optimization problems to a series of one-dimensional problems, which are easily solved. The hope is that eventually the process converges to a point where f is minimal in all directions.

The various direction set methods differ in terms of how they choose successive directions and approximants. However, they all roughly follow the same idea. First, given information about f at x^k , compute a search direction, s^k . Then find the scalar λ_k that minimizes f along the line $x^k + \lambda_k s^k$. The next iterate is $x^{k+1} = x^k + \lambda_k s^k$. If the derivatives of f at x^{k+1} and $\|\lambda_k s^k\|$ are close enough to zero, stop; otherwise, repeat.

Algorithm 4.5 Generic Direction Method

Initialization. Choose initial guess x^0 and stopping parameters δ and $\varepsilon > 0$.

Step 1. Compute a search direction s^k .

Step 2. Solve $\lambda_k = \arg \min_{\lambda} f(x^k + \lambda s^k)$.

Step 3. $x^{k+1} = x^k + \lambda_k s^k$.

Step 4. If $\|x^k - x^{k+1}\| < \varepsilon(1 + \|x^k\|)$, go to step 5; else go to step 1.

Step 5. If $\|\nabla f(x^k)\| < \delta(1 + f(x^k))$, STOP and report success; else STOP and report convergence to nonoptimal point.

Various direction set methods differ in how the directions are chosen. The directions may be confined to some specified set, or the search directions may vary as the algorithm proceeds. Some sensible choices are not good, but we will see that good direction sets can be derived. The main objective is that the search direction be a good descent direction at each iteration. This will force $f(x^{k+1})$ to be nontrivially less than $f(x^k)$ and impose the chance of convergence. This is not always true of Newton's method.

Table 4.2
Coordinate direction method applied to (4.3.5)

k	$x^k - x^*$	$x^k - x^{k-1}$	$\nabla f(x^k)/(1 + f(x^k))$
0	(2.1, 0.92)		(-0.43, -0.62)
1	(-4.8, 0.92)	(-6.9, 0)	(5.4(-7), -0.39)
2	(-4.8, 0.16)	(0, -0.76)	(2.2(-4), 3.3(-10))
3	(-0.49, 0.16)	(4.3, 0)	(-3.5(-9), -2.0(-2))
4	(-0.49, 0.065)	(0, -0.096)	(1.8(-3), -6.7(-12))
5	(-0.18, 0.065)	(0.31, 0)	(1.0(-9), -6.7(-3))
6	(-0.18, 0.028)	(0, -0.037)	(8.9(-4), 3.5(-13))
7	(-0.074, 0.028)	(0.11, 0)	(6.1(-10), -2.7(-3))
8	(-0.074, 0.012)	(0, -0.016)	(4.1(-4), 1.1(-11))
9	(-0.032, 0.012)	(0.042, 0)	(2.4(-10), -1.1(-3))

Coordinate Directions

The coordinate direction method cycles over the set of unit coordinate vectors, $\{e^j : j = 1, \dots, n\}$. That is, on iteration $mn + j + 1$, let λ^* minimize $f(x^{mn+j} + \lambda e^j)$, and make the next iterate $x^{mn+j+1} = x^{mn+j} + \lambda^* e^j$. The coordinate direction method will converge to a local minimum for smooth functions because it is always moving in a downhill direction, but it may converge slowly.

Table 4.2 displays the first ten iterates of the coordinate direction method applied to (4.3.5) with $x^0 = (y_0, z_0) = (1.5, 2.0)$. The format follows that of table 4.1. We see that convergence is rather slow. In fact the first iteration badly overshoots the solution in the x coordinate, but the algorithm recovers and heads to the optimal point. This is displayed in figure 4.3 by the dotted line beginning at A , proceeding to C , and then moving to the optimum. Even when close, the convergence is poor with the errors $x^k - x^*$ converging to zero slowly. The last column shows how each iteration sets one of the gradients close to zero, but then that gradient is nonzero in the next iterate. This is seen in figure 4.3 where the later iterates move right, then down, then right, and so on. At each corner one coordinate direction is tangent to the contour, but the other coordinate direction is orthogonal to the contour.

The coordinate direction method is one where we reduce a multivariate problem to a sequence of univariate ones. In this regard it is similar to the Gauss-Seidel approach to iteration discussed in section 3.6. We could also implement a block Gauss-Seidel extension of the coordinate direction method for optimization. This *block coordinate direction* method partitions the set of optimization variables into a collection of smaller sets and then optimizes over the variables in each smaller set, using the sets

Table 4.3
Steepest descent applied to (4.3.5)

k	$x^k - x^*$	$x^k - x^{k-1}$	$\nabla f(x^k)/(1 + f(x^k))$
0	(2.1, 0.92)		(-0.43, -0.62)
1	(1.1, -0.5)	(-1.0, -1.4)	(-0.014, 0.0098)
2	(0.38, -0.029)	(-0.68, 0.48)	(-0.0055, -0.0078)
3	(0.35, -0.084)	(-0.039, 0.055)	(-0.0026, 0.0018)
4	(0.26, -0.023)	(-0.088, 0.062)	(-0.003, -0.0043)
5	(0.24, -0.053)	(-0.021, -0.03)	(-0.0016, 0.0011)
6	(0.18, -0.017)	(-0.051, -0.036)	(-0.0019, -0.0027)
7	(0.17, -0.037)	(-0.014, -0.02)	(-0.0011, 0.00076)
8	(0.14, -0.013)	(-0.034, 0.024)	(-0.0013, -0.0019)
9	(0.13, -0.027)	(-0.0095, -0.014)	(-0.00077, 0.00054)

in succession. If each set of the partition is small enough, Newton's method can be used at each iteration, although it could not be used for the original problem. The advantage of a block approach is that we can exploit some of the multivariate structure in the objective without tackling all of the variables at once.

Steepest Descent

In the steepest descent method the search direction from a point x is taken to be that direction along which f falls most rapidly per unit length. The steepest ascent direction is the gradient. Therefore the steepest descent method chooses the search direction in step 1 in the generic direction method to be

$$s^{k+1} = -(\nabla f(x^k))^T. \quad (4.4.2)$$

At first the steepest descent method appears promising. It will always be a descent direction, and locally it is the best direction to go. However, this local property does not hold globally. Steepest descent often has a slow rate of convergence.

Table 4.3 displays the first ten iterates of the steepest descent method applied to (4.3.5) with $x^0 = (1.5, 2.0)$; we follow the format of table 4.1. We see that convergence is much slower than Newton's method. We see that initially the method does well, but that near the solution it zigzags into the solution. It is trying to find the bottom of a valley, but it essentially bounces from valley side to side because the steepest descent direction never points in the direction of the true minimum. Newton's method does much better. It takes a more truly multivariate view of the problem and goes in a direction that is more globally optimal instead of the locally optimal direction.

Both coordinate search and steepest descent are sensible methods, but they are slow. They suffer from a common problem. The direction searched generally misses the true optimal point. While the steepest descent direction is the best direction locally, it is unlikely to be pointing in the direction of the global optimum. We need to find better directions.

Newton's Method with Line Search

Newton's method is a special case of a direction set method. It can be expressed in the form of the generic direction set method with the choices $s^k = -H(x^k)^{-1}(\nabla f(x^k))^\top$ and $\lambda_k = 1$. Therefore Newton's method is a direction set method with a fixed choice of $\lambda_k = 1$. Direction methods generally differ from Newton's method in both the direction examined and in their strategy of choosing λ_k .

The problem with Newton's method is that it takes the quadratic approximation too seriously, treating it as a global approximation of f . An alternative view is to think of the local quadratic approximation as telling us a good direction to use but to recognize that the quality of that approximation will decay as we move away from its base at x^k . In fact the Newton direction is a good direction initially if $H(x^k)$ is positive definite because $f(x)$ does fall as we move away from x^k in the Newton direction. When f is not a linear-quadratic function, the ray $s^k = -H(x^k)^{-1}(\nabla f(x^k))^\top$ may initially go downhill, though a full step may send it uphill again. In Newton's method with line search, we take the Newton step direction and use line search to find the best point in that direction. Therefore, in the generic direction method algorithm, we implement step 1 with

$$s^k = -H(x^k)^{-1}(\nabla f(x^k))^\top. \quad (4.4.3)$$

The advantage of this method over steepest descent is that it uses a direction that may be good globally. In fact, if we are close to the minimum or minimizing a nearly quadratic function, the Newton direction is the best possible direction. The advantage of this method over Newton's method is that the line search step guarantees that we will go downhill if f is smooth, and thereby guaranteeing convergence to a local minimum. In contrast, Newton's method ($\lambda_k = 1$) may overshoot the best point in the Newton direction and push the iteration uphill.

Unfortunately, convergence may be slower. First, there is the cost of doing the line search. Second, when we are close to the solution, we really want to use Newton's method, that is, have $\lambda_k = 1$, because that will guarantee quadratic convergence; if λ_k does not converge to unity, the asymptotic convergence rate will be less than quadratic. As we approach a minimum, it may be advisable to switch to Newton's method, since we know that λ_k will be unity.

Table 4.4
Newton's method with linesearch applied to (4.3.5)

k	$x^k - x^*$	$x^k - x^{k-1}$	$\nabla f(x^k)/(1 + f(x^k))$	λ_k
0	(2.1, 0.92)		(-0.43, -0.62)	2.15
1	(0.75, -0.28)	(-1.3, -1.2)	(-0.007, 0.0076)	0.791
2	(0.35, -0.022)	(-0.4, 0.26)	(-0.005, -0.0078)	1.37
3	(0.047, -0.019)	(-0.3, 0.0027)	(1.5(-5), 0.0017)	0.966
4	(8.5(-4), -1.5(-5))	(-0.046, 0.019)	(-8.7(-6), -2.1(-5))	1.00
5	(3.6(-7), -1.5(-7))	(-8.5(-4), 1.5(-5))	(2.6(-10), 1.4(-8))	1.00
6	(8.2(-13), -3.2(-13))	(-3.6(-7), 1.5(-7))	(4.0(-17), 2.9(-14))	1.00
7	(5.8(-15), -6.7(-16))	(-8.1(-13), 3.1(-13))	(4.0(-17), 3.2(-16))	1.00
8	(3.4(-15), 1.8(-15))	(-2.3(-15), 2.4(-15))	(-8.1(-17), -3.2(-16))	

Table 4.4 displays the first ten iterates of Newton's method with line search applied to (4.3.5) with $x^0 = (1.5, 2.0)$; we follow the format of table 4.1, but we add a new column to display the λ_k sequence. Initially the step size parameter, λ_k , exceeds unity. As we get close, the quadratic approximation is quite good, and λ_k becomes 1, implying that the line search gives us the same point as would Newton's method. Figure 4.3 compares the Newton iterates and the iterates from Newton with line search, the latter beginning at A and proceeding along the finely broken line through B to the first iterate at E . Point E is clearly a better first iterate than B . In this case we want to overshoot the Newton iterate B because E has a lower objective value. From E we again use Newton with line search and converge to the solution more quickly than by Newton's method. Even though the line search feature will become less relevant as we get close, it will often substantially improve the initial progress.

There are potentially severe problems with any Newton-type method that uses $H(x)$. First, $H(x)$ which has n^2 elements can be quite costly to compute and store. Second, solving (4.4.3) will be costly if n is large. The next set of methods address these problems.

Quasi-Newton Methods

Quasi-Newton method takes Newton's method with line search and replaces the Hessian with another $n \times n$ matrix that performs the role of the Hessian in generating the search direction. This approximate Hessian is always positive semidefinite but easier to compute and invert. This means that at each iteration, we are looking for some easily computed, positive definite H_k that both generates a good search direction, $s^k = -H_k^{-1}(\nabla f(x^k))^\top$ which is like the Hessian in that

$$H_k^{-1}((\nabla f(x^k))^\top - (\nabla f(x^{k-1}))^\top) = x^k - x^{k-1},$$

and guarantees that s^k is a descent direction. Actually this last condition is assured if H_k is positive definite, since then $\nabla f(x^k)s^k = -\nabla f(x^k)H_k^{-1}(\nabla f(x^k))^\top < 0$. The key factor is the construction of a good H_k satisfying these first two conditions.

There are several good schemes for constructing H_k . One of the favorites is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update. The BFGS iteration is given in algorithm 4.6. The formula in (4.4.4) is the pure version of BFGS. However, to avoid overflow problems in (4.4.4), we set $H_{k+1} = H_k$ whenever $y_k^\top z_k$ is small relative to $\|y_k\|$.

Algorithm 4.6 BFGS Method

Initialization. Choose initial guess x^0 , initial hessian guess H^0 , and stopping parameters δ and $\varepsilon > 0$.

Step 1. Solve $H_k s^k = -(\nabla f(x^k))^\top$ for the search direction s^k .

Step 2. Solve $\lambda_k = \arg \min_\lambda f(x^k + \lambda s^k)$

Step 3. $x^{k+1} = x^k + \lambda_k s^k$.

Step 4. Update H_k :

$$z_k = x^{k+1} - x^k,$$

$$y_k = (\nabla f(x^{k+1}))^\top - (\nabla f(x^k))^\top,$$

$$H_{k+1} = H_k - \frac{H_k z_k z_k^\top H_k}{z_k^\top H_k z_k} + \frac{y_k y_k^\top}{y_k^\top z_k}. \quad (4.4.4)$$

Step 5. If $\|x^k - x^{k+1}\| < \varepsilon(1 + \|x^k\|)$, go to step 6; else go to step 1.

Step 6. If $\|\nabla f(x^k)\| < \delta(1 + |f(x^k)|)$, STOP and report success; else STOP and report convergence to nonoptimal point.

This H_k sequence is positive definite and symmetric as long as H_0 is. One common choice for H_0 is the identity matrix. In that case the initial step is the same as the steepest descent step. If a system is not too large, H_0 can be set equal to the Hessian at x^0 , making the initial step equal to Newton's method with line search. Since all H_k are positive definite, Cholesky factorization can be used to solve for the search direction in step 1.

Equation (4.4.4) is the BFGS update. An older method is the simpler Davidon-Fletcher-Powell (DFP) update, which replaces (4.4.4) with (4.4.5):

$$\begin{aligned}
 H_{k+1} = H_k + & \frac{(y_k - H_k z_k) y_k^\top + y_k (y_k - H_k z_k)^\top}{y_k^\top z_k} \\
 & - \frac{z_k^\top (y_k - H_k z_k) y_k y_k^\top}{(y_k^\top z_k)^2}.
 \end{aligned} \tag{4.4.5}$$

General experience indicates that BFGS works more often than DFP but that DFP does better in some cases. Therefore it is best to be familiar with and have software for both methods.

In Hessian update algorithms the H_k sequence does not necessarily converge to the true Hessian at the solution. Therefore, if it is needed, the Hessian should be computed at the end. One such case is computing a maximum likelihood estimator where the Hessian is used for inference. Since the final Hessian must be computed anyway in these econometric contexts, one should switch to Newton's method once the optimization procedure has apparently converged. This has little cost: If the problem has converged, then one iteration of Newton's method should suffice; if more is needed, then that is evidence that the search should continue.

Table 4.5 displays the first ten iterates of the BFGS method applied to (4.3.5) with $x^0 = (1.5, 2.0)$ and H_0 being the true Hessian at x^0 . We see that convergence is quite rapid, even though BFGS computes $H(x)$ only once. Figure 4.3 displays the BFGS method, beginning at A , proceeding along the steepest descent direction to D , and then converging to M . The first iterate of BFGS is better than the first Newton iterate at B , but not as good as the first iterate of Newton with line search at E . The Hessian approximation, though, allows BFGS to quickly catch up with Newton with line search, whereas continuing with the steepest descent direction method does less well.

Table 4.5
BFGS applied to (4.3.5)

k	$x^k - x^*$	$x^k - x^{k-1}$	$\nabla f(x^k)/(1 + f(x^k))$	λ_k
0	(2.1, 0.92)		(-0.43, -0.62)	2.16
1	(0.75, -0.28)	(-1.3, -1.2)	(-0.007, 0.0076)	20.5
2	(0.38, -0.026)	(-0.37, 0.25)	(-0.0055, -0.0081)	5.75
3	(0.15, -0.049)	(-0.23, -0.023)	(-3.3(-4), 0.0034)	0.928
4	(0.038, -0.0012)	(-0.11, 0.048)	(-0.0004, -8.9(-4))	1.61
5	(0.0015, -6.2(-4))	(-0.037, 5.9(-4))	(9.6(-7), 6.0(-5))	0.93
6	(4.6(-5), -1.1(-6))	(-0.0015, 6.2(-4))	(-4.6(-7), -1.1(-6))	

Conjugate Directions

Newton and quasi-Newton methods carry around an estimate of the Hessian. This estimate may be large in high-dimensional problems, and a computer may not be able to store such a large matrix for large n . For example, personal computers today have no problem storing vectors with 1000 elements but cannot store a 1000×1000 matrix under normal conditions. To solve large problems, we need to develop methods that use less storage. Steepest descent and coordinate direction methods have small storage needs, storing only a gradient, but are rather poor methods.

Conjugate direction methods are direction set methods that store only a gradient, but they are successful because they implicitly keep track of curvature information in a useful way without storing a Hessian or keeping a history of past search directions. The key insight about conjugate direction methods comes from considering the failings of the steepest descent and coordinate direction methods. One reason they do poorly is that after they minimize f in one direction, u , they move along another direction, v , and likely end up at a point where f is no longer a minimum along the u direction. After moving in the u direction, the algorithm has solved one first-order condition, but moving in the v direction will lose optimality in the u direction. A better algorithm would choose v to maintain optimality in the u direction as it moves in the v direction.

The concept of conjugate directions makes this idea formal and implements it. Let us focus on the generic quadratic function $f(x) = \frac{1}{2}x^T Ax + b^T x + c$. The gradient of f is $\nabla f = (Ax + b)^T$, and the change in ∇f as a point moves from x to $x + dx$ is $A dx$. After we minimize along a direction u , we get to a point, x , where

$$(Ax + b)^T u = 0. \quad (4.4.6)$$

If we next move from x to $x + v$, the change in (4.4.6) is $v^T Au$. If we want to keep the u -direction first-order condition (4.4.6) true, we must choose a direction v such that $0 = v^T Au$. If u and v satisfy $v^T Au = 0$, we say that u and v are *conjugate directions*.

We can now see a problem with applying the steepest descent and coordinate direction methods. In both cases the second direction, v , is orthogonal to the first, u , where v should instead be orthogonal to Au . In R^n there are many directions conjugate to u . There is a simple procedure that will generate a sequence of directions such that when applied to a quadratic problem, each direction is conjugate not only to the previous direction but to the previous $n - 1$ directions. We begin with a guess, x^0 , and an initial search direction, $s^0 = -(\nabla f(x^0))$. Thereafter the iterative scheme proceeds as described in algorithm 4.7. This method also has a nice characterization

relative to the steepest descent method. The conjugate direction step in (4.4.7) takes the steepest descent direction, $-(\nabla f(x^{k+1}))^\top$, and deflects it toward the last step, s^k .

Algorithm 4.7 Conjugate Gradient Method

Initialization. Choose initial guess x^0 , $s^0 = -\nabla f(x^0)$ and stopping parameters δ and $\varepsilon > 0$.

Step 1. Solve $\lambda_k = \arg \min_\lambda f(x^k + \lambda s^k)$.

Step 2. Set $x^{k+1} = x^k + \lambda_k s^k$.

Step 3. Compute the search direction

$$s^{k+1} = -(\nabla f(x^{k+1}))^\top + \frac{\|\nabla f(x^{k+1})\|^2}{\|\nabla f(x^k)\|^2} s^k. \quad (4.4.7)$$

Step 4. If $\|x^k - x^{k+1}\| > \varepsilon(1 + \|x^k\|)$, go to step 1. Otherwise, if $\|\nabla f(x^k)\| < \delta(1 + |f(x^k)|)$, STOP and report success; else STOP and report convergence to non-optimal point.

In the conjugate direction algorithm there is no explicit Hessian calculation. Implicitly it differs from quasi-Newton methods in that it assumes that a single quadratic form is a valid approximation globally, whereas a quasi-Newton method knows it is tracking a moving target and changes its estimation of the current quadratic approximation. The implicit global approximation may break down and lead to successive conjugate directions being highly collinear. To keep the sequence of directions in a conjugate direction method from degenerating, we restart the iteration every so often (approximately every n steps, where n is the dimension of x) by setting the search direction equal to the steepest descent direction as in, for example, $s^{(kn)} = -(\nabla f(x^{(kn)}))^\top$.

Table 4.6 displays the first ten iterates of the conjugate gradient method applied to (4.3.5) with $x^0 = (1.5, 2.0)$. We see that convergence is initially rapid but slows down as expected.

4.5 Nonlinear Least Squares

Least squares problems are common in economics, particularly in econometric analysis. The typical problem has the form

$$\min_x \frac{1}{2} \sum_{i=1}^m f^i(x)^2 \equiv S(x),$$

Table 4.6
Conjugate gradient method applied to (4.3.5)

k	$x^k - x^*$	$x^k - x^{k-1}$	$\nabla f(x^k)/(1 + f(x^k))$
0	(2.1, 0.92)		(-0.43, -0.62)
1	(1.1, -0.5)	(-1, -1.4)	(-0.014, 9.8(-3))
2	(0.34, -0.024)	(-0.73, 0.48)	(-4.5(-3), -6.9(-3))
3	(0.26, -0.068)	(-0.076, -0.045)	(-1.5(-3), 2.5(-3))
4	(0.2, -0.021)	(-0.061, 0.048)	(-2.0(-3), -2.6(-3))
5	(0.067, -0.025)	(-0.13, -3.7(-3))	(-5.6(-5), 2.0(-3))
6	(0.048, -0.006)	(-0.019, 0.019)	(-3.7(-4), -3.8(-4))
7	(0.041, -9.5(-3))	(-7.2(-3), -3.5(-3))	(-1.9(-4), 3.9(-4))
8	(0.02, -5.7(-4))	(-0.02, 8.9(-3))	(-2.1(-4), -4.8(-4))
9	(8.4(-3), -2.4(-3))	(-0.012, -1.8(-3))	(-2.4(-5), 1.6(-4))

where $f^i: R^n \rightarrow R$ for $i = 1, \dots, m$. Let $f(x)$ denote the column vector $(f^i(x))_{i=1}^m$. In econometric applications, $x = \beta$ and the $f^i(x)$ are $f(\beta, y^i)$, where β is the vector of unknown parameters and the y^i are the data. In these problems $f(\beta, y^i)$ is the residual associated with observation i , and $S(\beta)$ is the sum of squared residuals if β is the value of the parameters.

To solve a least squares problem, we can apply any general optimization method. However, there are methods that exploit the special structure of least squares problems. Let $J(x)$ be the Jacobian of $f(x) \equiv (f^1(x), \dots, f^m(x))^\top$. Let $f_l^i \equiv \partial f^i / \partial x_l$ and $f_{jl}^i \equiv \partial^2 f^i / \partial x_j \partial x_l$. The gradient of $S(x)$ is $J(x)^\top f(x)$, with element l being $\sum_{i=1}^m f_l^i(x) f^i(x)$. The Hessian of $S(x)$ is $J(x)^\top J(x) + G(x)$, where the (j, l) element of $G(x)$ is

$$G_{jl}(x) = \sum_{i=1}^m f_{jl}^i(x) f^i(x).$$

Note the special structure of the gradient and Hessian. The $f_j^i(x)$ terms are necessary for computing the gradient of $S(x)$. If $f(x)$ is zero, then the Hessian reduces to $J(x)^\top J(x)$, which is composed of $f_j^i(x)$ terms only. The $J(x)^\top J(x)$ piece of the Hessian is much easier to compute than the whole Hessian. It involves no evaluations of f and its derivatives other than those necessary for computing the gradient, which is necessary to compute in almost all algorithms. A problem where $f(x)$ is small at the solution is called a *small residual problem*, and a *large residual problem* is one where $f(x)$ is large at the solution. For small residual problems the Hessian becomes

nearly equal to $J(x)^\top J(x)$ near the solution. This means that we can compute a good, new approximation to the Hessian at each iteration at low cost.

This leads to the Gauss-Newton algorithm, which is Newton's method except that it uses $J(x)^\top J(x)$ as an estimate of the Hessian. The Gauss-Newton step is

$$s^k = -(J(x^k)^\top J(x^k))^{-1}(\nabla f(x^k))^\top. \quad (4.5.1)$$

This results in considerable savings, for no second derivatives of f are evaluated. When it works, it works very well, far faster than Newton's method because there is no evaluation of second derivatives.

The Gauss-Newton approach has many problems. First, the product $J(x)^\top J(x)$ is likely to be poorly conditioned, since it is the "square" of a matrix. Second, $J(x)$ may be poorly conditioned itself, which is easily the case in statistical contexts. Third, the Gauss-Newton step may not be a descent direction.

These problems lead to the *Levenberg-Marquardt algorithm*. This method uses $J(x)^\top J(x) + \lambda I$ as an estimate of the Hessian for some scalar λ where I is the identity matrix. The Levenberg-Marquardt step is

$$s^k = -(J(x^k)^\top J(x^k) + \lambda I)^{-1}(\nabla f(x^k))^\top \quad (4.5.2)$$

The extra term in the Hessian approximation reduces the conditioning problems. More important, the resulting step will be a descent direction for sufficiently large λ since s^k converges to the steepest descent direction for large λ .

Solving Multiple Problems

Sometimes we will want to solve several problems of the form $\min_x f(x; a)$, where a is a vector of parameters. The choice of initial conditions substantially affects the performance of any algorithm. In any Hessian updating method, the initial Hessian guess is also important. When we solve several such problems, it is convenient to use information about the solution to $\min_x f(x; a)$ when computing the solution to $\min_x f(x; a')$ for a' close to a . In Newton's method, the solution to $f(x; a) = 0$ will be a good initial guess when solving $\min_x f(x; a')$. In Hessian updating methods we can use both the final solution and the final Hessian estimate from $\min_x f(x; a)$ as the initial guesses when solving $\min_x f(x; a')$. In either case we call these *hot starts*. Because of the importance of initial guesses, solving N problems is generally cheaper than N times the cost of one problem. The order in which we solve a collection of problems will be important; it should be chosen to maximize the economies of scale from hot starts.

4.6 Linear Programming

The previous sections examined unconstrained optimization problems. We now begin our consideration of problems with constraints on the choice variables. A special case of constrained optimization is *linear programming* where both the objective and constraint functions are linear. A canonical form for linear programming problems is

$$\begin{aligned} \min_x \quad & a^T x \\ \text{s.t.} \quad & Cx = b, \\ & x \geq 0. \end{aligned} \tag{4.6.1}$$

This form may seem restrictive, since it does not explicitly include inequality constraints of the form $Dx \leq f$. But there is no restriction because we can add *slack variables*, s , and reformulate $Dx \leq f$ as $Dx + s = f, s \geq 0$.

Similarly constraints of the form $Dx \geq f$ can be replaced by $Dx - s = f, s \geq 0$, where now s is a vector of *surplus variables*. The constraint $x \geq d$ is handled by defining $y = x - d$ and reformulating the problem in terms of y . The case of x_i being free is handled by defining $x_i = y_i - z_i$, reformulating the problem in terms of y_i and z_i , and adding the constraints $y_i, z_i \geq 0$. With these transformation in hand, any linear programming problem can be expressed in the form in (4.6.1).

The basic method for linear programming is the *simplex method*. We will not go through the details here but its basic idea is simple. Figure 4.4 displays graphically the problem

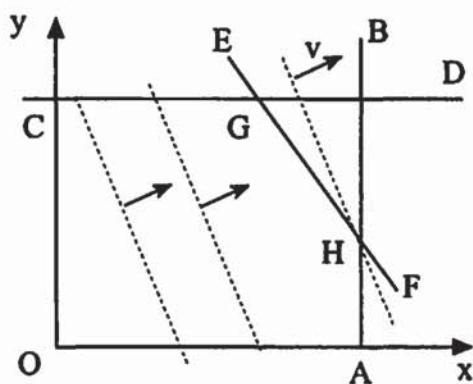


Figure 4.4
Linear programming

$$\begin{aligned} & \min_{x,y} -2x - y \\ \text{s.t. } & x + y \leq 4, \quad x, y \geq 0, \\ & x \leq 3, \quad y \leq 2. \end{aligned}$$

We do not convert this problem into the form of (4.6.1) since we want to illustrate it in x - y space. The inequality constraints define a convex feasible set with a piecewise linear surface. In figure 4.4 those constraints are $x + y \leq 4$, $x \leq 3$, and $y \leq 2$. The objective, $-2x - y$, is expressed as a series of dotted contour lines; the arrow points in the direction of decrease for the objective function.

The simplex method begins by finding some point on the surface of the constraint set, such as A in figure 4.4. We then move along the boundary of the feasible set to points that have a lower cost. The simplex method notes which constraints are active at A and points nearby. It then determines the feasible directions and determines which of these achieves the greatest rate of decrease in the objective. In figure 4.4 there are only two directions to go, one toward B and the other toward O , with the better one being toward B . The simplex algorithm follows that direction to the next vertex on the feasible set's boundary; more generally, it follows the direction of maximal decrease until a previously slack constraint becomes binding; in figure 4.4 that is point H . At H the set of binding constraints changes, and the simplex method notes this. With the new set of constraints at the new point, it repeats these steps to reach an even better point. This iteration continues until we reach a vertex from which all feasible movements increase the objective; in figure 4.4 that point is H . Since the constraint set is concave and the objective linear, this local minimum is also a global minimum. Since the simplex algorithm needs to visit only the finite set of vertices, it will converge in a finite number of steps.

The simplex method is an old and reliable way to solve linear programming problems, going back to Dantzig (1951). It is also fast; the average time for computing a solution is linear in the number of variables and constraints. The worst case running time is exponential, which is why the simplex algorithm is said to be exponential in time. Fortunately the worst-case scenario is unduly pessimistic in the case of the simplex method.

4.7 Constrained Nonlinear Optimization

Most economic optimization problems involve either linear or nonlinear constraints. Consumer optimization includes a budget constraint, optimal taxation problems

include revenue constraints, and principal-agent problems face incentive constraints. In this section I outline basic approaches to numerical solutions to general constrained optimization problems.

Let us first recall the basic theory. Suppose that we want to solve

$$\begin{aligned} \min_x f(x) \\ \text{s.t. } g(x) = 0, \\ h(x) \leq 0, \end{aligned} \tag{4.7.1}$$

where $f: R^n \rightarrow R$, $g: R^n \rightarrow R^m$, and $h: R^n \rightarrow R^l$, and f, g , and h are C^2 . The nature of the solution is displayed in its Kuhn-Tucker representation. The Kuhn-Tucker theorem says that if there is a local minimum at x^* and a constraint qualification holds,³ then there are multipliers $\lambda^* \in R^m$ and $\mu^* \in R^l$ such that x^* is a *stationary*, or *critical* point of \mathcal{L} , the *Lagrangian*,

$$\mathcal{L}(x, \lambda, \mu) = f(x) + \lambda^T g(x) + \mu^T h(x), \tag{4.7.2}$$

that is, $\mathcal{L}_x(x^*, \lambda^*, \mu^*) = 0$. The implied first-order conditions of the Lagrangian⁴ tells us that the multipliers λ^* and μ^* , with the constrained minimum of (4.7.1), x^* , solve the system

$$\begin{aligned} f_x + \lambda^T g_x + \mu^T h_x &= 0, \\ \mu_i h^i(x) &= 0 \quad i = 1, \dots, l, \\ g(x) &= 0, \\ h(x) &\leq 0, \\ \mu &\leq 0. \end{aligned} \tag{4.7.3}$$

A Kuhn-Tucker Approach

The following procedure reduces constrained optimization to a series of nonlinear equations. Let \mathcal{I} be the set $\{1, 2, \dots, l\}$. For a subset $\mathcal{P} \subset \mathcal{I}$, define the \mathcal{P} problem to be

- 3. As is typical in economic analysis, we ignore the constraint qualification condition in our discussion. However, it is a critical assumption in the foregoing. If one is not careful, it is easy to write down a problem where the constraint qualification fails.
- 4. Note that we assert only that x^* is a stationary point of \mathcal{L} ; it is not necessarily a minimum.

$$\begin{aligned}
 g(x) &= 0, \\
 h^i(x) &= 0, \quad i \in \mathcal{P}, \\
 \mu^i &= 0, \quad i \in \mathcal{J} - \mathcal{P}, \\
 f_x + \lambda^\top g_x + \mu^\top h_x &= 0.
 \end{aligned} \tag{4.7.4}$$

This is a set of nonlinear equations.⁵ For each subset \mathcal{P} we have a different combination of binding and nonbinding inequality constraints, and we may have a solution; unfortunately, there may not be a solution to some of the combinations. Since there are just a finite number of such combinations, we can search over them to find the sets \mathcal{P} with solutions to the Kuhn-Tucker conditions that satisfy all the constraints. There may be many such feasible solutions; we choose the one with the smallest objective value.

This procedure reduces the problem to solving nonlinear systems of equations. In some cases we may know that only some combinations of binding constraints are possible. Sometimes the resulting nonlinear equations may be simple to solve. In those cases this approach will have some advantages.

Penalty Function Method

Most constrained optimization methods use a *penalty function* approach. These are based on a simple idea: Permit anything to be feasible, but alter the objective function so that it is “painful” to make choices that violate the constraints. This approach replaces constraints with continuous penalty functions and forms an unconstrained optimization problem.

For example, consider the constrained problem

$$\begin{aligned}
 \min_x \quad &f(x) \\
 \text{s.t.} \quad &g(x) = a, \\
 &h(x) \leq b.
 \end{aligned} \tag{4.7.5}$$

We construct the penalty function problem

$$\min_x f(x) + \frac{1}{2} P \left(\sum_i (g^i(x) - a_i)^2 + \sum_j (\max[0, h^j(x) - b_j])^2 \right), \tag{4.7.6}$$

5. We defer discussing the details of solving nonlinear systems such as (4.7.4) until chapter 5.

where $P > 0$ is the penalty parameter. Denote the penalized objective in (4.7.6) $F(x; P, a, b)$; note that we are including the constraint constants a and b as parameters of the problem. If P is “infinite,” then the two problems (4.7.5) and (4.7.6) are identical. For any P the penalty function will generally not have the same global minima as the constrained problem (4.7.5). However, for P large, their solutions will likely be close.

Unfortunately, we cannot directly solve (4.7.6) for very large P on real-world computers because, when P is large, the Hessian of F , F_{xx} , is likely to be ill-conditioned at points away from the solution, leading to numerical imprecision and slow progress at best. The solution is to solve a sequence of problems. We first solve $\min_x F(x; P_1, a, b)$ with a small choice of P_1 , yielding a solution x^1 . We then execute the iteration

$$x^{k+1} \in \arg \min_x F(x; P_{k+1}, a, b), \quad (4.7.7)$$

where the conditioning problem is alleviated by using x^k for the initial guess in the iteration $k + 1$ minimization problem in (4.7.7). At each iteration k the Hessian at the solution, $F_{xx}(x^k; P_k, a, b)$, will generally be well-conditioned because the constraint violations will be small. Then, if P_k does not rise too rapidly, the initial Hessian in the $k + 1$ iteration, $F_{xx}(x^k; P_{k+1}, a, b)$, will be close to $F_{xx}(x^k; P_k, a, b)$ and not be ill-conditioned, in which case convergence to x^{k+1} should be as rapid as expected given the minimization method used.

Not only does the x^k sequence converge to the solution, but we can also compute the shadow prices to (4.7.5). The shadow prices of the constraints in (4.7.5) correspond to the shadow prices of the parameters a and b in (4.7.6). The shadow price of a_i in (4.7.6) is $F_{a_i} = P(g^i(x) - a_i)$. The shadow price of b_j in (4.7.6) is F_{b_j} . F_{b_j} is zero if the corresponding constraint is not violated; otherwise, it equals $P(h^j(x) - b_j)$. In general, the shadow price of a binding constraint is approximated by the product of the penalty parameter and the constraint violation. Let $\lambda(P, a, b)$ be the shadow prices, (F_a, F_b) , of the constraint constants (a, b) at a local solution to (4.7.6) with penalty parameter P . The following theorem states the properties of the penalty function method more precisely:

THEOREM 4.7.1 Let λ^* denote the vector of shadow prices of the constraints in (4.7.5) at a solution x^* . Suppose that the Jacobian of the active constraints at x^* is of full rank and that x^* is the unique solution to (4.7.5) on a compact set, C . Then, if $P_k \rightarrow \infty$, there is a sequence x^k satisfying (4.7.7) that asymptotically lies in C , and converges to x^* . Furthermore the shadow prices of the penalized problems converge to the constrained shadow price, namely $\lambda(P_k, a, b) \rightarrow \lambda^*$.

Since the shadow prices of the penalized problem converge to the constrained shadow prices, we conclude that $P(g^i(x) - a_i) \rightarrow \lambda_i^*$ and $P(h^j(x) - b_j) \rightarrow \lambda_j^*$. These limits show that constraint violations will asymptotically be inversely proportional to the penalty parameter. While theorem 4.7.1 is true for any unbounded sequence of penalty parameters, conditioning problems mean that in practice, we must keep the growth rate down. Typically we will choose $P_{k+1} = 10P_k$; however, we may occasionally need slower rates of increase in P_k .

From our description of penalty methods, we make an important practical observation. Constrained minimization routines will often examine f at infeasible points, so the objective function f and the constraint functions g and h must be defined everywhere. This may require some alteration of all these functions. If any of these functions is changed to extend its range, that extension should be smooth enough that derivative-based optimization methods can be used without fear of getting hung up on kinks. We will return to this domain problem in the next chapter.

We must be careful in how we interpret Theorem 4.7.1. Consider the problem $\min_{0 \leq x \leq 1} x^3$. This problem leads to the penalty function $x^3 + P[(\min[x, 0])^2 + \max[x - 1, 0])^2]$, which is unbounded below as $x \rightarrow -\infty$, but it does have a local minimum at $x = 0$. This example emphasizes the local nature of theorem 4.7.1 and penalty methods. In any use of penalty methods, one must still check that the constraints are not being violated.

We will illustrate the penalty function method by applying it to a basic problem. Suppose that the consumer has two products to consume, good y having a price of 1 and good z a price of 2. Also suppose that total income is 5. Suppose that utility is $u(y, z) = \sqrt{yz}$. The optimal consumption bundle is $(y^*, z^*) = (5/2, 5/4)$, and the shadow price on the budget constraint is $\lambda^* = 8^{-1/2}$. The constrained optimization problem is

$$\begin{aligned} & \max_{y,z} \sqrt{yz} \\ & \text{s.t. } y + 2z \leq 5. \end{aligned} \tag{4.7.8}$$

To solve this, we maximize the penalized objective $u(y, z) - \frac{1}{2}P(\max[0, y + 2z - 5])^2$. Table 4.7 displays the iterates as we increase P . Note that the errors do fall rapidly to give good approximations but that the process bogs down for large P . Note that the constraint violation, $y + 2z - 5$, is inversely proportional to P , as predicted above. The error in the estimated shadow price, $P(5 - y - 2z) - \lambda^*$, appears stuck at about 10^{-4} .

Sequential Quadratic Method

The Kuhn-Tucker and penalty methods are not generally useful methods by themselves. One popular method is the *sequential quadratic* method. It takes advantage

Table 4.7
Penalty function method applied to (4.7.8)

k	P_k	$(y, z) - (y^*, z^*)$	Constraint violation	Shadow error
0	10	$(8.8(-3), 0.015)$	$1.0(-1)$	$-5.9(-3)$
1	10^2	$(8.8(-4), 1.5(-3))$	$1.0(-2)$	$-5.5(-4)$
2	10^3	$(5.5(-5), 1.7(-4))$	$1.0(-3)$	$2.1(-2)$
3	10^4	$(-2.5(-4), 1.7(-4))$	$1.0(-4)$	$1.7(-4)$
4	10^5	$(-2.8(-4), 1.7(-4))$	$1.0(-5)$	$2.3(-4)$

of the fact that there are special methods that can be used to solve problems with quadratic objectives and linear constraints. Suppose that the current guesses are (x^k, λ^k, μ^k) . The sequential quadratic method then solves the quadratic problem

$$\begin{aligned} & \min_s (x^k - s)^\top \mathcal{L}_{xx}(x^k, \lambda^k, \mu^k)(x^k - s) \\ \text{s.t. } & g_x(x^k)(x^k - s) = 0, \\ & h_x(x)(x^k - s) \leq 0, \end{aligned} \tag{4.7.9}$$

for the step size s^{k+1} . The problem in (4.7.9) is a linear-quadratic problem formed from a quadratic approximation of the Lagrangian and linear approximations of the constraints. The next iterate is $x^{k+1} = x^k + s^{k+1}$; λ and μ are also updated but we do not describe the details here. The sequential quadratic algorithm proceeds through a sequence of quadratic problems. The sequential quadratic method inherits many of the desirable properties of Newton's method, including local convergence, and can similarly use quasi-Newton and line search adaptations.

Augmented Lagrangian methods combine sequential quadratic methods and penalty function methods by adding penalty terms to the Lagrangian function. Solving the constrained optimization problem (4.7.1) may be different from minimizing its Lagrangian, even using the correct multipliers, but adding penalty terms can make the optimization problem similar to unconstrained minimization of the augmented Lagrangian. This augmented Lagrangian may be used just during a line search step or may be included in the quadratic approximation as well. These extra terms help limit the extent to which the iterates violate the constraints.

Reduced Gradient Methods

The addition of Lagrangian and penalty terms increases the size and complexity of the objective. *Reduced gradient methods* instead follow a path close to the feasible set. The idea is to specify a set of independent variables, use the equality constraints and

binding inequality constraints to solve for the remaining “dependent” variables, and then optimize over the independent variables subject to the nonbinding inequality constraints only. The idea of expressing the dependent variables in terms of the independent variables is a nonlinear problem, and it is executed at each step in terms of linear approximations. The set of independent variables will change between iterations, since we do not know which inequality constraints will bind at a solution. We refer the reader to Luenberger (1984) and Gill et al. (1981, 1989) for presentations of the details.

Active Set Methods

The difficulty with the direct Kuhn-Tucker approach is that there are too many combinations to check; some choices of \mathcal{P} may have no solution, and there may be multiple local solutions to others. Penalty function methods will also be costly, since all constraints must be computed in the penalty function (4.7.5), even if we know that only a few bind at the solution. These problems are all addressed by active set methods, whose strategy is to consider a rationally chosen series of subproblems.

Again let \mathcal{I} be the set $\{1, 2, \dots, l\}$, and for $\mathcal{P} \subset \mathcal{I}$ we define the \mathcal{P} problem to be

$$\begin{aligned} (\mathcal{P}) \quad & \min_x f(x) \\ \text{s.t.} \quad & g(x) = 0, \\ & h^i(x) \leq 0, \quad i \in \mathcal{P}. \end{aligned} \tag{4.7.10}$$

Active set methods solve a series of these subproblems. A typical algorithm chooses a set of constraints, \mathcal{P} , to consider, and begins to solve (4.7.10). However, as it attempts to solve this problem, it periodically checks to see if some of the constraints in \mathcal{P} fail to bind, in which case they are dropped. It also periodically checks to see if some other constraints are being violated, and adds them if they are. As the algorithm progresses, the penalty parameters are increased and constraints are added and deleted as appropriate. Asymptotically such methods will converge to the true constrained optimum with appropriate choices of penalty parameters and strategies for choosing and changing \mathcal{P} . The simplex method for linear programming is really an active set method.

This discussion of constrained optimization has only scratched the surface. Modern algorithms use sophisticated combinations of penalty function, active set, reduced gradient, and Lagrangian ideas together with local approximation methods. In the following sections we will use simple penalty methods to illustrate constrained optimization ideas. However, any serious constrained optimization problem should be analyzed using the sophisticated software.

4.8 Incentive Problems

The literature on incentive problems provides us with a wealth of constrained optimization problems. In this section we will use penalty methods to solve two simple problems, the first a hidden action and the other a hidden information model where agents have asymmetric information. While these examples are the simplest possible of their kind, it is clear that the numerical methods we use here are applicable to a wide range of contract design, optimal regulation, multiproduct pricing, and security design problems under asymmetric information.

Principal-Agent Problems

In principal-agent problems a “principal” hires an “agent” who, after signing the contract, takes actions that affect the payoffs to the partnership but are not observable to the principal. More formally, final output, y , is a random quantity depending on the agent’s effort, L . We assume that y takes on values $y_1 < y_2 \dots < y_n$, effort takes on m values, $L_j, j = 1, \dots, m$, and $\text{Prob}\{y = y_i | L\} = g_i(L)$ represents the effect of effort on output. The agent chooses his effort, but the principal observes only output. Therefore the uncertain payment, w , to the agent can be a function of output only; hence the wage is a random variable where $w = w_i$ if and only if $y = y_i$. The payment schedule is agreed upon when the agent is hired, and known by the agent when he makes his effort choice.

The problem is for the principal to choose a level of effort and a wage schedule such that the agent will choose that level of effort and will prefer working for the principal to his best alternative, the utility of which is R . Let $U^P(\pi)$ be the principal’s von Neumann-Morgenstern utility function over profit π , and let $U^A(w, L)$ be the agent’s utility function over wage, w , and effort, L . The principal’s problem is expressed as the constrained optimization problem

$$\begin{aligned} & \max_{L, w_i} E\{U^P(y - w)\} \\ \text{s.t. } & L \in \arg \max_L E\{U^A(w, L)\}, \\ & E\{U^A(w, L)\} \geq R. \end{aligned} \tag{4.8.1}$$

The constraint on L in (4.8.1) can be represented as a collection of inequality constraints

$$E\{U^A(w, L)\} \geq E\{U^A(w, L_j)\}, \quad j = 1, \dots, m.$$

This is not an easy optimization problem; the difficulties are discussed in Grossman and Hart (1983).

We consider the following simple two-state case: First, we assume a risk-neutral principal, that is, $U^P(\pi) = \pi$. Output will be either 0 or 2: $y_1 = 0$, $y_2 = 2$. The agent chooses one of two effort levels: $L \in \{0, 1\}$. High output occurs with probability 0.8 if the effort level is high ($\text{Prob}\{y = 2 | L = 1\} = 0.8$) but with probability 0.4 otherwise ($\text{Prob}\{y = 2 | L = 0\} = 0.4$). We assume that the agent's utility is $U^A(w, L) = u(w) - d(L) = -e^{-2w} + 1 - d(L)$ and that the disutility of effort is $d(0) = 0$, $d(1) = 0.1$. We assume that the agent's best alternative job is equivalent to $w = 1$ and $L = 1$ surely, implying that the reservation utility level, R , is $u(1) - d(1)$.

Let

$$\Delta(w_1, w_2) = E\{u(w) - d(1) | L = 1\} - E\{u(w) - d(0) | L = 0\}$$

be the expected utility difference between choosing $L = 1$ and $L = 0$. The interesting case is where the principal wants⁶ the agent to choose $L = 1$. This means that the wages must be such that $\Delta(w_1, w_2) \geq 0$. Furthermore the agents expected utility from working and choosing $L = 1$, $E\{u(w)\} - d(1)$, must exceed R . In this case the penalty function for (4.8.1) is

$$\begin{aligned} & 0.8(2 - w_1) - 0.2w_2 - P \max[0, -(E\{u(w) - d(1) | L = 1\} - R)]^2 \\ & - P \max[0, -\Delta(w_1, w_2)]^2. \end{aligned} \quad (4.8.2)$$

We will again apply Newton's method to (4.8.2) as we increase P . Table 4.8 displays the iterates as we increase the penalty from 10 to 10,000. Note that the violations of the incentive and reservation value constraints, columns "IV" and "RV" in

Table 4.8
Principal-agent problem (4.10.2)

P	(w_1, w_2)	IV	RV
10	(0.7889, 0.4605)	0.2(-1)	0.1(0)
10^2	(1.1110, 0.5356)	0.6(-2)	0.1(-1)
10^3	(1.2142, 0.5452)	0.8(-3)	0.2(-2)
10^4	(1.2289, 0.5462)	0.9(-4)	0.2(-3)
10^5	(1.2304, 0.5463)	0.9(-5)	0.2(-4)

6. We leave it as an exercise to check if $L = 1$ is the principal's choice.

table 4.8, drop in proportion to P . One could stop at the fifth iterate where $P = 10^5$, since the constraint violations are all small relative to the rest of the problem.

Note some features of the problem. First, the expected wage is \$1.0937, compared to the \$1 certainty equivalent of the worker's reservation utility. This shows that the agent bears a risk that costs about 9.4 cents to him in utility. Note that the difference between success and failure for the agent is 69 cents, whereas the difference between success and failure for the project is 2; hence the agent receives only 35% of the marginal *ex post* profit from success.

Managerial Incentives

The solution to the principal-agent problem is trivial if the agent is risk-neutral, that is, $u(c) = c$. The solution is $w(y) = y - F$; that is, the agent "buys" the firm for F and then takes the output. Economists commonly use this case as a benchmark. The extent to which this is inappropriate is illustrated by the basic literature on managerial incentives.

Jensen and Murphy (1990) found that executives earn only about \$3 per extra \$1,000 of profits. They argued that executive compensation was not as connected to profits as desirable. Haubrich (1994), on the other hand, showed that the observed sharing rules were consistent with optimal contracting and risk-averse managers. Haubrich examined a broad range of numerical examples that showed that management would receive a small share of profits when one made reasonable assumptions concerning the risk-aversion of managers, the marginal product of their effort, and the riskiness associated with their decisions. His paper is a good example of where a numerical examination of a model yields answers dramatically different from results of a less realistic model with a closed-form solution.

Efficient Outcomes with Adverse Selection

The Rothschild-Stiglitz-Wilson (RSW) model of insurance markets with adverse selection is a simple model illustrating how asymmetric information affects equilibrium and efficient allocations. In this section we use the RSW model to study efficient allocations under asymmetric information.⁷

We examine a model of risky endowments where a type H agent has a probability π^H of receiving e_1 and probability $1 - \pi^H$ of receiving e_2 where $e_1 > e_2$. Similarly type L agents have a probability π^L of receiving e_1 where $\pi^H > \pi^L$. Let θ^H (θ^L) be the fraction of type H (L) agents. These risks are independent across agents; we assume an infinite number of each type and invoke a law of large numbers.

7. Readers not familiar with this model should consult Rothschild and Stiglitz (1976) and Wilson (1977).

Suppose that a social planner wants to help these agents deal with their risks through insurance contracts. He can observe whether an individual receives e_1 or e_2 but does not know whether he is a type H or L agent. The planner's budget constraint requires that he break even. We let $y = (y_1, y_2)$ denote an agent's net state-contingent income, combining endowment and insurance transfers; that is, he pays $e_1 - y_1$ to the insurer and consumes y_1 if the endowment is e_1 , and receives a payment equal to $y_2 - e_2$ and consumes y_2 otherwise. Let

$$U^t(y^t) = \pi^t u^t(y_1^t) + (1 - \pi^t) u^t(y_2^t), \quad t = H, L,$$

be the type t expected utility function of the type t net income, y^t .

The social planner would like to give different insurance policies to agents of different type but cannot observe an agent's type. Therefore the social planner offers a menu of insurance contracts and lets each agent choose the one best for him. The planner's profits are

$$\begin{aligned} \Pi(y^H, y^L) &= \theta^H(\pi^H(e_1 - y_1^H) + (1 - \pi^H)(e_2 - y_2^H)) \\ &\quad + \theta^L(\pi^L(e_1 - y_1^L) + (1 - \pi^L)(e_2 - y_2^L)). \end{aligned} \quad (4.8.3)$$

An allocation $y^H, y^L \in R^2$ in the RSW model is *constrained efficient* if it solves a constrained optimization problem of the form

$$\begin{aligned} \max \quad & \lambda U^H(y^H) + (1 - \lambda) U^L(y^L) \\ \text{s.t.} \quad & U^H(y^H) \geq U^H(y^L), \\ & U^L(y^L) \geq U^L(y^H), \\ & \Pi(y^H, y^L) \geq 0, \end{aligned} \quad (4.8.4)$$

where $0 \leq \lambda \leq 1$ is the welfare weight of type H agents. In (4.8.4), the social planner chooses an insurance contract for each type, but he is constrained by the revelation requirement that each type will prefer the contract meant for that type over the contracts meant for other types.

We adapt a penalty function approach to this problem. In table 4.9 we display the results for $e_1 = 1$, $e_2 = 0$, $\theta^H = 0.1, 0.75$, $\pi^H = 0.8$, and $\pi^L = 0.8, 0.79, 0.7, 0.6, 0.5$. This was computed by a penalty function implementation of (4.8.4) where $P_k = 10^{1+k/2}$. Note the slow increase in P_k ; it was used because the $P_k = 10^k$ sequence did not work.

When we solve constrained optimization problems, we must keep in mind that the "solutions" often violate many of the constraints. The "IV" column gives the

Table 4.9
Adverse selection example (4.10.3)

π^L	θ^H	(y_1^H, y_2^H)	(y_1^L, y_2^L)	IV	Profit
0.80	0.1	0.8011, 0.7956	0.8000, 0.7999	-1(-7)	-8(-7)
0.79	0.1	0.8305, 0.6780	0.7900, 0.7900	-6(-8)	-4(-8)
0.70	0.1	0.8704, 0.5183	0.7000, 0.7000	-8(-8)	-1(-7)
0.60	0.1	0.8887, 0.4452	0.6000, 0.6000	-2(-6)	-1(-7)
0.50	0.1	0.9040, 0.3842	0.5000, 0.5000	-0.9(-4)	-0.2(-3)
0.80	0.75	0.8000, 0.8000	0.7995, 0.8021	-4(-7)	-5(-7)
0.79	0.75	0.7986, 0.7931	0.7974, 0.7975	-2(-6)	-4(-8)
0.70	0.75	0.7876, 0.7328	0.7699, 0.7699	-1(-7)	-8(-9)
0.60	0.75	0.7770, 0.6743	0.7306, 0.7306	-9(-8)	-5(-9)
0.50	0.75	0.6791, 0.6273	0.6862, 0.6862	-2(-7)	-1(-7)

violation in the incentive constraint, $U^L(y^L) \geq U^L(y^H)$; the other constraint is never violated. A small negative value here indicates that the bundle which the type L agents receives is slightly less than the value of y^H to them, implying that type L agents would actually choose y^H . It may seem objectionable to accept the results in table 4.9 as solutions; however, the results together with the underlying continuity of the problem's structure do indicate that there exist incentive compatible bundles near the solutions in table 4.9, and a local search would probably find some if that were desired. The "profit" column displays the firm's profit in the solution. It is always negative, indicating that the firm actually loses money. Again, however, the violations are small indicating that profitable allocations are nearby. Of course what we really need to know is that there are allocations near the "solutions" in table 4.9 that are both incentive compatible and profitable. This takes us into the constraint qualification issues which we typically ignore. We will follow standard practice here and not go through the details of actually finding the nearby feasible allocations in this problem, but one must keep in mind that perverse situations may arise in complex nonlinear constrained optimization problems.

The results do reflect the predictions of adverse selection theory. In the case where there were many fewer type H agents, each type receives an allocation that produces zero profits, and there is no cross-subsidy. When there are many more type H agents than type L , cross-subsidies arise where the numerous type H agents accept actuarially unfair allocations, but by doing so, they receive relatively risk-free allocations. Also theory predicts that type L agents always receive a risk-free allocation; this is true to four digits in all entries of table 4.9. Note how rapidly the contracts become different as we increase the gap between the two type's probabilities of a high

endowment. In particular, the type H contract goes from $(0.80, 0.80)$ when $\pi^L = 0.8$ and to $(0.83, 0.68)$ when $\pi^L = 0.79$, a substantial increase in risk even when the probability gap is only 0.01. We return to this property in chapter 15.

4.9 Computing Nash Equilibrium

In this section we will discuss an optimization approach to computing Nash equilibria introduced by McKelvey (1992). We will closely follow McKelvey's notation and presentation of the method. We assume n players. Player i has a finite strategy set $S_i = \{s_{i1}, s_{i2}, \dots, s_{iJ_i}\}$ of J_i strategies. Let $S = \prod_{i=1}^n S_i$ denote the set of all possible strategy combinations. Player i has a payoff function $M_i: S \rightarrow R$. Therefore, if player 1 plays $s_1 \in S_1$, player 2 plays $s_2 \in S_2$, and so on, then player j receives $M_j(s_1, s_2, \dots, s_n)$. A game is denoted by the pair (M, S) .

A mixed strategy for player i is modeled by a mapping $\sigma^i: S_i \rightarrow [0, 1]$ which gives the probability that player i plays pure strategy $s_i \in S_i$; hence each σ^i must satisfy $\sum_{s_i \in S_i} \sigma^i(s_i) = 1$. The joint probability of a strategy profile $s \equiv (s_1, \dots, s_n)$ will be denoted $\sigma(s) = \prod_{i=1}^n \sigma^i(s_i)$. Let Δ_i be the set of all such mixed strategies for player i , and $\Delta \equiv \prod_{i=1}^n \Delta_i$ be the set of all mixed strategy profiles.

The payoff functions, M_i must be extended over the joint mixed strategies; the extension is

$$M_i(\sigma) = \sum_{s \in S} \sigma(s) M_i(s).$$

Let $M_i(q_i, \sigma_{-i})$ be the payoff to player i from playing the mixed strategy q_i while each of the other players plays his component of σ .

DEFINITION The vector of (possibly) mixed strategies $\sigma = (\sigma^1, \dots, \sigma^n)$ is a *Nash equilibrium* for the game (M, S) if for $i = 1, \dots, n$, and all strategies $q_i \in \Delta_i$, the payoff to i of playing q_i , $M_i(q_i, \sigma_{-i})$, does not exceed the return from playing σ , $M_i(\sigma)$.

Let $M_i(s_{ij}, \sigma_{-i})$ be the payoff to i of playing their j th pure strategy, while all other players play their components of σ . To compute a Nash equilibrium, we will construct a function over Δ whose global minima corresponds exactly to the Nash equilibria of the game. Consider the function

$$v(\sigma) = \sum_{i=1}^n \sum_{s_{ij} \in S_i} \{\max[M_i(s_{ij}, \sigma_{-i}) - M_i(\sigma), 0]\}^2.$$

Obviously $v(\sigma)$ is a nonnegative function.

THEOREM 4.9.1 (McKelvey) The global minima of $v(\sigma)$ where the elements of σ are probabilities (i.e., lie between 0 and 1, inclusive, and sum to unity) are the Nash equilibria of (M, S) ; they are also the zeros of $v(\sigma)$.

Theorem 4.9.1 says that we can reduce the computation of Nash equilibrium to a constrained minimization problem; hence it is called a *Lyapunov approach*. This method has the important feature of converging to any isolated equilibrium if it starts near enough. This Lyapunov approach together with extensive search can therefore find all Nash equilibria if there are only a finite number of equilibria.

One disadvantage is that this method may get stuck at some local minimum not corresponding to any Nash equilibrium. Fortunately there is no problem in determining whether a local minimum is a Nash equilibrium, since theorem 4.9.1 assures us that σ is a Nash equilibrium if and only if $v(\sigma) = 0$. Therefore, if we stop at a nonequilibrium σ , we will know that fact because $v(\sigma) \neq 0$, and we will then try a new initial guess and resume minimizing $v(\sigma)$.

We apply McKelvey's method to the simple coordination game:

	II	
I	1, 1	0, 0
	0, 0	1, 1

We will let p_j^i denote the probability that player i plays his j th strategy in the coordination game. The equilibrium payoff for each player is $p_1^1 p_1^2 + p_2^1 p_2^2$. The Lyapunov function for this game is

$$v(p_1^1, p_2^1, p_1^2, p_2^2) = \sum_{i,j=1}^2 \max[0, p_i^j - (p_1^1 p_1^2 + p_2^1 p_2^2)]^2.$$

The three global minima (and the three equilibria) are $p = (p_1^1, p_2^1, p_1^2, p_2^2) = (1, 0, 1, 0)$, $(0.5, 0.5, 0.5, 0.5)$, and $(0, 1, 0, 1)$. To deal with the constraint that the probabilities sum to 1, we minimize the function $V(p_1^1, p_1^2) \equiv v(p_1^1, 1 - p_1^1, p_1^2, 1 - p_1^2)$ which imposes the summation condition.

Table 4.10 displays the iterates of the BFGS procedure from section 4.4 applied to $V(p_1^1, p_1^2)$, with various initial guesses. The first three columns are examples where the algorithm converges quickly to an equilibrium. The fourth column is problematic. The algorithm is stuck at $(0.25, 0.25)$, which is not an equilibrium. The reason for converging to this point is that $V(p_1^1, p_1^2)$ has a saddle point at $(0.25, 0.25)$; here

Table 4.10
Coordination game

Iterate	(p_1^1, p_1^2)	(p_1^1, p_1^2)	(p_1^1, p_1^2)	(p_1^1, p_1^2)
0	(0.1, 0.25)	(0.9, 0.2)	(0.8, 0.95)	(0.25, 0.25)
1	(0.175, 0.100)	(0.45, 0.60)	(0.959, 8.96)	(0.25, 0.25)
2	(0.110, 0.082)	(0.471, 0.561)	(0.994, 0.961)	(0.25, 0.25)
3	(0, 0)	(0.485, 0.509)	(1.00, 1.00)	(0.25, 0.25)
4	(0, 0)	(0.496, 0.502)	(1.00, 1.00)	(0.25, 0.25)
5	(0, 0)	(0.500, 0.500)	(1.00, 1.00)	(0.25, 0.25)

the gradient is zero, and V is convex in the p_1^1 and p_1^2 axes but concave along the diagonal directions. This would be detected by testing the Hessian, and it is a good example of why one should always check the second-order conditions before accepting the output from an optimization routine.

4.10 A Portfolio Problem

We next describe basic portfolio problem. Suppose that an investor must allocate his current wealth across current consumption, c , which is also the numéraire, and n assets, where one unit of asset i is currently valued at p_i and will be worth Z_i tomorrow. In this section we assume that asset 1 is a safe asset with $p_1 = 1$; then $Z_1 = R > 0$ is the safe return and $r \equiv R - 1$ is the interest rate. Let the investor's endowment of asset i be e_i , $i = 1 \dots, n$. The investor's portfolio optimization problem is

$$\begin{aligned} \max_{\omega_i} u(c) + E \left\{ u \left(\sum_{i=1}^n \omega_i Z_i \right) \right\} \\ \text{s.t. } \sum_{i=1}^n p_i (\omega_i - e_i) + c = 0. \end{aligned} \tag{4.10.1}$$

Note that (4.10.1) is an optimization problem with one equality constraint.

In order to solve (4.10.1), we need to specify the asset return distribution. This generally requires some sort of discretization. In general, we assume a finite number of possible states of the world, S , and that the final value of asset i has a finite number of possible values; specifically, state s occurs with probability π_s and $Z_i = z_i^s$ in state s . This reduces (4.10.1) to the finite, computable optimization problem

Table 4.11
Portfolio problem

a	-0.5	-1	-5.0
ω^*	(-1.41, 0.80, 1.60)	(-0.21, 0.40, 0.80)	(0.76, 0.08, 0.16)
Iterate	L^2 error		
0	1(0)	5(-1)	5(-1)
1	2(0)	9(-1)	2(-1)
2	1(-2)	7(-3)	1(-3)
3	2(-5)	3(-5)	7(-6)
4	4(-12)	2(-11)	4(-12)
5	2(-14)	2(-14)	7(-16)
cond. H	1(3)	1(3)	1(3)

$$\begin{aligned} \max_{\omega} & u(c) + \sum_{s=1}^S \pi_s u\left(\sum_{i=1}^n \omega_i z_i^s\right) \\ \text{s.t. } & \sum_{i=1}^n p_i (\omega_i - e_i) + c = 0. \end{aligned} \tag{4.10.2}$$

We illustrate these ideas in a simple, three-asset problem. We assume that $u(c) = -e^{-ac}$, $a \in \{-0.5, -1, -5\}$. Asset 1 has $Z_1 = R = 2$. We assume that Z_2 is uniformly distributed over $\{0.72, 0.92, 1.12, 1.32\}$ and that Z_3 is uniformly distributed over $\{0.86, 0.96, 1.06, 1.16\}$. The asset returns of the two risky assets are assumed to be independent random variables. Hence there are 16 equiprobable states. The initial endowment is 2, which implies that consumption is about 1 in each period which in turn implies that relative risk aversion nearly equals the absolute risk aversion coefficient, a . Table 4.11 displays the results. In each case we begin with the initial guess $\omega_i = 0.33$, $i = 1, 2, 3$.

We then display the L^2 norms of the errors of the first ten iterates. Note the rapid convergence of all three cases. The final iterates are all very close, indicating convergence. We do not display the gradients at the solutions, but they are close to machine epsilon. We also display the condition number of the Hessian at the final solution; the small condition numbers indicate that the problems are well-behaved, and the solutions are close to the true optima.

Suppose that we slightly change the example and assume that the price of each of the two risky assets is 0.5 while the price of the safe asset remains 1. In this case each risky asset returns more than the safe asset in each state of the world, and both risky assets dominates the safe asset. At these prices there are arbitrage opportunities, since

an investor with zero wealth can short the safe asset and use the proceeds to buy the risky assets and surely have positive final wealth. In this case there is no solution to (4.10.1). This shows that it is easy to construct a problem where there is no solution.

There are some ways to adjust (4.10.1) to avoid this problem. First, one could check for asset dominance, but that may not be convenient. Second, one could impose some constraints, as in

$$\begin{aligned} \max_{\omega_i} & u(c) + E \left\{ u \left(\sum_{i=1}^n \omega_i Z_i \right) \right\} \\ \text{s.t. } & \sum_{i=1}^n p_i (\omega_i - e_i) + c = 0, \\ & \omega_i \geq \bar{\omega}_i, \quad i = 1, \dots, n, \end{aligned} \tag{4.10.3}$$

where the $\bar{\omega}_i$ are often nonpositive to model shorting constraints. The new problem (4.10.3) has a solution. Unfortunately, it is a constrained optimization problem. A third idea is to smoothly penalize extreme portfolios. This can be accomplished by examining the problem

$$\begin{aligned} \max_{\omega_i} & u(c) + E \left\{ u \left(\sum_{i=1}^n \omega_i Z_i \right) \right\} - P \sum_i^n \omega_i^2 \\ \text{s.t. } & \sum_{i=1}^n p_i (\omega_i - e_i) + c = 0, \end{aligned} \tag{4.10.4}$$

where P is a positive penalty term. This is a smooth unconstrained problem and will have a solution under some conditions, in particular, when u is bounded. The penalty term is economically sensible, for it may model transaction costs which are related to total volume.

This simple portfolio problem can be generalized in several directions. There may be restrictions on the final portfolio. For example, we could add the constraint $\omega_i \geq 0$ to forbid short sales of asset i . Another restriction, often induced by tax laws or other regulations, is a maximum on the holding of asset i . Transaction costs could also be introduced into the problem, causing the budget constraint to be nonlinear.

4.11 A Simple Econometric Example

We will use a simple nonlinear least squares problem to illustrate the differences between Newton's method and the Gauss-Newton method. Consider the empirical

model

$$y = \alpha x_1 + \beta x_2 + \beta^2 x_3 + \varepsilon, \quad (4.11.1)$$

where $\varepsilon \sim N(0, 1)$. The nonlinear relation between the coefficients on x_2 and x_3 creates a nonlinear least squares problem. The data (available in the archive for this book) consists of twenty values for the dependent variable, y , and each of the independent variables, (x_1, x_2, x_3) . The values of the independent variables and the error were generated by a normal random variable with α and β both set equal to zero. The nonlinear least squares criterion with this data is

$$\begin{aligned} S(\alpha, \beta) &\equiv \sum_{i=1}^{20} (y_i - (\alpha x_1 + \beta x_2 + \beta^2 x_3))^2 \\ &= 38.5 - 5.24\alpha - 7.56\alpha^2 - 6.10\beta + 9.96\alpha\beta \\ &\quad - 3.44\beta^2 + 11.4\alpha\beta^2 + 11.6\beta^3 + 7.71\beta^4. \end{aligned} \quad (4.11.2)$$

We first illustrate the application of Newton's method to (4.11.2). The objective $S(\alpha, \beta)$ has two local minima, at $A = (-0.2194, 0.5336)$ and $B = (0.1475, -1.1134)$, plus a saddle point at $C = (0.4814, -0.3250)$; the global minimum is at A . Since the local minima and the saddlepoint satisfy the first-order conditions for a minimum, they are potential convergence points for Newton's method. The problem of multiple local minima occurs frequently in nonlinear least squares and maximum likelihood estimation. The only thing we know from theorem 4.3.1 is that if we begin with an initial guess close enough to the global minimum Newton's method will converge to the global minimum. Further analysis (see chapter 5) shows that we will converge to any point satisfying the first-order conditions if we begin close enough to it. Otherwise, we know nothing at all about the ability of Newton's method to converge, to say nothing of converging to the global minimum. In figure 4.5 we display the convergence behavior of Newton's method applied to (4.11.2). The white areas are sets of points such that if Newton's method begins there it will converge to the global minimum after ten iterations. The light gray areas are points from which Newton's method will lead to the other local minimum. The dark gray areas are points from which Newton's method will lead to the saddlepoint. The black areas are points from which Newton's method will lead, after ten iterations, to points far from any critical point.

The patterns that emerge display several important points. First, the domain of convergence for each local optimum contains a nontrivial neighborhood of the criti-

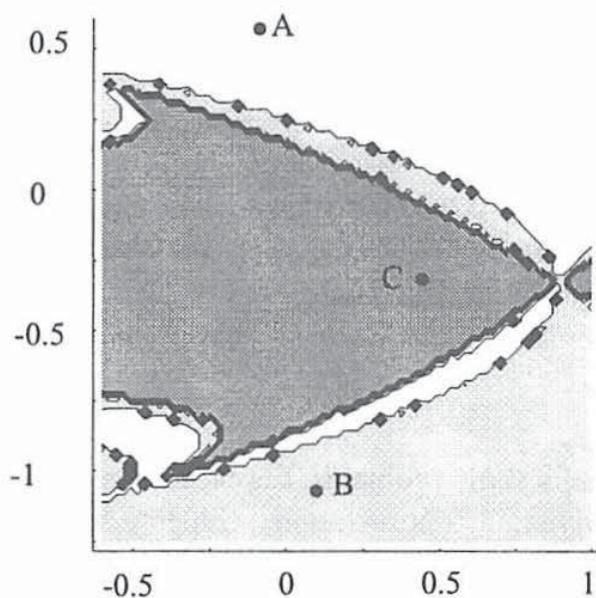


Figure 4.5
Domains of attraction for Newton's method applied to (4.11.2)

Table 4.12
Nonlinear least squares methods

Iterate	Newton	Gauss-Newton	Gauss-Newton	Gauss-Newton
0	(0, -0.5)	(0, -0.5)	(0, 1)	(0, -1.5)
1	(0.4956, -0.4191)	(0.4642, -0.7554)	(-0.6073, 0.7884)	(-0.0084, -1.2813)
2	(0.4879, -0.3201)	(0.2914, -1.0142)	(-0.3024, 0.6050)	(0.1192, -1.1524)
3	(0.4814, -0.3251)	(0.1741, -1.0914)	(-0.2297, 0.5426)	(0.1454, -1.1165)
4	(0.4814, -0.3251)	(0.1527, -1.1085)	(-0.2196, 0.5338)	(0.1475, -1.1134)
5	(0.4814, -0.3251)	(0.1487, -1.1123)	(-0.2194, 0.5336)	(0.1475, -1.1134)
6	(0.4814, -0.3251)	(0.1475, -1.1134)	(-0.2194, 0.5336)	(0.1475, -1.1134)

cal point, indicating that the closeness condition of theorem 4.3.1 is not a tightly binding restriction in practice.

Second, the regions are not connected. This is particularly the case near the boundaries of the convergence domains. Some initial guesses that cause Newton's method to converge to the global minimum are close to points that lead to the other local minimum, and some are close to points that lead nowhere. In some cases these boundary regions are actually fractal in nature, indicating the complexity and the erratic behavior of Newton's method in some cases.

We next examine the application of Gauss-Newton to solving (4.11.2). In table 4.12 we report the iterates of the Gauss-Newton method for various initial points.

Since Gauss-Newton does not have the quadratic convergence property, it is not as good as Newton in the later iterations. However, this is not important. The accuracy being attained at this point is not useful econometrically because of the nontrivial standard error.

Note that the Newton and Gauss-Newton methods may converge to different points even when they begin at the same point. Adding line search steps to these methods would improve both, and this is typically the way in which both are executed.

4.12 A Dynamic Optimization Problem

The previous examples have all been essentially static problems. Let us now examine a dynamic optimization problem. Many dynamic optimization problems require special methods because of the large number of choice variables. However, some have a special structure that can be exploited to improve speed.

We examine a simple life-cycle savings problem. Suppose that an individual lives for T periods, earns wages w_t in period t , $t = 1, \dots, T$, consumes c_t in period t , earns interest on savings per period at rate r , and has the utility function $\sum_{t=1}^T \beta^t u(c_t)$. If we define S_t to be savings at the end of period t , then $S_{t+1} = (1 + r)S_t + w_{t+1} - c_{t+1}$. The constraint $S_T = 0$ expresses the dynamic budget constraint. Let us also assume that S_0 , the initial asset level, is also zero. The substitution $c_t = S_{t-1}(1 + r) + w_t - S_t$ results in the problem

$$\max_{S_t} \sum_{t=1}^T \beta^t u(S_{t-1}(1 + r) + w_t - S_t) \equiv U(S) \quad (4.12.1)$$

$$\text{s.t. } S_T = S_0 = 0$$

a problem with $T - 1$ choice variables. This may appear intractable for large T . However, there are two ways to exploit the special structure of this problem and to efficiently solve this problem.

First, one could use the conjugate gradient procedure. This has the advantage of computing only gradients. As long as $u(c)$ is concave, $U(S)$ is concave and the conjugate gradient procedure should proceed quickly to the solution. We will illustrate this in the case where $u(c) = -e^{-c}$, $\beta = 0.9$, $T = 6$, $r = 0.2$, $w_1 = w_2 = w_3 = w_4 = 1$ and $w_5 = w_6 = 0$. This models the case where the agent earns 1 per period in the first four periods of life, followed by two periods of no wage income which models retirement. Table 4.13 shows the result of applying the conjugate gradient method to (4.12.1) with these parameters and with the initial condition $S_1 = S_2 = S_3 = S_4 =$

Table 4.13
Conjugate gradient method for (4.14.1)

Iterate	S_1	S_2	S_3	S_4	S_5
1	0.4821	1.0170	1.015	1.326	0.7515
2	0.4423	0.7916	1.135	1.406	0.7270
3	0.3578	0.7631	1.120	1.446	0.7867
4	0.3680	0.7296	1.106	1.465	0.8258
5	0.3672	0.7351	1.100	1.468	0.8318
6	0.3710	0.7369	1.103	1.468	0.8370
7	0.3721	0.7447	1.110	1.476	0.8418
8	0.3742	0.7455	1.114	1.480	0.8427
9	0.3739	0.7456	1.114	1.480	0.8425

$S_5 = 0$. Row i of table 4.13 is the value of (S_1, \dots, S_5) produced by iteration i of the conjugate gradient method, with no restarts.

The rows in table 4.13 display several points. First, the method converged at the three-digit level after seven iterations and four digits after nine iterations. Those iterations were fast ones, avoiding any Hessian computation or storage. If the objective were quadratic, we would have convergence after five iterations. The objective in (4.12.1) is not quadratic, but the conjugate gradient method still converges quickly. Second, early iterates are surprisingly good approximations of the solution. For example, the worst component of iterate 3 has only a 10 percent error. This possibility is important, for in large problems with n variables we hope to be close after fewer than n iterations.

Third, the solution displays economically important properties. At early ages the worker saves, building up his assets. These assets are then dissaved during retirement. Given the solution for S , we can compute lifetime utility and any other index of interest. More complex versions of this model are frequently used in both theoretical and empirical work to analyze life-cycle savings and consumption patterns, and the conjugate gradient method can be used in these cases also. The ease with which we can solve this simple version numerically indicates that we can add much richness to our model and still solve it numerically with ease.

We can also use Newton's method to solve (4.12.1). Since Newton-like methods require a Hessian or an approximation to the Hessian, it would appear that they are not feasible, particularly if $T = 600$. However, consider the Hessian of $U(S)$; since S_t is separated from S_{t-2}, S_{t-3}, \dots , and from S_{t+2}, S_{t+3}, \dots , in $U(S)$ the Hessian is tri-diagonal. Therefore computing a Newton step is easy, with a time and space cost of $\mathcal{O}(T)$. The key feature of (4.12.1) is that the objective is sum of functions, each of

which depends only on successive pairs of the state variable, which in turn leads to a nearly diagonal Hessian. As long as the objective is representable in a similar, nearly separable, fashion, the Hessians of more complex problems will still be block diagonal and relatively easy to store and solve. Since these Hessians have special, tractable forms, any quasi-Newton method is also feasible.

4.13 Software

There are many optimization programs available. MINPACK contains basic optimization codes. Unfortunately, there is a cost to getting the best software, particularly if one wants sophisticated constrained optimization methods. The NPSOL and MINOS optimization packages are particularly noteworthy. Both are Fortran based and are among the best general purpose optimization packages available. The input interface of NPSOL is much more convenient. NPSOL is available in the NAG library. MINOS may be better for very large problems, since it has some sparse matrix capabilities. GAMS is a modeling language that accesses many algorithms. More and Wright (1993) is a very extensive review of optimization methods and available software packages, complete with instructions as to how to acquire them. The web site <http://www.mcs.anl.gov/home/otc/Guide/SoftwareGuide/> contains links to many optimization software packages.

The exercises below encourage the reader to implement the simple penalty function approach to constrained optimization. While this will be instructive, anyone who is going to solve many such problems should definitely get one of the sophisticated optimization packages. Many readers could write acceptable unconstrained optimization code, but it is much more difficult to write good, stable, reliable code for constrained optimization.

4.14 Further Reading and Summary

Optimization lies at the heart of most economic analyses. They range from the relatively simple problems of consumer demand to complex incentive mechanism design problems, multimodal maximum likelihood problems, and large dynamic optimization problems.

The texts Bazaraa and Shetty (1979), Luenberger (1984), and Gill et al. (1981), and the surveys of Dennis and Schnabel (1989), Lemarachal (1989), and Gill, Murray, Saunders and Wright (1989) are excellent introductions to optimization theory and recent developments. The quadratic approximation idea of Newton's method and the Hessian-free approach of the conjugate gradient method are just two points on a

continuum of possibilities. This is illustrated by some recent efforts in optimization theory. The tensor work of Schnabel and Chow (1985) goes beyond the quadratic approximation by adding higher-order terms. Greenblatt (1994) applies tensor methods to some econometric problems. All the methods we examined can give us only local optima. The global optimization literature is surveyed in Kan et al. (1989) and Horst and Pardalos (1995).

Linear programming is an important part of optimization theory; the reader is presumed to be familiar with the basics. Luenberger (1984) is an excellent text and Goldfarb and Todd (1989) surveys recent developments. Of particular interest are the new methods arising from Karmarkar (1984), which dominate the simplex method for large problems.

The McKelvey method is probably not the most efficient way for computing Nash equilibrium; however, it is the easiest to present and program. Alternative approaches are outlined in Wilson (1971), Lemke and Howson (1964), and Nagurney (1993); see also the McKelvey (1996) survey.

We finish this chapter by summarizing the lessons of numerical optimization. First, one should do a grid search if the nature of the objective is not known *a priori*. The grid search may include Hessian computations at the more promising points; this will alert one to possible ill-conditioning.

Then the proper choice of algorithm depends on the nature of the problem. If the objective is rough, a polytope method is advised, along with several restarts to avoid local solutions. If a problem is smooth and small, Newton's method is advised. If a problem is smooth but of moderate size, BFGS and DFP are advised to economize on Hessian computation. For large problems, the conjugate gradient method is available. If the objective is a sum of squares, then one should use specialized least squares methods.

Constrained problems are much more difficult than unconstrained methods, and one should use well-tested software. Linear programming problems are relatively simple to solve with the simplex method. In general, it is a good idea to make the constraints as linear as possible, putting nonlinearities in the objective function. General constrained problems are difficult, but there the sequential quadratic and reduced gradient methods are good methods that combine penalty function and local approximation ideas to solve them.

Exercises

1. Use the polytope, steepest descent, Newton, Newton with line search, BFGS, and conjugate gradient methods to solve

$$\min_{x,y} 100(y - x^2)^2 + (1 - x)^2.$$

Which methods do well? poorly? Hint: Plot contours. Note that $(x, y) = (1, 1)$ is unique solution.

2. One of the classical uses of optimization in economics is the computation of the Pareto frontier. Consider the endowment economy with m goods and n agents. Assume that agent i 's utility function over the m goods is

$$u^i(x) = \sum_{j=1}^m a_j^i x_j^{v_j^i+1} (1 + v_j^i)^{-1}.$$

Suppose that agent i 's endowment of good j is e_j^i . Assume that $a_j^i, e_j^i > 0 > v_j^i$, $i = 1, \dots, n$, $j = 1, \dots, m$. (For $v_j^i = -1$, we replace $x_j^{v_j^i+1} (1 + v_j^i)^{-1}$ with $\ln x_j$.) Write a program (using the method of your choice) that will read in the v_j^i , a_j^i , and e_j^i values and the social weights $\lambda_i > 0$, and output the solution to the social planner's problem. First choose $m = n = 3$. For what preferences is convergence rapid? slow? Can your program handle $m = n = 5$? $m = n = 10$?

3. Solve the portfolio problem described in section 4.10. Table 4.11 used Newton's method. Implement the polytope, steepest descent, coordinate direction search, quasi-Newton method, BFGS, DFP, and conjugate gradient algorithms. Compare the performance of the methods.
4. Write a program for bimatrix games that implements McKelvey's Lyapunov approach to computing Nash equilibria. Compute equilibria for the following games:

Game 1

		II
		1, 1
I		1, 7
		3, 0
		5, 5
		3, 0
		2, 1
		2, 2

Game 2

		II		
		1, 1	4, 2	3, 2
I		0, 3	2, 2	4, 1
		0, 0	5, 1	3, 3
		2, 1	5, 3	3, 1
				10, 0
				4, 4

5. Use your program to solve figure 4.1.24 on page 39 and figure 4.2.5 on page 55 in Fudenberg and Tirole (1991) (FT). Generalize McKelvey's approach for subgame perfect equilibrium in two-stage games with perfect observation, and solve figure 4.3.3 in FT. Generalize this approach further to games with imperfect information and solve figures 3.4, 3.6, 3.15, 3.20, and 8.5 in FT.

6. Solve the dynamic life-cycle consumption problem (4.12.1) where $w_t = 1 + 0.2t - 0.003t^2$ in year t , $T = 50$, $\beta = 0.96$, $r = 0.06$, $u(c) = -1/c$, and t is understood to be the year. Use both conjugate gradient method, and the Newton method with tridiagonal Hessian inversion. Compare the timing. Next solve the equivalent problem with a one-month time period with both methods, and again compare performance. Next assume that utility is $u(c) + v(l)$, where l is now labor supply; let $v(l) = -l^{1+\eta}/(1+\eta)$, where $\eta = 1, 2, 10, 100$. Recompute the annual and monthly life-cycle problems.
7. Write a program that computes Pareto-efficient allocations for the RSW model. Assume that all agents have the same utility but allow n risk types. Assume CRRA utility, and use the single-crossing property when specifying the constraint set. Next, write a program that solves any Paretian social planner's problem for a generalized RSW model that allows agents to differ in their (CRRA) utility function as well as in their probability of high income. Note that single crossing will not be useful here. Which incentive compatibility constraints now bind? Vary the social planner's weights. How does that affect which constraints bind?

5 Nonlinear Equations

Concepts of economic equilibrium are often expressed as systems of nonlinear equations. Such problems generally take two forms, zeros and fixed points. If $f : R^n \rightarrow R^n$, then a *zero of f* is any x such that $f(x) = 0$, and a *fixed point* of f is any x such that $f(x) = x$. These are essentially the same problem, since x is a fixed point of $f(x)$ if and only if it is a zero of $f(x) - x$. The existence of solutions is examined in Brouwer's theorem and its extensions.¹ In this chapter we examine numerical methods for solving nonlinear equations.

The Arrow-Debreu concept of general equilibrium reduces to finding a price vector at which excess demand is zero; it is the most famous nonlinear equation problem in economics. Other examples include Nash equilibria of games with continuous strategies and the transition paths of deterministic dynamic systems. More recently, economists have solved nonlinear dynamic problems in infinite-dimensional spaces by approximating them with nonlinear equations in finite-dimensional Euclidean spaces. Nonlinear equation solving is a central component in many numerical procedures in economics.

In this chapter we examine a wide range of methods for solving nonlinear equations. We first discuss techniques for one-dimensional problems. We then turn to several methods for solving general finite-dimensional problems, including Gauss-Jacobi, Gauss-Seidel, successive approximation, Newton's method, and homotopy methods. These methods offer a variety of approaches, each with its own strengths and weaknesses. We apply these methods to problems from static general equilibrium and oligopoly theory.

5.1 One-Dimensional Problems: Bisection

We first consider the case, $f : R \rightarrow R$, of a single variable and the problem $f(x) = 0$. This special case is an important one; it is also the basis of some multidimensional routines. We use this case to simply exposit ideas that generalize to n dimensions.

Bisection

The first method we examine is bisection, which is displayed in figure 5.1. Suppose f is continuous and $f(a) < 0 < f(b)$ for some a, b , $a < b$. Under these conditions, the intermediate value theorem (IVT) tells us that there is some zero of f in (a, b) . The bisection method uses this result repeatedly to compute a zero.

Consider $c = \frac{1}{2}(a + b)$, the midpoint of $[a, b]$. If $f(c) = 0$, we are done. If $f(c) < 0$, the IVT says that there is a zero of f in (c, b) . The bisection method then continues

1. For a review of fixed-point theory, see Border (1985).

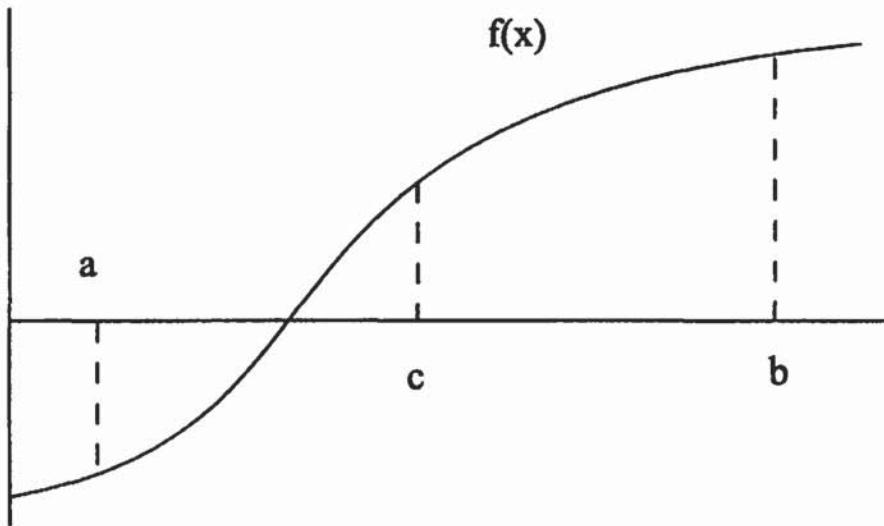


Figure 5.1
Bisection method

by focusing on the interval (c, b) . If $f(c) > 0$, as in figure 5.1, there is a zero in (a, c) , and the bisection method continues with (a, c) . In either case, the bisection method continues with a smaller interval. There could be zeros in both (a, c) and (b, c) , but our objective here is to find some zero, not all zeros. The bisection method applies this procedure repeatedly, constructing successively smaller intervals containing a zero.

Algorithm 5.1 Bisection Method

Objective: Find a zero of $f(x), f : R^1 \rightarrow R^1$.

Initialization. Initialize and bracket a zero: Find $x^L < x^R$ such that $f(x^L) \cdot f(x^R) < 0$, and choose stopping rule parameters $\varepsilon, \delta > 0$.

Step 1. Compute midpoint: $x^M = (x^L + x^R)/2$.

Step 2. Refine the bounds: If $f(x^M) \cdot f(x^L) < 0$, $x^R = x^M$ and do not change x^L ; else $x^L = x^M$ and leave x^R unchanged.

Step 3. Check stopping rule: If $x^R - x^L \leq \varepsilon(1 + |x^L| + |x^R|)$ or if $|f(x^M)| \leq \delta$, then STOP and report solution at x^M ; else go to step 1.

While the bisection algorithm is simple, it displays the important components of any nonlinear equation solver: Make an initial guess, compute iterates, and check if the last iterate is acceptable as a solution. We next address the stopping criterion of the bisection method.

Stopping Rules

Iterative schemes seldom land on the true solution. Furthermore, continuing the iteration is eventually of no value since round-off error in computing $f(x)$ will eventually dominate the differences between successive iterates. Therefore a critical component of any nonlinear equation solver is the stopping rule. A stopping rule generally computes some estimate of the distance to a solution, and then stops the iteration when that estimate is small. In the case of the bisection method, it is clear that $x^R - x^L$ is an overestimate of the distance to a solution since the zero is bracketed. Step 3 says to stop in either of two cases. First, we can stop whenever the bracketing interval is so small that we do not care about any further precision. This is controlled by the choice of ε , which need not be small. For example, if the problem is finding the equilibrium price for a market, there is little point in being more precise than finding the solution to within a penny if the equilibrium price is on the order of \$1,000.

If we want high precision, we will choose small ε , but that choice must be reasonable. Choosing $\varepsilon = 0$ is nonsense, since it is unachievable; equally pointless is choosing $\varepsilon = 10^{-20}$ on a 12-digit machine where f can be calculated with at most 12 digits of accuracy. Note also that we use a relative step size condition in step 3 of the algorithm 5.1 for stopping because the computer can really detect only relatively small intervals. If x^L and x^R are of the order 10^{10} , demanding $x^R - x^L < 10^{-5}$ would be the same as choosing $\varepsilon = 0$ on a machine that has only 12-digit precision. This consideration alone would argue for a purely relative change test of the form $x^R - x^L < \varepsilon(|x^L| + |x^R|)$. Such a test would have problems in cases where the solution is close to $x = 0$ and x^L and x^R converges to 0. The “1” in the stopping criterion in step 3 avoids this problem.

Second, we must stop when round-off errors associated with computing f make it impossible to pin down the zero any further; that is, we stop when $f(x^M)$ is less than the expected error in calculating f . This is controlled in step 3 by δ ; therefore δ should be at least the error expected in the computation of f . In some cases the computation of f may be rather complex and subject to a variety of numerical errors so that δ may be much larger than the machine precision. In other cases we may be satisfied with an x that makes $f(x)$ small but not zero. In those cases we should choose δ to be the maximal value for $f(x)$ that serves our purposes.

Convergence

If an iterative method is to be useful, it must converge to the solution and do so at a reasonable speed. Convergence properties for bisection are obvious. Bisection will

always converge to a solution once we have found an initial pair of points, x^L and x^R , that bracket a zero.

Being essentially a linearly convergent iteration, bisection is a slow method. Suppose that you want to increase the accuracy by one significant decimal digit; that is, reduce the maximum possible error, $|x_k^L - x_k^R|$, by 90 percent. Since each iteration of bisection reduces the error by only 50 percent, it takes more than three iterations to add a decimal digit of accuracy.

5.2 One-Dimensional Problems: Newton's Method

While bisection is a reliable procedure, its slow speed makes it a poor method for solving one-dimensional equations. It is also a method that assumes only continuity of f . Newton's method uses smoothness properties of f to formulate a method that is fast when it works but may not always converge. Newton's method approximates $f(x)$ by a succession of linear functions, and it approximates a zero of f with the zeros of the linear approximations. If f is smooth, these approximations are increasingly accurate, and the successive iterates will converge rapidly to a zero of f . In essence Newton's method reduces a nonlinear problem to a sequence of linear problems, where the zeros are easy to compute.

Formally Newton's method is a simple iteration. Suppose that our current guess is x_k . At x_k we construct the linear approximation to f at x_k , yielding the function $g(x) \equiv f'(x_k)(x - x_k) + f(x_k)$. The functions $g(x)$ and $f(x)$ are tangent at x_k , and generally close in the neighborhood of x_k . Instead of solving for a zero of f , we solve for a zero of g , hoping that the two functions have similar zeros. Our new guess, x_{k+1} , will be the zero of $g(x)$, implying that

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \quad (5.2.1)$$

Figure 5.2 graphically displays the steps from x_1 to x_5 . The new guess will likely not be a zero of f , but the hope is that the sequence x_k will converge to a zero of f . Theorem 2.1 provides a sufficient condition for convergence.

THEOREM 2.1 Suppose that f is C^2 and that $f(x^*) = 0$. If x_0 is sufficiently close to x^* , $f'(x^*) \neq 0$, and $|f''(x^*)/f'(x^*)| < \infty$, the Newton sequence x_k defined by (5.2.1) converges to x^* , and it is quadratically convergent, that is,

$$\limsup_{k \rightarrow \infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^2} < \infty. \quad (5.2.2)$$

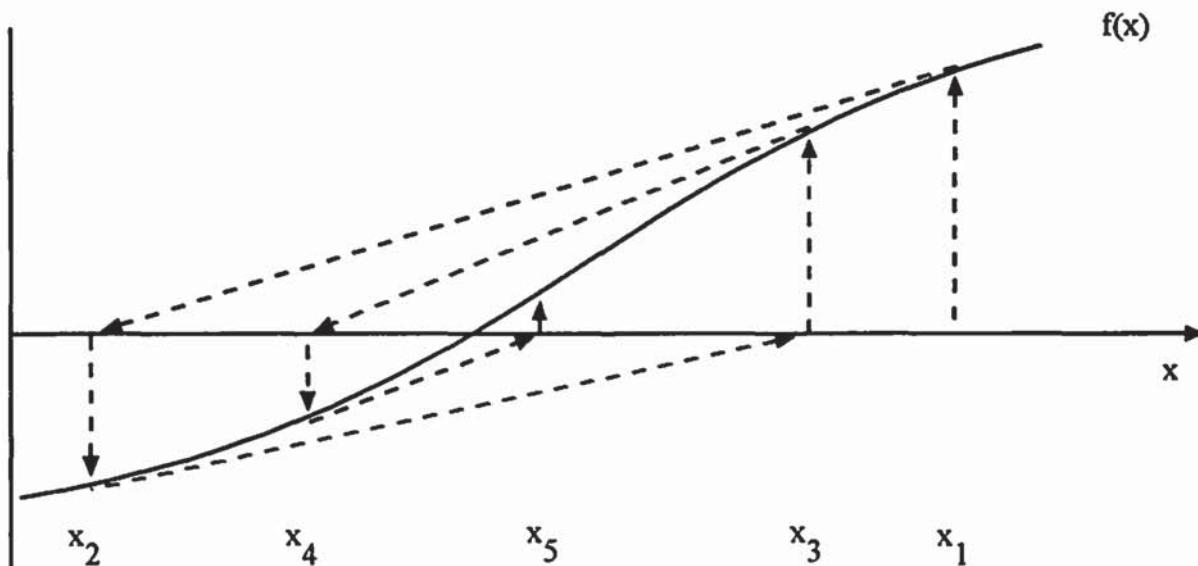


Figure 5.2
Newton method

Proof The proof of theorem 4.1.1 for Newton's method for optimization was really a proof that Newton's method for nonlinear equations converged when applied to the first-order conditions of an optimization problem. That proof applies here without change. In particular,

$$\lim_{k \rightarrow \infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^2} = \frac{1}{2} \frac{|f''(x^*)|}{|f'(x^*)|} \quad (5.2.3)$$

which is finite if $f'(x^*) \neq 0$. ■

We see the usual trade-off when we compare Newton's method with bisection. Bisection is a safe method, always converging to a zero; unfortunately, it is slow. Newton's method sacrifices reliability for a substantial gain in speed if it converges. In choosing between the two methods, one must judge the relative importance of these features and the likelihood of Newton's method not converging.

Convergence Tests

Equation (5.2.1) is just the iterative portion of Newton's method. To complete the algorithm, we must append a convergence test, particularly since Newton's method does not bracket a zero. Typically an implementation of Newton's method applies a two-stage test in the spirit of the convergence tests discussed in chapter 2. First, we ask if the last few iterations have moved much. Formally we specify an ε and

conclude that we have converged if $|x_k - x_{k-l}| < \varepsilon(1 + |x_k|)$ for $l = 1, 2, \dots, L$; often we take $L = 1$.

Second, we ask if $f(x_k)$ is “nearly” zero. More precisely we stop if $|f(x_k)| \leq \delta$ for some prespecified δ . The tightness of this criterion is governed by the choice of δ , and the considerations here are the same as the choice of δ in bisection method. The full Newton algorithm is presented in algorithm 5.2.

Algorithm 5.2 Newton’s Method

Objective: Find a zero of $f(x)$, $f : R^1 \rightarrow R^1$, $f \in C^1$.

Initialization. Choose stopping criterion ε and δ , and starting point x_0 . Set $k = 0$.

Step 1. Compute next iterate: $x_{k+1} = x_k - f(x_k)/f'(x_k)$.

Step 2. Check stopping criterion: If $|x_k - x_{k+1}| \leq \varepsilon(1 + |x_{k+1}|)$, go to step 3. Otherwise, go to step 1.

Step 3. Report results and STOP: If $|f(x_{k+1})| \leq \delta$, report success in finding a zero; otherwise, report failure.

Even if we have found a point that satisfies both the ε and δ tests, it may still not be a zero, or close to one. Consider the case of $f(x) = x^6$. Applying step 1 to x^6 implies that $x_{k+1} = \frac{5}{6}x_k$, which is a slow, linearly convergent iteration. The problem is that x^6 is flat at its zero. More generally, if a function is nearly flat at a zero, convergence can be quite slow, and loose stopping rules may stop far from the true zero.

These stopping rule issues are more important in the case of nonlinear equations than in optimization problems. In optimization problems one is often satisfied with some point x that is nearly as good as the true optimum x^* in optimizing the objective, $F(x)$. In such cases one does not care if the error $x - x^*$ is large. We are often more demanding when solving nonlinear equations, requiring the error $x - x^*$ to be small. Therefore stopping rule issues are more important.

Importance of Units

At this point we turn to an important consideration in constructing equations and implementing the stopping criterion in step 3. In step 3 we declare success as long as we have found some x such that $f(x)$ is less than δ in magnitude. This is actually an empty test, for we can always take $f(x)$, multiply it by 10^{-k} for some large k , and have a new function with the same zeros. We may now find many x such that $10^{-k}f(x) < \delta$. While we would not do this intentionally, we could easily do this accidentally.

The general point is that we need to keep track of the units we are implicitly using, and formulate the problem and stopping rules in unit-free ways. Step 3 must be

carefully formulated for it to be meaningful. For example, in computing the equilibrium price, p , of a good, let $f(p) = D(p) - S(p)$ be excess demand and assume that D and S can be computed with double precision accuracy. In this case the stopping rule in step 3 could be reformulated as $|f(p)| < \delta D(p)$ with $\delta = 10^{-8}$. This stopping rule avoids all unit problems; it stops when excess demand is small relative to total demand. Simple adjustments to step 3 like this are ideal because they avoid unit problems without affecting the algorithm and can often be implemented.

Pathological Examples

Newton's method works well when it works, but it can fail. Consider the case of $f(x) = x^{1/3}e^{-x^2}$. The unique zero of f is at $x = 0$. However, Newton's method produces (see Donovan et al. 1993 for details) the iteration

$$x_{n+1} = x_n \left(1 - \frac{3}{1 - 6x_n^2} \right) \quad (5.2.4)$$

which has two pathologies. First, for x_n small, (5.2.4) reduces to $x_{n+1} = -2x_n$, showing that (5.2.4) converges to 0 only if $x_0 = 0$ is the initial guess. For large x_n , (5.2.4) becomes $x_{n+1} = x_n(1 + 2/x_n^2)$, which diverges but slowly. In fact, for any $\varepsilon, \delta > 0$, there will ultimately be large x_n and x_{n+1} such that $|f(x_{n+1})| < \delta$ and $|x_{n+1} - x_n| < \varepsilon$; reducing δ and ε will cause Newton's method to stop at a value even farther from the true zero.

What goes wrong? The divergence of Newton's method is due to the infinite value of $f''(0)/f'(0)$. The second problem arises because the e^{-x^2} factor squashes the function at large x , leading Newton's method, (5.2.4), to "think" that it is getting close to a zero; in some sense it is, since $f(\pm\infty) = 0$.

Newton's method can also converge to a cycle. Figure 5.3 illustrates how easy it is to construct such a case. To get Newton's method to cycle between, say, -1 and 1 , we just need $f'(1) = 0.5 = f'(-1)$ and $f(1) = 1 = -f(-1)$. While these examples are contrived, their lessons should not be ignored. In particular one should be sure that $f'(x^*) \neq 0$ and $f''(x^*)$ are not too large. While this may seem trite, it is easy to write examples where some of the derivatives at the solution are very large, and the radius of convergence is small. In the less pathological case of figure 5.3, Newton's method will converge if we begin with $x \in [-0.5, 0.5]$, showing the importance of a good initial guess.

A General Equilibrium Example with the Limited Domain Problem

Our discussion has implicitly assumed that f is defined at each Newton iterate; this is not true in many economic examples. We have also not considered how multiple

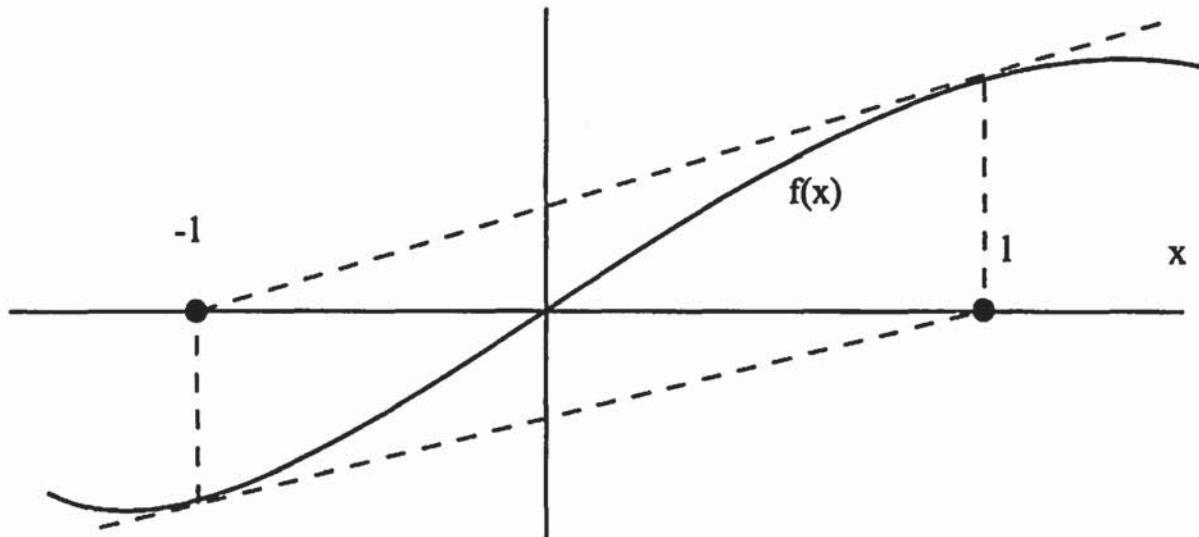


Figure 5.3
Newton method cycle

solutions affect the performance of Newton's method. We now examine both of these possibilities in the context of a simple equilibrium example.

Many economically interesting problems contain multiple solutions. This is illustrated in the following example of multiple equilibria taken from Kehoe (1991). Assume two goods and two consumers in an exchange economy. Agent i , $i = 1, 2$, has the utility function over consumption of goods 1 and 2:

$$u_i(x_1, x_2) = \frac{a_1^i x_1^{(\gamma_i+1)}}{\gamma_i + 1} + \frac{a_2^i x_2^{(\gamma_i+1)}}{\gamma_i + 1},$$

where $a_j^i \geq 0$, $\gamma_i < 0$. Let $\eta_i \equiv -1/\gamma_i$ be the constant elasticity of substitution in the utility function of agent i . If agent i has endowment $e^i \equiv (e_1^i, e_2^i)$ and the price of good j is p_j , then his demand function is $d_j^i(p) = \theta_j^i I^i p_j^{-\eta_i}$ where $I^i = p \cdot e^i$ is i 's income and $\theta_j^i \equiv (a_j^i)^{\eta_i} / \sum_{l=1}^2 (a_l^i)^{\eta_l} p_l^{(1-\eta_l)}$. Assume that $a_1^1 = a_2^1 = 1024$, $a_1^2 = a_2^2 = 1$, $e_1^1 = e_2^1 = 12$, $e_1^2 = e_2^2 = 1$, $\gamma_1 = \gamma_2 = -5$, implying that $\eta_1 = \eta_2 = 0.2$. An equilibrium is a solution to the system

$$\sum_{i=1}^2 d_1^i(p) = \sum_{i=1}^2 e_1^i, \quad p_1 + p_2 = 1, \tag{5.2.5}$$

where the first equation imposes supply equals demand for good 1 and the second equation normalizes the price vector to lie on the unit simplex. The pair of equations

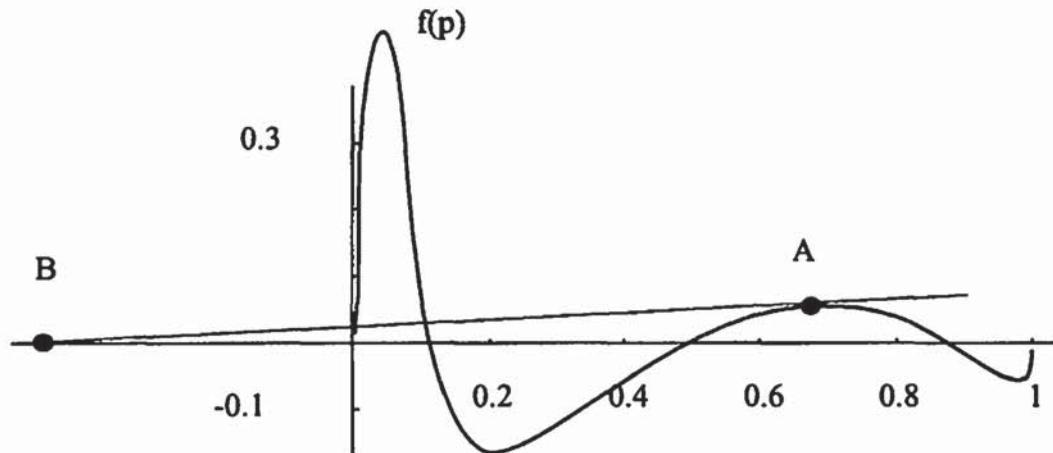


Figure 5.4
Excess demand function in (5.2.6)

in (5.2.5) suffice since Walras's law implies that supply will equal demand for good 2 at any solution to (5.2.5).

There are three equilibria to (5.2.5). They are $p^1 = (0.5, 0.5)$, the symmetric outcome, and two asymmetric equilibria at $p^2 = (0.1129, 0.8871)$ and $p^3 = (0.8871, 0.1129)$. We can reduce this problem to a one-variable problem when we make the substitution $p_2 = 1 - p_1$, yielding the problem

$$f(p_1) \equiv \sum_{i=1}^2 d_1^i(p_1, 1 - p_1) - \sum_{i=1}^2 e_1^i = 0. \quad (5.2.6)$$

The graph of f is displayed in figure 5.4.

We can use this example to examine the “dynamic” behavior of Newton’s method. Each of the three equilibria are locally stable attraction points for Newton’s method since the d_1^i have finite derivatives at each equilibrium.

However, between the equilibria Newton’s method is not so well-behaved. A serious problem arises because the excess demand function f is defined only for positive p_1 . Upon applying Newton’s method to (5.2.6), one will often experience a severe problem; Newton’s method will sometimes try to evaluate f at a negative price, where f is not defined. This *limited domain problem* is demonstrated in figure 5.4. If we began Newton’s method at A , the next guess for p_1 would be B , outside the domain of f .

There are three approaches to deal with the limited domain problem. First, one could check at each iteration whether $f(x^{k+1})$ is defined, and if it is not, move x^{k+1} toward x^k until f is defined. This will prevent a crash due to undefined numerical

operations. Unfortunately, this strategy requires access to the source code of the zero-finding routine, something that is often not possible. This new point may also generate an iterate outside the domain of f . With this strategy, one must be prepared to start over at a different initial guess.

Second, one could extend the definition of f so that it is defined at any price. Newton's method will now be globally defined. However, this is not generally easy. For Newton's method to have a good chance of getting back to the correct region, the extension should be a smooth extension of the existing function. In this case we also have the problem that demand is infinite at $p_1 = 0$, implying that any successful change must involve changing f for some region of positive p_1 values. One way to implement this *extension method* replaces f with $\tilde{f}(p_1)$, defined by

$$\tilde{f}(p_1) = \begin{cases} f(p_1), & p_1 > \varepsilon, \\ f(\varepsilon) + f'(\varepsilon)(p_1 - \varepsilon) + \frac{f''(\varepsilon)(p_1 - \varepsilon)^2}{2}, & p_1 \leq \varepsilon, \end{cases} \quad (5.2.7)$$

for some small $\varepsilon > 0$. This replaces f with a C^2 function that agrees with f at most positive prices and is defined for all prices. We may have introduced extra solutions, but if Newton's method converges to a solution with a negative price, we should just try again with a different initial guess. The trick here is to choose ε so that there are no solutions to $f(p_1) = 0$ in $(0, \varepsilon)$, the set of positive prices where $\tilde{f}(p_1) \neq f(p_1)$. Outside of this region, any positive zero of \tilde{f} is also a zero of f . This approach is easy to apply to single variable problems but not so easy to implement in multivariate problems.

Third, one can change the variable so that Newton's method stays in the appropriate region. This typically involves a nonlinear change of variables. In this case we want p_1 to stay within $[0, 1]$. If $P : R \rightarrow (0, 1)$ is C^2 , monotonically increasing, and onto, then restating the problem in terms of $z \equiv P^{-1}(p_1)$ produces a new system, $f(P(z)) = 0$, which is defined for all $z \in R$ and whose zeros match one-to-one with the zeros of $f(p_1)$. In particular, one could use $p_1 = P(z) \equiv e^z/(e^z + e^{-z})$ with the inverse map $z = (1/2) \ln(p_1/(1 - p_1))$. Newton's method applied to

$$g(z) \equiv f(P(z)) = 0 \quad (5.2.8)$$

results in the iteration

$$z_{k+1} = z_k - \frac{g(z_k)}{g'(z_k)}. \quad (5.2.9)$$

Equation (5.2.9) implies the p_1 iteration

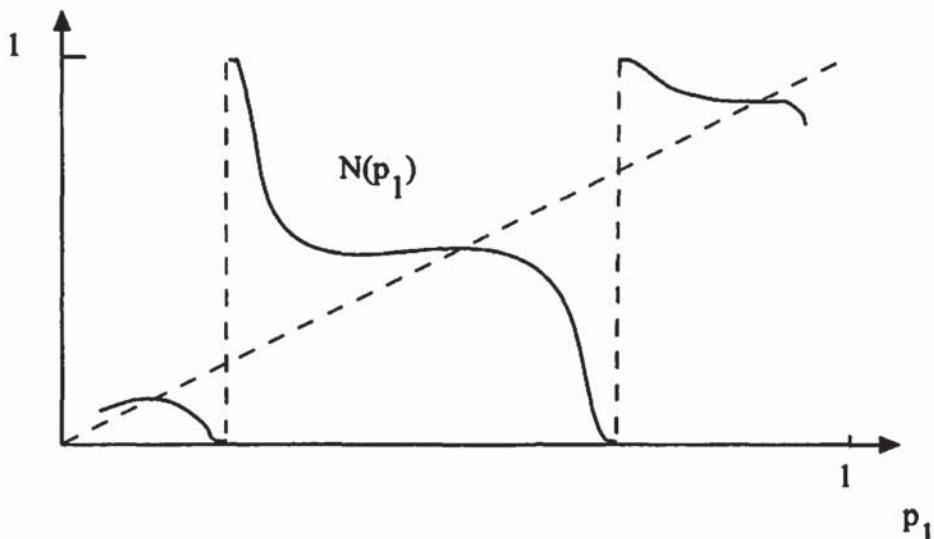


Figure 5.5
First iterate of Newton method applied to (5.2.6)

$$p_{1,k+1} = P \left(P^{-1}(p_{1,k}) - \frac{f(p_{1,k})}{f'(p_{1,k})P'(P^{-1}(p_{1,k}))} \right) \equiv N(p_{1,k}), \quad (5.2.10)$$

where each iterate is positive.

The limited domain problem may arise in many numerical contexts. We touched on it briefly in the optimization chapter. The extension approach and nonlinear transformations will also work for domain problems that arise in optimization problems. Domain problems can arise in almost any kind of numerical problem. In general, one of these tricks will be necessary to construct a reliable algorithm when some function used by the algorithm is not globally defined.

Even after transforming the variable, Newton's method can behave strangely. We plot the transformed Newton's iteration (5.2.10) as $N(p_1)$ in figure 5.5. Note that it is discontinuous near $p_1 = 0.21$ and $p_1 = 0.7$. In this case let us consider the domains of attraction for each of the solutions. Figure 5.6 displays the fourth iterate of N . Figure 5.6 shows that (5.2.10) converges to some equilibrium after four iterates for most initial guesses but that the domains of attraction are not connected. In particular, we find that (5.2.10) converges to $p_1 = 0.1129$ from $p_1 \in [0, 0.2]$ and $p_1 \in [0.67, 0.70]$ but not from most intervening initial guesses. In particular, there is no reason to trust that Newton's method will converge to the equilibrium nearest the initial guess.

These examples illustrate both the power of and problems with Newton's method. Newton's method is very useful for solving nonlinear equations, but it must be

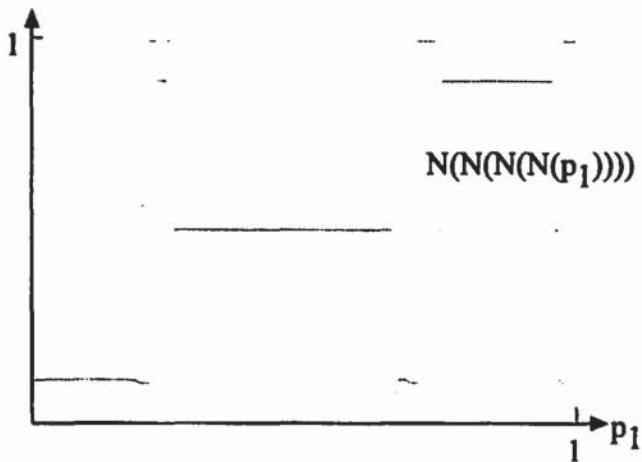


Figure 5.6
Fourth iterate of Newton method applied to (5.2.6)

applied carefully, with the user alert for the potential problems. These lessons will apply even more forcefully to the multivariate case studied below.

5.3 Special Methods for One-Dimensional Problems

We next consider two special one-dimensional methods. These do not directly generalize to multivariate problems but are quite useful in the single-variable case.

Secant Method

A key step in Newton's method is the computation of $f'(x)$, which may be costly. The *secant method* employs the idea of linear approximations but never evaluates f' . Instead, the secant method approximates $f'(x_k)$ with the slope of the secant of f between x_k and x_{k-1} , resulting in the iteration

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}. \quad (5.3.1)$$

The secant method is the same as Newton's method except that step 1 uses equation (5.3.1) to compute the next iterate. The secant method suffers the same convergence problems as Newton's method, and when it converges, convergence is slower in terms of the number of required evaluations of f because of the secant approximation to the derivative. However, the running time can be much less because the secant method never evaluates f' . The convergence rate is between linear and quadratic.

THEOREM 3.1 If $f(x^*) = 0$, $f'(x^*) \neq 0$, and $f'(x)$ and $f''(x)$ are continuous near x^* , then the secant method converges at the rate $(1 + \sqrt{5})/2$, that is

$$\limsup_{k \rightarrow \infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^{(1+\sqrt{5})/2}} < \infty. \quad (5.3.2)$$

Proof See Young and Gregory (1988, vol. 1, pp. 150ff). ■

Fixed-Point Iteration

As with linear equations, we can often rewrite nonlinear problems in ways that suggest a computational approach. In general, any fixed-point problem $x = f(x)$ suggests the iteration $x_{k+1} = f(x_k)$. Consider the problem

$$x^3 - x - 1 = 0. \quad (5.3.3)$$

Equation (5.3.3) can be rewritten in the fixed-point form $x = (x + 1)^{1/3}$, which suggests the iteration

$$x_{k+1} = (x_k + 1)^{1/3}. \quad (5.3.4)$$

The iteration (5.3.4) does converge to a solution of (5.3.3) if $x_0 = 1$. However, if we rewrite (5.3.3) as $x = x^3 - 1$, the suggested scheme

$$x_{k+1} = x_k^3 - 1 \quad (5.3.5)$$

diverges to $-\infty$ if $x_0 = 1$.

The fixed-point iteration approach to solving nonlinear equations is often useful but not generally reliable. We have focused on the more reliable methods, but that does not mean that one should ignore fixed-point iteration schemes. Also we should be flexible, looking for transformations that can turn unstable schemes into stable schemes, just as we transformed (5.3.3) into (5.3.4). In economic analysis the objective is to solve the problem $f(x) = 0$; how one finds the solution is of secondary importance.

5.4 Elementary Methods for Multivariate Nonlinear Equations

Most problems have several unknowns, requiring the use of multidimensional methods. Suppose that $f : R^n \rightarrow R^n$ and that we want to solve $f(x) = 0$, a list of n equations in n unknowns:

$$\begin{aligned} f^1(x_1, x_2, \dots, x_n) &= 0, \\ &\vdots \\ f^n(x_1, x_2, \dots, x_n) &= 0. \end{aligned} \tag{5.4.1}$$

This section examines simple methods for solving (5.4.1).

Gauss-Jacobi Algorithm

In chapter 3 we discussed Gauss-Jacobi and Gauss-Seidel iterative schemes for solving linear equations. Both of these methods can be generalized to nonlinear problems.

The simplest iteration method is the Gauss-Jacobi method. Given the known value of the k th iterate, x^k , we use the i th equation to compute the i th component of unknown x^{k+1} , the next iterate. Formally x^{k+1} is defined in terms of x^k by the equations in (5.4.2):

$$\begin{aligned} f^1(x_1^{k+1}, x_2^k, x_3^k, \dots, x_n^k) &= 0, \\ f^2(x_1^k, x_2^{k+1}, x_3^k, \dots, x_n^k) &= 0, \\ &\vdots \\ f^n(x_1^k, x_2^k, \dots, x_{n-1}^k, x_n^{k+1}) &= 0. \end{aligned} \tag{5.4.2}$$

Each equation in (5.4.2) is a single nonlinear equation with one unknown, allowing us to apply the single-equation methods presented in the previous sections. The Gauss-Jacobi algorithm procedure reduces the problem of solving for n unknowns simultaneously in n equations to that of repeatedly solving n equations with one unknown.

The Gauss-Jacobi method is affected by the indexing scheme for the variables and equations. There is no natural choice for which variable is variable 1 and which equation is equation 1. Therefore there are $n(n - 1)/2$ different Gauss-Jacobi schemes. It is difficult to determine which scheme is best, but some simple situations come to mind. For example, if some equation depends on only one unknown, then that equation should be equation 1 and that variable should be variable 1. In general, we should choose an indexing so that (5.4.2) “resembles” a nonlinear form of back-substitution.

Each step in the Gauss-Jacobi method is a nonlinear equation and is usually solved by some iterative method. There is little point in solving each one precisely, since we

must solve each equation again in the next iteration. We could just approximately solve each equation in (5.4.2). This leads to the *linear Gauss-Jacobi method*, which takes a single Newton step to approximate the components of x^{k+1} . The resulting scheme is

$$x_i^{k+1} = x_i^k - \frac{f^i(x^k)}{f_{x_i}^i(x^k)}, \quad i = 1, \dots, n. \quad (5.4.3)$$

Gauss-Seidel Algorithm

In the Gauss-Jacobi method we use the new guess of x_i , x_i^{k+1} , only after we have computed the entire vector of new values, x^{k+1} . The basic idea of the Gauss-Seidel method is to use the new guess of x_i as soon as it is available. In the general nonlinear case, this implies that given x^k , we construct x^{k+1} componentwise by solving the following one-dimensional problems in sequence:

$$\begin{aligned} f^1(x_1^{k+1}, x_2^k, x_3^k, \dots, x_n^k) &= 0, \\ f^2(x_1^{k+1}, x_2^{k+1}, x_3^k, \dots, x_n^k) &= 0, \\ &\vdots \\ f^{n-1}(x_1^{k+1}, \dots, x_{n-2}^{k+1}, x_{n-1}^{k+1}, x_n^k) &= 0, \\ f^n(x_1^{k+1}, \dots, x_{n-2}^{k+1}, x_{n-1}^{k+1}, x_n^{k+1}) &= 0. \end{aligned} \quad (5.4.4)$$

Again we solve f^1, f^2, \dots, f^n in sequence, but we immediately use each new component. Now the indexing scheme matters even more because it affects the way in which later results depend on earlier ones. We can also implement a *linear Gauss-Seidel* method

$$x_i^{k+1} = x_i^k - \left(\frac{f^i}{f_{x_i}^i} \right) (x_1^{k+1}, \dots, x_{i-1}^{k+1}, x_i^k, \dots, x_n^k), \quad i = 1, \dots, n, \quad (5.4.5)$$

to economize on computation costs at each iteration.

Gaussian methods are often used, but they have some problems. As in the linear case they are risky methods to use if the system is not diagonally dominant. One can apply extrapolation and acceleration methods to attain or accelerate convergence just as with linear equations. However, convergence is at best linear, and the discussion of convergence in chapter 3 applies here. The key fact for any iterative scheme

$x^{k+1} = G(x^k)$ is that the spectral radius of the Jacobian evaluated at the solution, $G_x(x^*)$, is its asymptotic linear rate of convergence.

Stopping Rule Problems for Multivariate Systems

We need to construct stopping rules for multivariate algorithms. We can use the same ideas we used for univariate problems, but implementing them is more difficult. We first need a rule for stopping. If we are using an iterative scheme $x^{k+1} = G(x^k)$ (e.g., Gauss-Jacobi) and we want to stop when $\|x^k - x^*\| < \varepsilon$, we must at least continue until $\|x^{k+1} - x^k\| \leq (1 - \beta)\varepsilon$ where $\beta = \rho(G_x(x^*))$. Computing β directly would be impractical. To implement this test, we could estimate β with

$$\hat{\beta} = \max \left\{ \frac{\|x^{k-j+1} - x^k\|}{\|x^{k-j} - x^k\|} \mid j = 1, \dots, L \right\}$$

for some L . The estimate $\hat{\beta}$ should be close to $\rho(G_x(x^*))$ if $x^k \approx x^*$ and allow us to use the stopping rule $\|x^{k+1} - x^k\| \leq (1 - \hat{\beta})\varepsilon$.

Even if x^k satisfies our stopping rule, we want to check that $f(x^k)$ is close to zero. A natural way to implement this is to require that $\|f(x^k)\| \leq \delta$ for some small δ . As with univariate problems, we must be careful about units. Moreover the test $\|f(x^k)\| \leq \delta$ implicitly makes trade-offs across the different equations. For example, some equations in f may be closer to zero at x^k than they are at x^{k+1} , but a stopping rule will choose x^{k+1} over x^k if $\|f(x^{k+1})\| \leq \delta \leq \|f(x^k)\|$. This would not happen if $\|\cdot\|$ is the supremum norm but could happen if one chooses the euclidean norm for $\|\cdot\|$. In either case the system f must be constructed so that δ is a reasonable approximation to zero for each component function.

In summary, we stop if $\|x^{k+1} - x^k\| < \varepsilon$, and we accept x^{k+1} as the solution if $\|f(x^{k+1})\| < \delta$ where $\delta > 0$ and $\varepsilon > 0$ are our stopping criterion parameters. Due to standard round-off problems, δ and ε should be about the square root of the error in computing f . In practice, nonlinear equation solvers give the user several options governing the stopping criterion to be used.

A Duopoly Example

We will use a two-player duopoly model to illustrate various multivariate methods for solving equations. We assume that there are two goods, Y and Z , and that the utility function over those goods and money M is

$$\begin{aligned} U(Y, Z) &= u(Y, Z) + M \\ &= (1 + Y^\alpha + Z^\alpha)^{\eta/\alpha} + M. \end{aligned}$$

with $\alpha = 0.999$ and $\eta = 0.2$. We assume that the unit cost of Y is 0.07 and of Z is 0.08.

The Cournot duopoly game assumes that each firm simultaneously chooses its output, and each attempts to maximize its profits. If the firms choose outputs Y and Z , then $u_Y(Y, Z)$ is the price of Y and $u_Z(Y, Z)$ is the price of Z . Profit for the Y firm is $\Pi^Y(Y, Z) \equiv Y(u_Y(Y, Z) - 0.07)$ and for the Z firm is $\Pi^Z(Y, Z) \equiv Z(u_Z(Y, Z) - 0.08)$. Equilibrium is any pair of quantity choices (X, Y) that solves the problems

$$Y \in \arg \max_Y \Pi^Y(Y, Z),$$

$$Z \in \arg \max_Z \Pi^Z(Y, Z).$$

In practice, we compute equilibrium by finding (Y, Z) which solves each firm's first-order condition; that is, we solve the system

$$\Pi_1^Y(Y, Z) = 0, \tag{5.4.6a}$$

$$\Pi_2^Z(Y, Z) = 0. \tag{5.4.6b}$$

We can check the second-order conditions of each firm by checking

$$\Pi_{11}^Y(Y, Z), \Pi_{22}^Z(Y, Z) < 0,$$

but strictly speaking, we should check that each firm's choice is a global optimum of its problem.

This problem has the limited domain problem that utility is not defined for negative Y or Z . Since we need to keep Y and Z positive, we instead solve the system

$$\Pi_1^Y(e^y, e^z) = 0, \tag{5.4.7a}$$

$$\Pi_2^Z(e^y, e^z) = 0, \tag{5.4.7b}$$

for $y \equiv \ln Y$ and $z \equiv \ln Z$.

The game-theoretic examination of this problem often focuses on the reaction functions. Figure 5.7 displays the reaction function, $R_Y(z)$, which expresses the Y firm's choice of y if it expects the Z firm to choose log output equal to z ; symmetrically, $R_Z(y)$ is the reaction function of firm Z to firm Y 's choice of log output y . There is a unique Cournot-Nash equilibrium at the intersection of the reaction curves, $(y^*, z^*) = (-0.137, -0.576)$, or, equivalently, $(Y^*, Z^*) = (0.87, 0.56)$.

We will use the duopoly problem expressed in logarithmic terms, (5.4.7), to illustrate several iterative methods. Table 5.1 compares the performance of several Gaussian

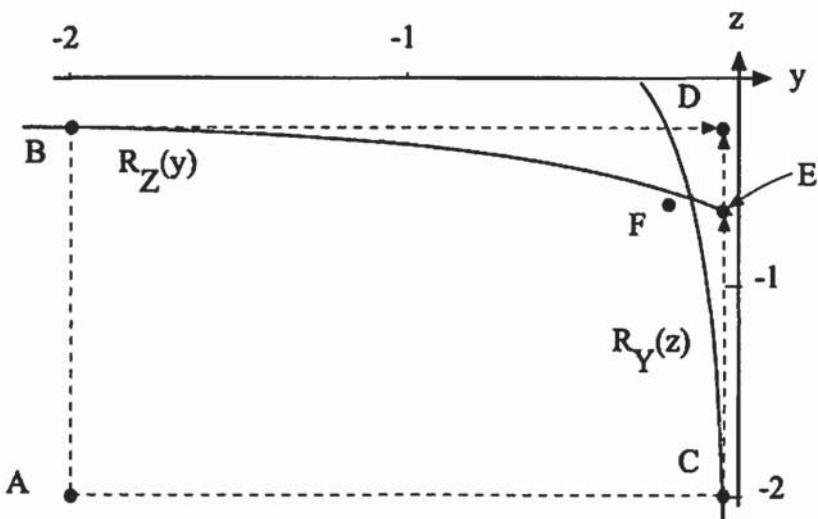


Figure 5.7
Solving the duopoly problem (5.4.7)

Table 5.1
Errors of Gaussian methods applied to (5.4.7)

Iteration	Gauss-Jacobi	Linear Gauss-Jacobi	Gauss-Seidel	Linear Gauss-Seidel
1	(1(-1), 3(-1))	(1(0), 1(0))	(1(-1), -6(-2))	(1(0), -8(0))
2	(-7(-2), -6(-2))	(-8(-1), -1(-1))	(1(-3), -6(-3))	(-6(-1), 4(2))
3	(1(-2), 4(-2))	(2(-1), 1(-1))	(1(-4), -6(-4))	*
4	(-7(-3), -6(-3))	(-8(-1), -2(-1))	(1(-5), -6(-5))	*
5	(1(-3), 4(-3))	(1(-1), 3(-1))	(9(-6), -5(-6))	*
6	(-7(-4), -5(-4))	(-5(-2), -6(-2))	(9(-7), -5(-7))	*
7	(9(-5), 4(-4))	(9(-3), 3(-2))	(8(-8), -5(-8))	*
8	(-6(-5), -5(-5))	(-5(-3), -5(-3))	(8(-9), -5(-9))	*

Note: The * means that the iterates became infinite.

methods. For example, if we use the initial guess $(y_0, z_0) = (-2, -2)$, point A in figure 5.7, the Gauss-Jacobi method generates the sequence of errors in column 2 of table 5.1; that is, the entry in row i , column 2 of table 5.1 equals $(y_i, z_i) - (y^*, z^*)$. In figure 5.7 the Gauss-Jacobi method computes C to be firm Y 's reaction to firm Z 's choice at A , and computes B to be Z 's reaction to firm Y being at A . The Gauss-Jacobi method combines these reactions to take us to D . The next iteration of Gauss-Jacobi is F . The result is a slowly convergent sequence.

The Gauss-Seidel procedure is the more familiar alternating reaction sequence used in game theory. If we begin at A , then firm Y 's reaction follows the broken line in figure 5.7 to $y = R_Y(Z)$ at C , and then firm Z 's reaction to the situation at C follows

the broken line to point E on R_Z . The amount of computation that takes us to E using Gauss-Seidel equals the computational effort in Gauss-Jacobi getting to D , an inferior approximation of equilibrium. Column 4 of table 5.1 contains the errors of the Gauss-Seidel iterates applied to (5.4.7). We see here that the Gauss-Seidel method converges more rapidly than Gauss-Jacobi.

The linearized methods display mixed performance. The linearized Gauss-Jacobi method converges relatively slowly, but the work per iteration is much less than with either Gauss-Jacobi or Gauss-Seidel. We also see that the linear Gauss-Seidel method blows up. The linearized Gauss-Seidel iterates do not keep us on the firms' reaction curves, for this method computes only the first step in a Newton computation of that reaction. Since the profit functions are significantly nonquadratic, this first step is not adequate, and the linear Gauss-Seidel method diverges even though the Gauss-Seidel method converges.

Fixed-Point Iteration

The simplest iterative procedure for solving the fixed point problem $x = f(x)$ is

$$x^{k+1} = f(x^k). \quad (5.4.8)$$

This process is called fixed-point iteration; it is also known as *successive approximation*, *successive substitution*, or *function iteration*. The process (5.4.8) generalizes to nonlinear equation systems the fixed-point iteration method discussed in section 3.6.

For a special class of functions, fixed-point iteration will work well.

DEFINITION A *differentiable contraction map* on D is any $C^1 f : D \rightarrow R^n$ defined on a closed, bounded, convex set $D \subset R^n$ such that

1. $f(D) \subset D$, and
2. $\max_{x \in D} \|J(x)\|_\infty < 1$ where $J(x)$ is the Jacobian of f .

We have an exceptionally strong constructive existence theory for the fixed points of contraction mappings.

THEOREM 5.4.1 (Contraction mapping theorem) If f is a differentiable contraction map on D , then

1. the fixed point problem $x = f(x)$ has a unique solution, $x^* \in D$;
2. the sequence defined by $x^{k+1} = f(x^k)$ converges to x^* ;
3. there is a sequence $\varepsilon_k \rightarrow 0$ such that

$$\|x^* - x^{k+1}\|_\infty \leq (\|J(x^*)\|_\infty + \varepsilon_k) \|x^* - x^k\|_\infty.$$

The contraction mapping theorem uses a contraction condition to prove convergence of our simple iterative scheme (5.4.8), and shows that in the limit, fixed-point iteration converges linearly at the rate $\|J(x^*)\|_\infty$. If $D = R^n$, then we have a global contraction condition for global convergence and uniqueness. Theorem 5.4.2 is also a useful local result.

THEOREM 5.4.2 Suppose that $f(x^*) = x^*$ and is Lipschitz at x^* . If $\rho(J(x^*)) < 1$, then for x^0 sufficiently close to x^* , $x^{k+1} = f(x^k)$ is convergent.

Theorem 5.4.2 says that if the Jacobian at a fixed point is a contraction when viewed as a linear map in R^n , then iterating f will converge if the initial guess is good. The intuition is clear: Near a zero, a smooth function f is very nearly like its Jacobian at the zero and will inherit the iteration properties of the Jacobian. Furthermore a linear map is a contraction if and only if its spectral radius is less than 1. Hence the critical condition is $\rho(J(x^*)) < 1$. The convergence rate indicated in theorem 5.4.1 is linear. This is a lower bound on the rate of convergence, since fixed-point iteration methods may converge more rapidly. In fact Newton's method can be thought of as a successive approximation method, and it converges quadratically.

As with linear equations (see chapter 3) there are ways to accelerate convergence rates of convergent methods, even contraction mapping iterations, or stabilize unstable implementations of the fixed-point iteration method. The basic method is again *extrapolation*, where we add the parameter ω and form the system

$$x^{k+1} = \omega f(x^k) + (1 - \omega)x^k. \quad (5.4.9)$$

If the original system (5.4.8) is unstable, we try $\omega \in (0, 1)$ in (5.4.9) hoping to dampen the instabilities. If the original system (5.4.8) is converging too slowly, we try some $\omega > 1$ hoping to accelerate convergence.

We next apply fixed-point iteration to our duopoly example. Our example system (5.4.7) is written as a zero problem, but the fixed-point problem

$$\Pi_1^Y(e^y, e^z) + y = y,$$

$$\Pi_2^Z(e^y, e^z) + z = z,$$

is equivalent. Fixed-point iteration generates the sequence

$$\begin{aligned} y_{k+1} &= \Pi_1^Y(e^{y_k}, e^{z_k}) + y_k, \\ z_{k+1} &= \Pi_2^Z(e^{y_k}, e^{z_k}) + z_k. \end{aligned} \quad (5.4.10)$$

Column 2 in table 5.2 displays the errors, $(y, z) - (y^*, z^*)$, of the iterates of fixed-point iteration applied to (5.4.7).

Table 5.2
Errors of fixed-point iteration in (5.4.11)

$\omega = 1.0$		$\omega = 0.9$	
Iterate	Error	Iterate	Error
20	(-0.74(0), -0.47(0))	5	(-0.11(0), 0.42(-2))
40	(-0.31(0), -0.14(0))	10	(-0.14(-1), 0.20(-1))
60	(-0.13(0), -0.25(0))	15	(-0.31(-2), 0.67(-2))
80	(-0.62(-1), 0.87(-2))	20	(-0.82(-3), 0.20(-2))
100	(-0.30(-1), 0.15(-1))	25	(-0.23(-3), 0.57(-3))

We can use extrapolation to improve the performance of (5.4.10). We introduce the parameter ω , form the system

$$\begin{aligned}y_{k+1} &= \omega(\Pi_1^Y(e^{y_k}, e^{z_k}) + y_k) + (1 - \omega)y_k, \\z_{k+1} &= \omega(\Pi_2^Z(e^{y_k}, e^{z_k}) + z_k) + (1 - \omega)z_k,\end{aligned}\tag{5.4.11}$$

and hope that for some ω , (5.4.11) will converge. Column 4 in table 5.2 contains the errors of the iterates of (5.4.11) for $\omega = 0.9$.

Fixed-point iteration may appear to be of limited usefulness due to the strong contraction conditions necessary for convergence. However, it forms the basis of many methods. The basic idea is to find another function that has the same zeros as f but for which fixed-point iteration converges. If possible, we find such a function that satisfies the conditions of theorems 5.4.2 or 5.4.3 and results in rapid convergence. We saw this in chapter 3 when we discussed stabilization and acceleration methods, and we will frequently return to this below when we examine applications. The most famous application of this idea is Newton's method, to which we now turn.

5.5 Newton's Method for Multivariate Equations

In Newton's method to solve (5.4.1), we replace f with a linear approximation, and then solve the linear problem to generate the next guess, just as we did in the one-dimensional case. By Taylor's theorem, the linear approximation of f around the initial guess x^0 is $f(x) \doteq f(x^0) + J(x^0)(x - x^0)$. We can solve for the zero of this linear approximation, yielding $x^1 = x^0 - J(x^0)^{-1}f(x^0)$. This zero then serves as the new guess around which we again linearize. The Newton iteration scheme is

$$x^{k+1} = x^k - J(x^k)^{-1}f(x^k)\tag{5.5.1}$$

THEOREM 5.5.1 If $f(x^*) = 0$, $\det(J(x^*)) \neq 0$, and $J(x)$ is Lipschitz near x^* , then for x^0 near x^* , the sequence defined by (5.5.1) satisfies

$$\lim_{k \rightarrow \infty} \frac{\|x^{k+1} - x^*\|}{\|x^k - x^*\|^2} < \infty. \quad (5.5.2)$$

As with the one-dimensional case, Newton's method is quadratically convergent. The critical assumption is that $\det(J(x^*)) \neq 0$. The basic approach of the proof² is to show that the map $g(x) = x - J(x)^{-1}f(x)$ is a contraction near x^* .

Algorithm 5.3 Newton's Method for Multivariate Equations

Initialization. Choose stopping criterion ε and δ , and starting point x^0 . Set $k = 0$.

Step 1. Compute next iterate: Compute Jacobian $A_k = J(x^k)$, solve $A_k s^k = -f(x^k)$ for s^k , and set $x^{k+1} = x^k + s^k$.

Step 2. Check stopping criterion: If $\|x^k - x^{k+1}\| \leq \varepsilon(1 + \|x^{k+1}\|)$, go to step 3. Otherwise, go to step 1.

Step 3. STOP and report result: If $\|f(x^{k+1})\| \leq \delta$, report success in finding a zero; otherwise, report failure.

We again emphasize that f must be carefully defined if step 3 is to be meaningful. All the comments in section 5.2 concerning units apply here. One will typically use a package to solve $f(x) = 0$ and the user will define f to facilitate convergence. In such cases the package will apply a version of step 3 to that definition of f , which may not be the appropriate unit-free test; in that case, one applies step 3 to a unit-free version of the problem after the package has produced a candidate solution.

The rapid convergence of Newton's method is illustrated by our example system (5.4.7). Column 2 of table 5.3 contains errors, $(y_k, z_k) - (y^*, z^*)$, of the Newton iterates beginning at $(-2, -2)$. Note the rapid convergence, with each iteration doubling the number of accurate digits.

Secant Method (Broyden)

One would typically use finite differences to compute $J(x)$ in Newton's method but that requires n^2 evaluations of f . Explicit computation of the Jacobian is often costly to compute and code. One solution to the problem of Jacobian calculation is to begin with a rough guess of the Jacobian and use the successive evaluations of f and its

2. See, for example, Ortega and Rheinboldt (1970).

Table 5.3
Errors of Newton and Broyden methods applied to (5.4.7)

Iterate k	Newton	Broyden
0	(-0.19(1), -0.14(1))	(-0.19(1), -0.14(1))
1	(0.55(0), 0.28(0))	(0.55(0), 0.28(0))
2	(-0.59(-1), 0.93(-2))	(0.14(-1), 0.65(-2))
3	(0.15(-3), 0.81(-3))	(-0.19(-2), 0.40(-3))
4	(0.86(-8), 0.54(-7))	(0.45(-3), 0.24(-3))
5	(0.80(-15), 0.44(-15))	(-0.11(-3), 0.61(-4))
6	(0, 0)	(0.26(-4), -0.14(-4))
7	(0, 0)	(-0.60(-5), 0.33(-5))
8	(0, 0)	(0.14(-5), -0.76(-6))
9	(0, 0)	(-0.32(-6), 0.18(-6))
10	(0, 0)	(0.75(-7), 0.41(-7))

gradient to update the guess of J . In this way we make more use of the computed values of f and avoid the cost of recomputing J at each iteration.

The one-dimensional secant method did this, but the problem in n dimensions is more complex. To see this, suppose that we have computed f at y and z . In the one-dimensional case the slope, m , near y and z is approximated by the solution to the scalar equation $f(y) - f(z) = m(y - z)$, which is unique whenever $y \neq z$. The n -dimensional analogue to the slope, m , is the Jacobian, M , which near y and z approximately satisfies the multidimensional secantlike equation $f(y) - f(z) = M(y - z)$. There is no unique such matrix: Since $f(y) - f(z)$ and $y - z$ are column vectors, this equation imposes only n conditions on the n^2 elements of M . We need some way to fill in the rest of our estimate of M .

Broyden's method is the R^n version of the secant method. It produces a sequence of points x^k , and matrices A_k which serve as Jacobian guesses. Suppose that after k iterations our guess for x is x^k and our guess for the Jacobian at x^k is A_k . We use A_k to compute the Newton step; that is, we solve $A_k s^k = -f(x^k)$ for s^k , and define $x^{k+1} = x^k + s^k$. We next compute A_{k+1} to be our new approximation to $J(x^{k+1})$. If J is continuous, $J(x^{k+1})$ should be close to $J(x^k)$, which was approximated by A_k . The Broyden idea is to choose A_{k+1} consistent with the secant equation $f(x^{k+1}) - f(x^k) = A_{k+1}s^k$ but keep it as "close" to A_k as possible, where by "close" we refer to how it acts as a linear map. The basic property of the Jacobian is that $J(x)q$ should approximate $\delta(q, x) \equiv f(x + q) - f(x)$ for all directions q . For the specific direction $q = s^k$ we know $\delta(q, x^k) = f(x^{k+1}) - f(x^k)$. However, for any direction q orthogonal to s^k we have no information about $\delta(q, x^k)$. The Broyden method fixes A_{k+1} by

assuming that $A_{k+1}q = A_kq$ whenever $q^\top s^k = 0$; that is, the predicted change in directions orthogonal to s^k under the new Jacobian guess, A_{k+1} , should equal the predicted change under the old Jacobian guess, A_k . Together with the secant condition this uniquely determines A_{k+1} to be

$$A_{k+1} = A_k + \frac{(y_k - A_k s^k)(s^k)^\top}{(s^k)^\top s^k},$$

where $y_k \equiv f(x^{k+1}) - f(x^k)$. Broyden discusses some alternative choices of A_{k+1} , but this one has been the most successful in practice. Algorithm 5.4 describes Broyden's method.

Algorithm 5.4 Broyden's Method

Initialization. Choose stopping criterion ε and δ , starting point x^0 , and initial Jacobian guess $A_0 = I$. Set $k = 0$.

Step 1. Compute next iterate: Solve $A_k s^k = -f(x^k)$ for s^k , and set $x^{k+1} = x^k + s^k$.

Step 2. Update Jacobian guess: Set $y_k = f(x^{k+1}) - f(x^k)$ and

$$A_{k+1} = A_k + \frac{(y_k - A_k s^k)(s^k)^\top}{(s^k)^\top s^k}.$$

Step 3. Check stopping criterion: If $\|x^k - x^{k+1}\| \leq \varepsilon(1 + \|x^{k+1}\|)$, go to step 4. Otherwise, go to step 1.

Step 4. STOP and report result: If $\|f(x^{k+1})\| \leq \delta$, report success in finding a zero; otherwise, report failure.

The algorithm will stop when $f(x^k)$ is close to zero, or when the steps, s^k , become small. The last stopping condition is particularly desirable here, since the division in the computation of A_{k+1} will be problematic if s^k is small.

The convergence properties of Broyden's method are inferior to Newton's method, but still good.

THEOREM 5.5.3 There exists $\varepsilon > 0$ such that if $\|x^0 - x^*\| < \varepsilon$ and $\|A_0 - J(x^*)\| < \varepsilon$, then the Broyden method converges superlinearly.

Note that convergence is only asserted for the x sequence. The A sequence need not converge to the $J(x^*)$. Each iteration of the Broyden method is far less costly to compute because there is no Jacobian calculated, but the Broyden method will generally need more iterations than Newton's method. For large systems, the Broyden method may be much faster, since Jacobian calculation can be very expensive. For

highly nonlinear problems the Jacobian may change drastically between iterations, causing the Broyden approximation to be quite poor. Of course that will also give Newton's method problems since the underlying assumption of any Newton method is that a linear approximation is appropriate, which is the same as saying that the Jacobian does not change much.

These features of the Broyden method is illustrated by our example system (5.4.7). Column 3 of table 5.3 contains the errors of the Broyden method. Note that convergence is slower than Newton's method, in terms of iterations required, but much faster than the Gaussian or fixed-point iteration methods.

Solving Multiple Problems

Sometimes we will want to solve several problems of the form $f(x; a) = 0$ for x where a is a vector of parameters. The choice of initial conditions substantially affects the performance of any algorithm. In the case of Broyden's method, we must also make an initial guess for the Jacobian. When we solve several problems, we can use the same hot start idea discussed in chapter 4 for optimization problems. In Newton's method the solution to $f(x; a) = 0$ will be a good initial guess when solving $f(x; a') = 0$ if $a' \approx a$. If we are using Broyden's method, we will want to use both the final solution and the final Jacobian estimate for $f(x; a) = 0$ as the initial guesses when solving $f(x; a') = 0$. As with optimization problems, hot starts can produce significant economies of scale when we want to solve several problems.

5.6 Methods That Enhance Global Convergence

Outside of the special case of contraction mappings, none of these methods is globally convergent. As we saw in the one-dimensional case, it is easy to construct cases where Newton's method cycles. There are systematic ways to enhance convergence to a zero that combine minimization ideas with the nonlinear equation approach.

Optimization Ideas and Nonlinear Equations

There are several strong connections between optimization problems and nonlinear equations. First, one can often be transformed into the other. For example, if $f(x)$ is C^2 , then the solution to $\min_x f(x)$ is also a solution to the system of first-order conditions $\nabla f(x) = 0$. Newton's method for optimization in fact solves minimization problems by solving the first-order conditions.

We can also go in the opposite direction, converting a set of nonlinear equations into an optimization problem. Sometimes there is a function $F(x)$ such that $f(x) =$

$\nabla F(x)$, in which case the zeros of f are exactly the local minima of F . Such systems $f(x)$ are called *integrable*. Since we have globally convergent schemes for minimizing F we can use them to compute the zeros of f . In some economic contexts we can approach a problem in this fashion. For example, the real business cycle literature frequently computes dynamic equilibria by solving the equivalent social planner's problem. However, this approach is limited in its applicability because few systems $f(x)$ are integrable.

In one general sense any nonlinear equation problem can be converted to an optimization problem. Any solution to the system $f(x) = 0$ is also a global solution to

$$\min_x \sum_{i=1}^n f^i(x)^2, \quad (5.6.1)$$

and any global minimum of $\sum_{i=1}^n f^i(x)^2$ is a solution to $f(x) = 0$. While this approach is tempting, it has several problems. First, it is a global minimum problem because we are interested only in values of x that makes the objective in (5.6.1) zero. There may be local minima that are not near any solution to $f(x) = 0$.

Second, the condition number of the Hessian of (5.6.1) is roughly the square of the condition number of the Jacobian of $f(x) = 0$ at any solution. To see this, consider using (5.6.1) to solve the linear problem $Ax - b = 0$. The objective in (5.6.1) would become $x^T A^T Ax - b^T Ax - x^T A^T b + b^T b$, a function of x with a Hessian equal to $A^T A$. Since the spectral condition number of $A^T A$ is the square of the spectral condition number of A (or equivalently, the log condition number is doubled), the minimization problem is poorly conditioned relative to the matrix A . For example, if $\text{cond}_*(A) = 10^8$, the linear equation problem may be well-behaved, but the objective in (5.6.1) has a log condition number of 16, making it practically singular. This approach is used sometimes when A is sparse and positive definite, since then conjugate gradient methods can be used. Outside of small problems and special circumstances, the simple minimization approach in (5.6.1) is not a good method by itself.

There are ways in which solving (5.6.1) may assist. It is often difficult to find good initial guesses for a system of nonlinear equations, whereas optimization problems are not as sensitive to initial guesses. Furthermore the objective (5.6.1) may not be ill-conditioned away from a minimum. One way to find a reasonable guess is to solve (5.6.1) with a loose stopping rule. Hopefully the minimization process will move us toward a solution, but the loose stopping rule keeps us from wasting time dealing with poorly conditioned Hessians. Even with a loose stopping criterion, the outputted value for x produces a small value for $\sum_{i=1}^n f^i(x)^2$; therefore each component $f^i(x)$ is small, and x has a good chance of being a good initial guess for a conventional nonlinear solver.

Another situation where the minimization approach (5.6.1) may be useful is where there are many algebraic solutions to $g(x) = 0$ that are not of interest. Suppose that you only wanted solutions to $g(x) = 0$ at which all components of x were positive. Then one might solve the constrained problems

$$\begin{aligned} \min & \sum_{i=1}^n g^i(x)^2 \\ \text{s.t. } & x_i \geq 0, \quad i = 1, \dots, n. \end{aligned} \tag{5.6.2}$$

Even when doing this, (5.6.2) should be solved with loose stopping rules, producing an initial guess for a good nonlinear equation algorithm.

Enhancing Convergence with Powell's Hybrid Method

The methods we have discussed present us with a choice. Newton's method will converge rapidly if it converges but it may diverge. On the other hand, the minimization idea in (5.6.1) will converge to something, but it may do so slowly, and it may converge to a point other than a solution to the nonlinear system. Since these approaches have complementary strengths and weaknesses, we are tempted to develop a hybrid algorithm combining their ideas. If we define $SSR(x) \equiv \sum_{i=1}^n f^i(x)^2$, then the solutions to the nonlinear equation $f(x) = 0$ are exactly the global solutions to the minimization problem (5.6.1). Perhaps we can use values of $SSR(x)$ to indicate how well we are doing and help restrain Newton's method when it does not appear to be working.

Powell's hybrid scheme is one such algorithm. Powell's method modifies Newton's method by checking if a Newton step reduces the value of SSR . More specifically, suppose that the current guess is x^k . The Newton step is then $s^k = -J(x^k)^{-1}f(x^k)$. Newton's method takes $x^{k+1} = x^k + s^k$ without hesitation; Powell's method checks $x^k + s^k$ before accepting it as the next iteration. In particular, Powell's method will accept $x^k + s^k$ as the next iteration only if $SSR(x^k + s^k) < SSR(x^k)$, that is, only if there is some improvement in SSR .

Powell's original proposal was to set $x^{k+1} = x^k + \lambda s^k$ for some $0 < \lambda < 1$ such that $SSR(x^k + \lambda s^k) < SSR(x^k)$. As long as $J(x^k)$ is nonsingular, there is such a λ . The idea is that we move in the Newton direction but only to the extent consistent with moving downhill in the sense of SSR .

Unfortunately, experience showed that this method may converge inappropriately to some point where J is singular. Powell then modified this procedure by choosing a direction equal to a combination of the Newton step and the gradient of $-SSR$, which is the steepest descent direction; Powell (1970) provides the details.

These methods will converge to a solution of $f(x) = 0$ or will stop if they come too near a local minimum of SSR . If there is no such local minimum then we will get global convergence. If we do get stuck near a local minimum x^m , we will know that we are not at the global solution, since we will know that $f(x^m)$ is not zero, and we can continue by choosing a new starting point.

Solving Overidentified Systems

Sometimes we may want to solve a nonlinear system $f(x) = 0$ where $f : R^n \rightarrow R^m$ and $n < m$; such a problem is likely not to have a solution. In these cases we could proceed as we did in section 3.14 with overidentified linear systems. Specifically, we solve the least squares problem

$$\min_x f(x)^T f(x). \quad (5.6.3)$$

The result is typically a nonlinear least squares problem that can be solved using the optimization methods described in section 4.5. In particular, we may want to use a small residual method. While this may appear to be the same as using (5.6.1) to solve n equations in n variables, there is an important difference. Since there is likely no exact solution to an overidentified problem, the conditioning problems in (5.6.1) are not likely to occur. Therefore, for overidentified systems, a nonlinear least squares procedure is a sensible choice.

5.7 Advantageous Transformations

When solving nonlinear systems of equations, preliminary transformations of the equations may prove useful. The key assumption of Newton's method is that the system is well-approximated by a linear function. Changes in the equations can make that assumption more valid and help Newton's method to converge. We will examine the value of nonlinear transformations which turn a nonlinear system into an equivalent one easier to solve.

The general point can be illustrated in two simple examples. Suppose that we want to solve

$$x^9 - 1 = 0. \quad (5.7.1)$$

Equation (5.7.1) has a unique real solution, $x = 1$. If we apply Newton's method directly to (5.7.1), with an initial guess of $x = 2$, we get the sequence 2, 1.778, 1.582, 1.409, 1.259, 1.137, 1.050, 1.009, 1.000.

Table 5.4
Newton's method applied to (5.7.2) and (5.7.3)

Iteration	(5.7.2)	(5.7.3)	(5.7.2)	(5.7.3)
0	(2, 2)	(2, 2)	(3, 3)	(3, 3)
1	(0.6118, 0.7992)	(0.9420, 1.0580)	(−0.2791, 0.3613)	(0.8122, 1.188)
2	(0.9259, 0.9846)	(0.9976, 0.9998)	*	(0.9791, 0.9973)
3	(0.9978, 0.9998)	(1.0000, 1.0000)	*	(0.9999, 1.0000)
4	(1.0000, 1.0000)	(1.0000, 1.0000)	*	(1.0000, 1.0000)

Note: The * indicates complex values.

On the other hand, (5.7.1) is equivalent to $x^9 = 1$, which is equivalent to $x = 1$ and $x - 1 = 0$. This transforms the highly nonlinear equation (5.7.1) into the trivial linear equation, $x - 1 = 0$, which has the same real solutions and which Newton's method will solve exactly after one iteration.

This is a trivial application of the idea, but it clearly illustrates the point that transformations can improve convergence of Newton's method. A more substantive example is the system

$$\begin{aligned}x^{0.2} + y^{0.2} - 2 &= 0, \\x^{0.1} + y^{0.4} - 2 &= 0,\end{aligned}\tag{5.7.2}$$

which has a solution $(x, y) = (1, 1)$. If we apply Newton's method directly to (5.7.2) with an initial guess of $(2, 2)$, we get the (x, y) sequence in column 2 of table 5.4. If we instead apply Newton's method to the equivalent system

$$(x^{0.2} + y^{0.2})^5 - 32 = 0,\tag{5.7.3a}$$

$$(x^{0.1} + y^{0.4})^4 - 16 = 0,\tag{5.7.3b}$$

we get the sequence in column 3 of table 5.4. Equations (5.7.3) are useful because any solution of (5.7.2) is a solution of (5.7.3). Columns 4 and 5 report the results with a starting point of $(3, 3)$. We see that if we use (5.7.2), we need to use complex arithmetic; most software would not use complex arithmetic and quit at iteration 2 when it tries to compute a root of a negative number. Newton's method applied to (5.7.3) starting at $(3, 3)$ had no such problem. In general, we see that both sequences converge to $(1, 1)$ more rapidly and more reliably for (5.7.3) than for (5.7.2).

The intuition for these results is clear. The transformed equation (5.7.3a) replaced $x^{0.2} + y^{0.2}$ with a CRTS function; in particular, the resulting function is linear along any line of the form $y = mx$. In (5.7.3b), the exponent 4 counters the “average”

exponent 0.25 in the expression $x^{0.1} + y^{0.4}$. In both equations the exponents unwrap some of the nonlinearity of the expression to which it is applied.

These examples are only suggestive. The general idea is that a global nonlinear transformation may create an algebraically equivalent system on which Newton's method does better because the new system is more linear. Unfortunately, there is no general way to apply this idea; its application will be problem-specific. Fortunately simple and useful transformations are often suggested by the structure of the problem.

5.8 A Simple Continuation Method

No Newton-type method, nor any version of Gauss-Jacobi, Gauss-Seidel, or fixed-point iteration, is globally convergent; the best we are able to prove is that a method converges to a solution if we have a good initial guess or if the system of equations has some special structure. Also, for each regular zero, Newton's method has a non-trivial basin of attraction. These are good properties, but in practice, we often have no good guesses and only hope that a method will converge starting from our rough guesses. Sometimes we are not so lucky.

In this section we discuss a *continuation method* that uses the local convergence properties of a standard method to improve the chances of global convergence. The idea of all continuation methods is to construct a sequence of problems that ultimately leads to the problem of interest. Suppose that we want to solve $f(x; t) = 0$, where $f : R^n \times R \rightarrow R^n$, for some specific value $t = t^*$. This is natural structure in economic problems where x is a list of endogenous variables and t is a parameter of taste, technology, or policy. Further suppose that we do not have a solution for $f(x; t^*) = 0$ but we do know that for t^0 , $f(x; t^0) = 0$ has a solution x^0 . If t^* is near t^0 , x^0 will be a good initial guess when trying to solve $f(x; t^*) = 0$. Furthermore, in methods where Jacobians are sequentially approximated such as with Broyden, an appropriate initial guess for the Jacobian for the t^* problem is the last approximation for the Jacobian of the t^0 problem.

If t^* is not close to t^0 , we cannot rely on the local convergence property of the nonlinear equation method. We can still use this idea by constructing a sequence of problems of the form $f(x; t) = 0$ satisfying $t^0 \approx t^1 \approx t^2 \approx \dots \approx t^n \approx t^*$. We set up a sequence of problems to which we can apply the hot start idea, using the solution from the t^k problem as the initial guess to the t^{k+1} problem. The hope is that each intermediate problem can be solved quickly and that we end up with a solution to $f(x; t^*) = 0$.

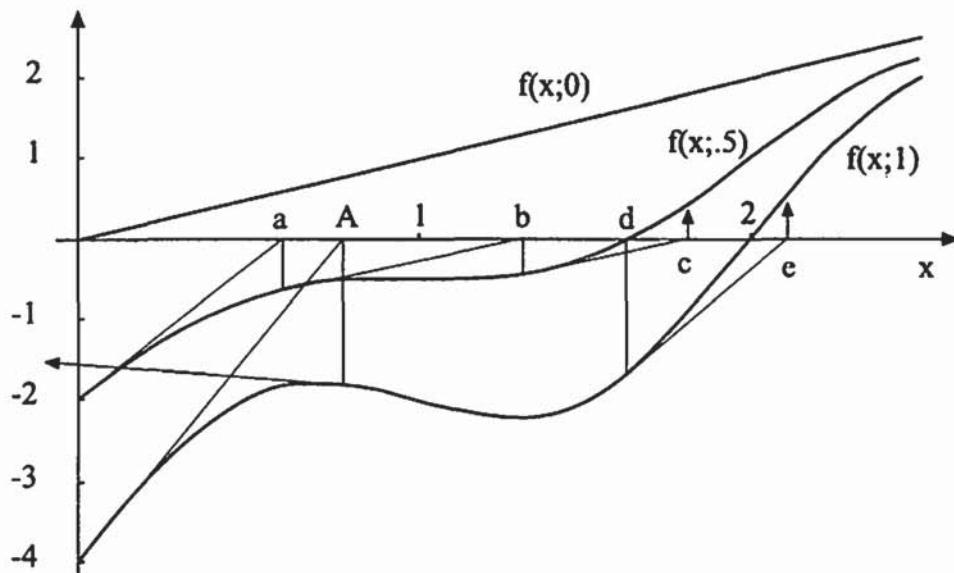


Figure 5.8
Continuation method for (5.8.1)

Algorithm 5.5 Simple Continuation Method

Objective: Find a zero of $f(x; t^*)$, $f : R^n \times R \rightarrow R^n$ where we know $f(x^0; t^0) = 0$.

Initialization. Form the sequence $t^0 \approx t^1 \approx t^2 \dots \approx t^n = t^*$; set $i = 0$.

Step 1. Solve $f(x; t^{i+1}) = 0$ using x^i as the initial guess; set x^{i+1} equal to the solution.

Step 2. If $i + 1 = n$, report x^n as the solution to $f(x; t^*)$ and STOP; else go to step 1.

Consider the following example: Let

$$f(x; t) = (1 - t)x + t(2x - 4 + \sin(\pi x)). \quad (5.8.1)$$

In figure 5.8 we display the three problems corresponding to $t = 0, 0.5, 1.0$. Because of the oscillations in $\sin(\pi x)$, Newton's method applied to $f(x; 1) = 0$ starting with $x = 0$ will have problems. In fact the first iterate is $x = A$ in figure 5.8, but because the slope of $f(x; 1)$ is slightly negative at $(A, f(A; 1))$, the second iterate is negative, as illustrated in figure 5.8. On the other hand, the solution to the $t = 0$ problem, $x = 0$, is trivial. We proceed in two steps. We use $x = 0$, the solution to the $t = 0$ problem, as the initial guess for the $t = 0.5$ problem. Figure 5.8 illustrates the first three Newton iterates, points a, b , and c , after which Newton's method converges to the unique zero, $x = d$, for the $t = 0.5$ problem. We next take $x = d$ as the initial guess for the $t = 1$ problem and find that Newton's method leads us first to $x = e$ and

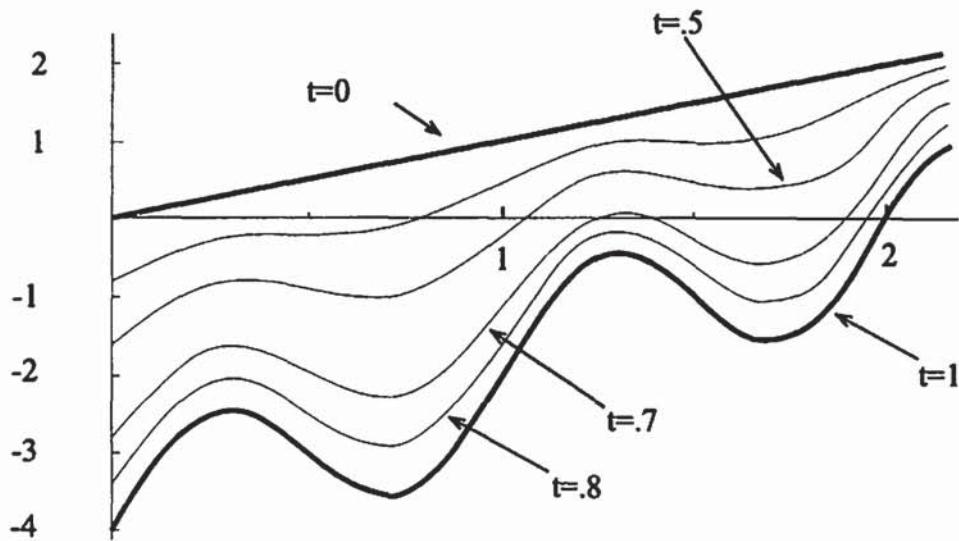


Figure 5.9
Graphs of (5.8.2)

then to the solution $x = 2$. Therefore Newton's method and simple continuation works well solving a problem starting at $x = 0$, whereas Newton's method alone starting at $x = 0$ would fail.

While this procedure is intuitive and will often work, it may fail. To see this, consider the slightly different example

$$H(x; t) = (1 - t)x + t(2x - 4 + \sin(2\pi x)). \quad (5.8.2)$$

The only difference between f and H is that the sine argument is $2\pi x$ in H instead of πx . Figure 5.9 displays $H(x; t)$ for $t = \{0, 0.5, 0.7, 0.8, 1.0\}$. Again, if we try to solve for $H(x; 1) = 0$ with Newton's method with an initial guess $x = 0$, we will not do well. We will be tempted to try a simple continuation procedure. At $t = 0$ we start with the equation $x = 0$ and find the unique zero at $x = 0$. Suppose that we use the continuation approach and increase t in increments of 0.01. Note that $H(x; 0.70)$ has three zeros on $[0, 2]$. We will eventually come to the problem $H(x; 0.74) = 0$ and find the solution $x = 1.29$. We now encounter a difficulty. A solution to the problem $H(x; 0.74) = 0$ is $x = 1.29$, but the problem $H(x; 0.75) = 0$ has a unique solution of $x = 1.92$ and no solution near $x = 1.29$. If we use $x = 1.29$ as the initial guess for Newton's method for $H(x; 0.75) = 0$, the first five iterates are 1.29, 1.31, 1.23, 1.27, and 1.30, which do not appear to be converging to anything.

In figure 5.9 we see what went wrong. Close examination of $H(x; t)$ shows that $x = 1.28$ is close to a zero (it is actually close to two zeroes) of $H(x; 0.70)$ but is not close to any zero of $H(x; 0.75) = 0$. Therefore a small change in the problem causes a

large change in the solution. This violates the basic premise of the simple continuation method. In the next section we will see how to improve on the continuation idea and construct a reliable algorithm.

5.9 Homotopy Continuation Methods

The idea behind continuation methods is to examine and solve a series of problems, beginning with a problem for which we know the solution and ending with the problem of interest. In reality we are deforming the problem as we progress. Homotopy methods formalize the notion of deforming a simple problem into a hard problem, computing a series of zeros of the intervening problems in order to end with a zero to the problem of interest. With homotopy methods, we finally have a globally convergent way to find zeros of $f : R^n \rightarrow R^n$.

We start by constructing homotopy functions, $H(x, t)$, $H : R^{n+1} \rightarrow R^n$, $H \in C^0(R^{n+1})$. A *homotopy function* H that continuously deforms g into f is any continuous function H where

$$H(x, 0) = g(x), \quad H(x, 1) = f(x). \quad (5.9.1)$$

In practice, we take $H(x, 0)$ to be a simple function with easily calculated zeros, and $H(x, 1)$ to be the function whose zeros we want. Let us consider some simple examples. The *Newton homotopy* is $H(x, t) = f(x) - (1-t)f(x^0)$ for some x^0 . At $t = 0$, $H = f(x) - f(x^0)$ which has a zero at $x = x^0$. At $t = 1$, $H = f(x)$. This is a simple homotopy, since the difference between $H(x, t)$ and $H(x, s)$ is proportional to $t - s$. The *fixed-point homotopy* is $H(x, t) = (1-t)(x - x^0) + tf(x)$ for some x^0 . It transforms the function $x - x^0$ into $f(x)$. A more general homotopy is the *linear homotopy*, $H(x, t) = tf(x) + (1-t)g(x)$, which transforms g into f , since at $t = 0$, $H = g$, and at $t = 1$, $H = f$.

The basic object of our attention is the set

$$H^{-1}(0) = \{(x, t) \mid H(x, t) = 0\}.$$

If $H(x, 0)$ and $H(x, 1)$ have zeros, the hope is that there is a continuous path in $H^{-1}(0)$ which connects zeros of $H(x, 0)$ (the simple function) to zeros of $f(x) = H(x, 1)$. An example of such a zero set is displayed in figure 5.10. We return to the example in figure 5.9 for which simple continuation failed. We again take $f(x) = 2x - 4 + \sin(2\pi x)$ and construct the fixed-point homotopy

$$H(x, t) = (1-t)x + t(2x - 4 + \sin(2\pi x)) \quad (5.9.2)$$

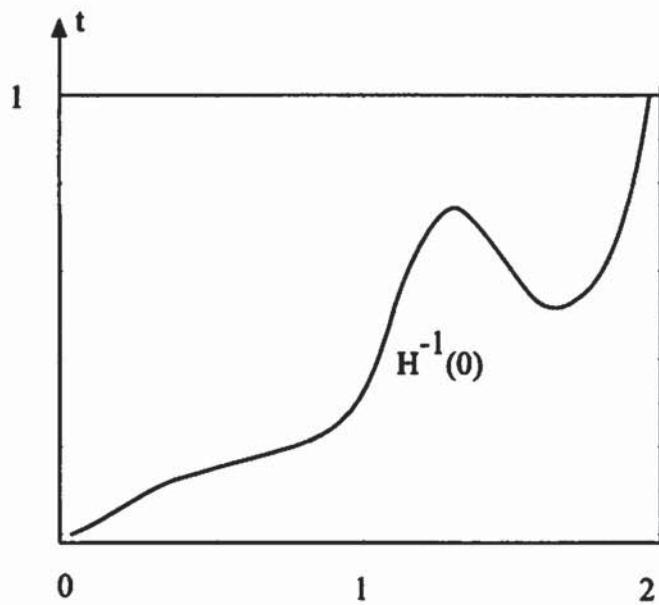


Figure 5.10
 $H^{-1}(0)$ set for (5.9.2)

At $t = 0$ the unique zero is $x = 0$. Figure 5.9 displayed $H(x, t)$ for $t \in \{0, 0.5, 0.7, 0.8, 1.0\}$. Note that $H(x, 0.7)$ has three zeros and that $H(x, 1)$ has a unique zero at $x = 2$.

Figure 5.10 displays $H^{-1}(0)$ for (5.9.2) in (x, t) space. At each t we plot all x such that $H(x, t) = 0$. At $t = 0$ this is $x = 0$ only. For $t \in (0.53, 0.74)$ there are three zeros of $H(x, t)$. At $t = 1$ the unique zero is $x = 2$.

The key fact is that $H^{-1}(0)$ for (5.9.2) is a smooth curve connecting the zero of the $t = 0$ problem to the zero of the $t = 1$ problem. Homotopy methods focus on following paths in $H^{-1}(0)$ connecting the $t = 0$ problem to the $t = 1$ problem. These paths may not be simple. For example, in figure 5.10 the values of t are not monotonically increasing as we move along $H^{-1}(0)$ from $(0, 0)$ to $(2, 1)$. Simple continuation assumes that we can proceed from the zero of the $t = 0$ problem to the zero of the $t = 1$ problem by taking an increasing sequence of t values, a strategy that fails near $(x, t) = (1.31, 0.745)$ in figure 5.10. Homotopy methods instead follow the path $H^{-1}(0)$, tracing it wherever it goes and allowing t to increase and decrease as necessary to stay on $H^{-1}(0)$.

Before we indicate how this tracing is accomplished, we will discuss the forms $H^{-1}(0)$ may take for general homotopies $H : R^{n+1} \rightarrow R^n$. The path $H^{-1}(0)$ in figure 5.10 is not too bad. Figure 5.11 illustrates several possible features of $H^{-1}(0)$. $H^{-1}(0)$ generally consists of several pieces. Path AB is a simple, clean component of $H^{-1}(0)$.

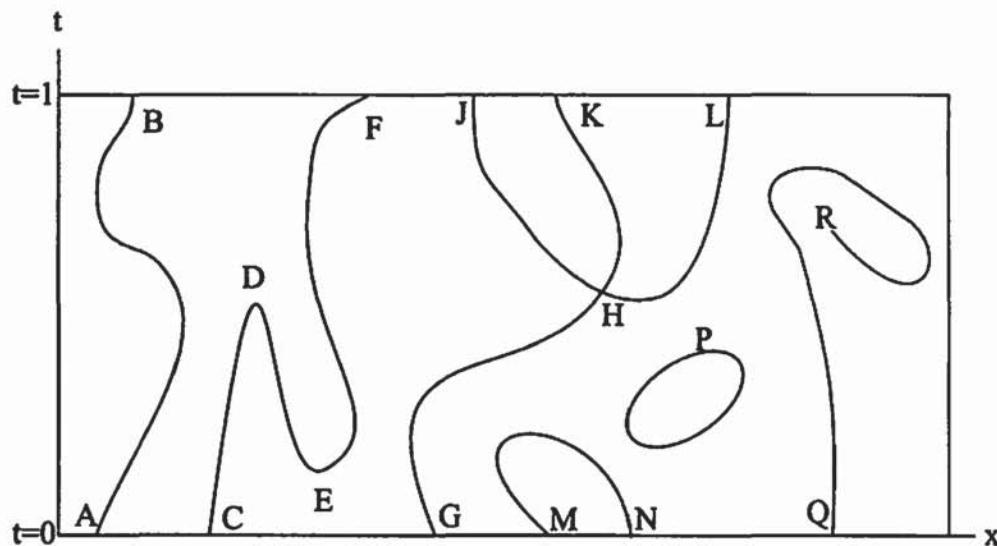


Figure 5.11
Possible $H^{-1}(0)$ sets

leading from $R^n \times \{0\}$, the bottom axis, to $R^n \times \{1\}$, the top axis. The key simplifying property of AB is that t is increasing along AB . The simple continuation method of the previous section will successfully traverse paths like AB if we take sufficiently small increments in t . The example in figure 5.8 has a $H^{-1}(0)$ set like AB .

Component $CDEF$ is quite different. Component $CDEF$ contains *turning points* at D and E , similar to the problematic path in figure 5.10. We can see here how the simple continuation method we described in the last section may have difficulty when it reaches a turning point. At a turning point there are no nearby solutions x to $H(x, t) = 0$ as we try to increase t . The component beginning at G presents even worse problems at the *branch point* H where three branches sprout, one for each of the solutions at J , K , and L . At branch points there are “too many” nearby solutions. At both turning and branch points, the Jacobian $H_x(x, t)$ is singular. Using Newton’s method, or any other method, at such a point would send the iterates off in essentially random directions, and end any advantage of the continuation process.

Figure 5.11 also displays other kinds of problematic components. MN is a component that begins and ends at the $t = 0$ boundary. P is a closed curve touching neither the $t = 0$ nor $t = 1$ boundaries. QR begins with a solution to $H(x, 0) = 0$ at Q but ends at R before reaching the $t = 1$ boundary.

Despite the two-dimensional picture figure 5.11 represents the general case. Think of the bottom axis representing a compact subset of R^n . The paths pictured in figure 5.11 are one-dimensional threads that run through $x - t$ space where $x \in R^n$. The

paths in the general case can be very complex, but we will see that we may restrict our attention to “nice” ones.

Parametric Path Following

Let us now consider how we trace out the path $H^{-1}(0)$. To follow paths like AB and $CDEF$, we view them as *parametric paths*. A parametric path is a set of functions $(x(s), t(s))$ that describes a path through (x, t) space. In particular, we want to construct parametric paths to traverse paths like AB and $CDEF$. Instead of parameterizing x in terms of t , the simple continuation method approach, we parameterize both x and t in terms of a third parameter, s .

The key advantage of parametric paths is that x and t are now viewed symmetrically, with both x and t being functions of some parameter s . The parametric path should satisfy $H(x(s), t(s)) = 0$ for all s . If we are on the path, the change in x and t necessary to maintain the condition $H(x(s), t(s)) = 0$ can be determined by implicit differentiation. Implicit differentiation of $H(x(s), t(s)) = 0$ with respect to s yields n conditions on the $n + 1$ unknowns,

$$\sum_{i=1}^n H_{x_i}(x(s), t(s))x'_i(s) + H_t(x(s), t(s))t'(s) = 0. \quad (5.9.3)$$

Because of the one dimension of indeterminacy, there are many solutions to the parametric equations in (5.9.3). However, we only care about $(x(s), t(s))$ pairs, not their dependence on s . Hence we need to find only one solution. If we define $y(s) = (x(s), t(s))$, then y obeys the system of differential equations

$$\frac{dy_i}{ds} = (-1)^i \det\left(\frac{\partial H}{\partial y}(y)_{-i}\right), \quad i = 1, \dots, n+1, \quad (5.9.4)$$

where $(\cdot)_{-i}$ means we remove the i th column.

Garcia and Zangwill (1981) call (5.9.4) the *basic differential equation*. When we reformulate the problem in this differential equation form, we immediately see a problem. If the $((\partial H / \partial y)(y)_{-i})$ matrices in (5.9.4) are ever all singular before $t(s) = 1$, then the system (5.9.4) will come to a stop. To avoid this, the $n \times (n + 1)$ matrix $H_y = (H_x, H_t)$ must full rank, that is, a rank of n , for all (x, t) on the path. Since the rank of $H_y(y)$ is so important, we have a special name for homotopies that have full rank:

DEFINITION The homotopy H is *regular* iff $H_y(y)$ is of full rank at all points in $H^{-1}(0)$, where $y \equiv (x, t)$.

Therefore, if a homotopy has full rank, the differential equation system (5.9.4) will have a solution that parameterizes a path leading to a solution of $H(x, 1) = 0$. The full rank condition is weaker than it may initially appear to be. H_y has full rank along simple path like AB but also at turning points like D and E in figure 5.11. This is substantial progress over simple continuation which requires H_x be nonsingular. Unfortunately, the homotopy is not regular at branch points like H in figure 5.11.

When we apply this method to our example (5.9.2), the system (5.9.4) becomes

$$\begin{pmatrix} dx/ds \\ dt/ds \end{pmatrix} = \begin{pmatrix} -H_t \\ H_x \end{pmatrix} = \begin{pmatrix} x - (2x - 4 + \sin(2\pi x)) \\ 1 - t + t(2 + 2\pi \cos(2\pi x)) \end{pmatrix}. \quad (5.9.5)$$

To find a zero of $H(x, 1)$ in (5.9.2), we start with $(x, t) = (0, 0)$ and then solve the differential equation (5.9.5) until we reach a s such that $t(s) = 1$. At that s , $x(s)$ will be a zero of $H(x(s), t(s)) = 0$.

We typically solve (5.9.4) numerically. We will discuss numerical methods for solving differential equations in chapter 10, but we will use a simple scheme to solve our example, (5.9.5). We let $s_i = i/1000$ for $i = 0, 1, \dots, 720$, and define

$$\begin{aligned} x_{i+1} &= x_i + h(x_i - (2x_i - 4 + \sin 2\pi x_i)), \\ t_{i+1} &= t_i + h(1 - t_i + t_i(2 + 2\pi \cos 2\pi x_i)), \end{aligned} \quad (5.9.6)$$

where $h = 0.001$ is the step size corresponding to ds and $x_0 = t_0 = 0$ are initial conditions. The iteration (5.9.6) approximates the solution to (5.9.5) at the s_i points. Table 5.5 displays several of these points as computed by (5.9.6).

The (t_i, x_i) pairs in table 5.5 are so close to the $H^{-1}(0)$ set graphed in figure 5.10 that a graph cannot distinguish them. The value $t = 1$ is achieved between $i = 711$ and $i = 712$; the natural estimate of the zero of $2x - 4 + \sin 2\pi x$ is 2.003, since $t = 1$ is halfway between t_{711} and t_{712} . This answer is correct to three significant digits. Furthermore, once we have the estimate of 2.003 for the root, we can use it as an initial guess for Newton's method, which will quickly converge to the true solution $x = 2.0$ in this case.

Good Homotopy Choices

Figure 5.11 illustrates several possible paths, some do connect the $t = 0$ line to the $t = 1$ line, but many others don't. We will choose homotopies that avoid the bad possibilities. Some problems are easily avoided. For example, MN in figure 5.11 turns back to the $t = 0$ line, something that is possible only if $H(x, 0)$ has multiple zeros. To avoid this, we just choose a homotopy where $H(x, 0)$ has a unique zero.

Table 5.5
Homotopy path following in (5.9.6)

Iterate i	t_i	x_i	True solution $H(x, t_i) = 0$
50	0.05621	0.16759	0.16678
100	0.11130	0.30748	0.30676
200	0.15498	0.64027	0.64025
300	0.35543	1.02062	1.01850
400	0.69540	1.23677	1.23288
500	0.67465	1.41982	1.42215
600	0.51866	1.68870	1.69290
650	0.61731	1.84792	1.84391
700	0.90216	1.97705	1.97407
711	0.99647	2.00203	1.99915
712	1.00473	2.00402	2.00114

Closed curves are no problem as long as there are components that join the $t = 0$ and $t = 1$ cases.

The more serious problems arise from branch points. They can be avoided if we do a good job constructing our homotopy. A basic theorem is the next one:

THEOREM 5.9.1 Suppose that $f \in C^2$. Let D be compact with nonempty interior. Define

$$H(x, t) = (1 - t)(x - x^0) + tf(x).$$

If H is regular, and $H(x, t) \neq 0$ for all $0 \leq t < 1$ and $x \in \partial D$, then f has a zero that is at the end of a path joining $t = 0$ and $t = 1$ in $H^{-1}(0)$.

A key assumption in this method is that H is regular. In figure 5.11 the presence of the branch point at H or a dead end at QR means that the homotopy is not regular. Determining that a homotopy is regular is a computational task as demanding as finding zeros. It would appear that we have not made any progress. Fortunately the following theorems tell us that many simple homotopies are regular:

THEOREM 5.9.2 (Chow et al. 1978) If $B^n \equiv \{x \in R^n \mid |x| < 1\}$ and $f : \overline{B^n} \rightarrow \overline{B^n}$ is C^1 , then the homotopy

$$H : B^n \times (0, 1) \times B^n \rightarrow R^n,$$

$$H(a, t, x) = (1 - t)(x - a) + t(x - f(x)),$$

is regular and for almost all $a \in B^n$, $H^{-1}(0)$ is a smooth curve joining $(0, a)$ to a fixed point of f at $t = 1$.

THEOREM 5.9.3 (Mas-Colell) There is an open and dense subset of C^2 functions, \mathcal{F} , such that for $f \in \mathcal{F}$, $H^{-1}(0)$ is a smooth curve for the homotopy $H(x, t) = tf(x) - x$.

We should comment that $H^{-1}(0)$ still could have infinite length and oscillate as $t \rightarrow 1$. However, that is of only moderate practical concern. Indeed, by continuity of f , we will reach a point where $|f(x) - x| < \varepsilon$ in finite time for any $\varepsilon > 0$.

When we put together the ideas above, we arrive at one simple homotopy solution method to find a fixed point for $f(x) = x$. Remember that theorem 5.9.2 says that it will work with probability one; if the algorithm appears to be failing, then start over at the initialization step of algorithm 5.6 with another a .

Algorithm 5.6 Generically Convergent Euler Homotopy Method

Initialization. Choose an $a \in R^n$ and form the homotopy. $H(a, t, x) = (1 - t)(x - a) + t(x - f(x))$. Let $x_0 = a$, $s_0 = 0$, and $t_0 = 0$. Choose step size $\varepsilon > 0$.

Step 1. Given x_i , s_i , and t_i , compute x_{i+1} and t_{i+1} using (5.9.4). Let $s_{i+1} = s_i + \varepsilon$.

Step 2. If $t_{i+1} > 1$ go to step 3; else go to step 1.

Step 3. Stop and return the last iterate of x as the solution.

Theorems 5.9.2 and 5.9.3 say that we may still have to handle the curves like *CDEF* in Figure 5.11 where the index t is not monotonic but that for many simple homotopies we won't have to worry about paths with branch points. These theorems essentially say that the homotopy methods are almost surely convergent. This high degree of reliability is good, but it comes at a high cost, since tracking one of these curves may involve a great deal of calculation.

While theorems 5.9.2 and 5.9.3 present cases of generic regularity, we could first try more natural homotopies. An alternative approach is to let $H(x, t) = (1 - t)g(x) + tf(x)$ where $g(x)$ is similar to $f(x)$. For example, if $f(x)$ is a system of excess demand equations, we could let $g(x)$ be a system of excess demand equations with a known solution. Constructing such a $g(x)$ is often easy; for example, we could let $g(x)$ be the case where each agent has the same endowment and tastes with the common endowment and tastes being the “average” endowment and tastes behind the excess demand system in $f(x)$. There is also no requirement that $H(x, t)$ be linear in t . While these approaches may lead to problems with branch points, they may also be much faster. As always in numerical analysis, it sometimes pays to try approaches that do not fall into the safe categories; novel ones may be turn out to be very effective and one can always go back to the safe methods.

Simplicial Homotopy Methods

Homotopy methods using differential equations still need to compute or approximate Jacobians. That will become increasingly impractical as the problem size increases.

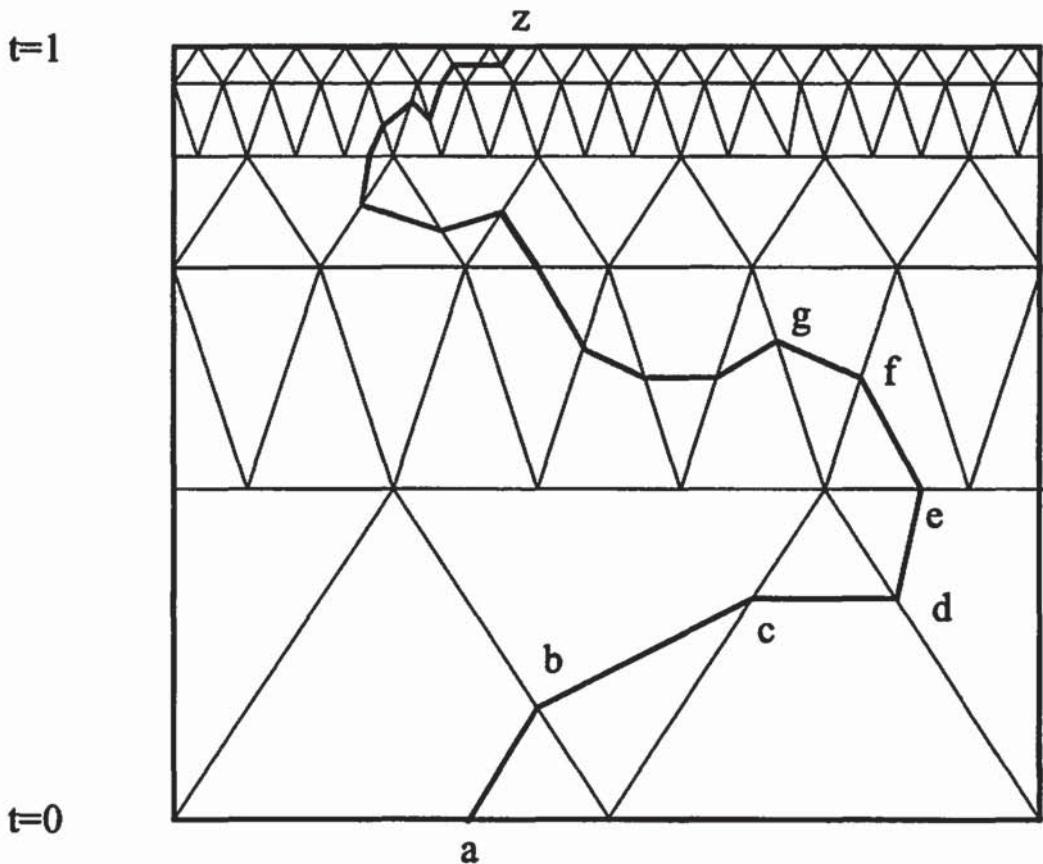


Figure 5.12
Piecewise-linear homotopy

Simplicial homotopy methods continue to use the continuation ideas but depend on a “rougher” deformation of $H(x, 0)$ into f . The resulting method uses only function evaluations of f .

We will first describe a simple version of simplicial methods. We begin with a homotopy function $H(x, t)$. Next, divide (x, t) space into simplices, such as the triangular decomposition in figure 5.12. The next, and defining, step is to define a new function $\hat{H}(x, t)$, agreeing with H at the simplicial nodes but linear within each simplex. We then construct the path of zeros of \hat{H} . Figure 5.12 shows such a simplicial division and path of zeros $abcdefg\dots$ that leads to a solution at z .

In a simplicial method we let the piecewise linear function $\hat{H}(x, t)$ deform $\hat{H}(x, 0)$ into $\hat{H}(x, 1)$ which is a piecewise linear approximation of f . The advantage of this method is that the computation of $\hat{H}^{-1}(0)$ is relatively easy, since \hat{H} is locally linear and each of the $\hat{H}^{-1}(0)$ is piecewise linear. The cost is that the final zero is a zero of $\hat{H}(x, 1)$, not f . A potential problem is that $\hat{H}^{-1}(0)$ may lead to points that are not near zeros of f , or that the path may lead nowhere. The following theorem says

that the method will work if the simplicial decomposition of (x, t) space is sufficiently fine:

THEOREM 5.9.4 For all ε there is an δ such that if the simplices in \hat{H} are less than δ on each edge, then $\hat{H}^{-1}(0)$ is within ε of some path in H^{-1} .

Combining Homotopy Methods and Newton's Method

Homotopy methods have good convergence properties, but they may be slower than alternatives. In many homotopy methods each step involves computing a Jacobian; in such cases each step is as costly as a Newton step. This observation suggests that we first use a homotopy method to compute a rough solution and then use it as the initial guess for Newton's method.

The slow convergence of homotopy methods also implies that it is difficult to satisfy the tough convergence criterion. In such cases applying Newton's method to the final iterate of a homotopy method is a natural step to improve on the homotopy result. In either case we can implement this by switching to Newton's method at some $t < 1$.

5.10 A Simple CGE Problem

One of the classical uses of nonlinear equations in economics is the computation of competitive equilibrium. In this section we will consider the computation of equilibrium in a simple endowment economy.

Assume that there are m goods and n agents. Assume that agent i 's utility function over the m goods is $u^i(x)$, $x \in R^m$. Suppose that agent i 's endowment of good j is e_j^i . Economic theory proceeds in the following fashion: First, we "construct" the demand function for each agent; that is, we define

$$d^i(p) = \arg \max_x u^i(x) \tag{5.10.1}$$

$$\text{s.t. } p \cdot (x - e^i) = 0,$$

where $p \in R^m$ is the price vector. We then construct the excess demand function, $E : R^m \rightarrow R^m$, defined by $E(p) = \sum_i^n (d^i(p) - e^i)$. Equilibrium is any $p \in R^m$ such that $E(p) = 0$. By the degree zero homogeneity of demand in prices, if $p \in R^m$ is an equilibrium price, so is λp for any $0 \neq \lambda \in R$; in particular, there is an equilibrium on the unit simplex.

To prove existence of an equilibrium price vector, we construct a map $g(p)$ on the unit simplex $S^{m-1} = \{p | \sum_{j=1}^m p_j = 1\}$ such that at any fixed point of $g(p)$, say p^* , we

have $E(p^*) \leq 0$. One such function (see Varian 1978) is

$$g_j(p) = \frac{p_j + \max(0, E_j(p))}{1 + \sum_{j=1}^m \max(0, E_j(p))}. \quad (5.10.2)$$

Since $g : S^{m-1} \rightarrow S^{m-1}$ is continuous, it has a fixed point.

We have several alternative computational approaches to compute a equilibrium price vector. We will now discuss the various methods below.

Fixed-Point Iteration

We could use fixed-point iteration to compute the equilibrium price. Since all equilibria are solutions to the fixed-point problem $p = g(p)$, we could execute the iteration $p^{k+1} = g(p^k)$. If we have closed-form expressions for the individual demand functions, then this approach is particularly easy to implement and avoids the complications of computing Jacobians. Since the map g has range in the unit simplex, there is no chance that some p_k iterate falls outside the permissible range of prices. Of course there is no assurance that the fixed-point iteration approach will converge.

$E(p) = 0$ as a Nonlinear System

One approach is to solve the system $E(p) = 0$ for p . If we can analytically solve for the individual demand functions, then we can construct a routine that explicitly computes the excess demand function and feeds it to a nonlinear equation solver. However, we must take into account the degree zero homogeneity of E . Strictly speaking, if there is one solution, then there is a continuum of solutions. One way to deal with this is to add an equation forcing the prices to lie on the unit simplex, $\sum_{i=1}^m p_i = 1$. This gives us one more equation than unknown, but Walras's law tells us that if p satisfies $E(p) = 0$ for $m - 1$ components of E , then the m th component of E is also zero. Hence the system of equations we send to a nonlinear equation solver is

$$E_1(p) = 0,$$

\vdots

$$E_{m-1}(p) = 0,$$

$$\sum_{i=1}^m p_i = 1.$$

In this form we can use Newton's method or a homotopy method to solve for equilibrium.

Hierarchical Numerical Methods

Our discussion has implicitly assumed that we can express $E(p)$ in closed form. If we don't have closed-form expressions for the individual demand functions, then we will have to solve for the individual demands numerically. For each price p we will have to solve a constrained optimization problems for each agent. This produces a hierarchical structure in the resulting algorithm. Suppose that we use the fixed-point iteration approach to find a fixed point of $g(p)$ defined in (5.10.2). At each price iterate p^k , we will need to compute $g(p^k)$. To compute this excess demand vector, the algorithm will have to call upon an optimization routine to solve the demand problem, (5.10.2), for each agent with prices p^k and then use these solutions to compute aggregate excess demand and $g(p^k)$. More generally, we must construct a subroutine $E(p)$ to compute excess demand. In the end the nonlinear equation solver calls a routine for $E(p)$ that in turn calls an optimization routine.

This hierarchical structure generates some new problems. Convergence of any nonlinear equation solution method applied to $E(p)$ depends on the accuracy with which we solve the individual demand problems, which in turn is determined by the convergence factors we choose in the optimization algorithm chosen to solve (5.10.1). The convergence criterion of the nonlinear equation solver and the optimization routine must be consistent. We generally will demand a tight convergence criterion for the optimization routine because it uses an analytical representation for $u(x)$. That optimization problem will have some error in excess of machine precision. Since the convergence criterion for the nonlinear equation solver depends on the error with which $E(p)$ is computed, we need to impose a looser criterion for the nonlinear equation solver than we do for the optimization routine called by $E(p)$. In general, the outer routine of any hierarchical numerical approach must have looser convergence criterion so that it can tolerate the numerical errors generated by the inner numerical procedure.

First-Order Conditions and Market Balance

An alternative approach simultaneously solves for the individual demands and the equilibrium in one large nonlinear system. To do this, we specify a long list of equations, combining the first-order conditions of the individual optimization problem, their budget constraints, and the equilibrium conditions. In an m -good, n -agent example this results in combining the first-order conditions for each good and each agent

$$u_j^i(x^i) = p_j \lambda^i, \quad i = 1, \dots, n, j = 1, \dots, m, \quad (5.10.3)$$

the budget constraint for each agent

$$p \cdot (x^i - e^i) = 0, \quad i = 1, \dots, n, \quad (5.10.4)$$

the supply equals demand conditions for goods³ 1 through $m - 1$,

$$\sum_{i=1}^n (x_j^i - e_j^i) = 0, \quad j = 1, \dots, m - 1, \quad (5.10.5)$$

and the simplex condition for prices

$$\sum_j p_j = 1 \quad (5.10.6)$$

to yield a system of nonlinear equations in the unknowns p , x^i , and λ^i , $i = 1, \dots, n$. The system (5.10.3–5.10.6) can be sent to a nonlinear equation solver to determine the equilibrium value of λ and x as well as p .

This joint approach appears to be inefficient because of the large number of unknowns and equations. However, the Jacobian of the resulting joint system will be sparse. This approach is also easy to program, an advantage that may be decisive in small problems.

Negishi Method

A final procedure exploits the first theorem of welfare economics. This theorem states that any competitive equilibrium in an Arrow-Debreu model of general equilibrium is Pareto efficient; that is, for any equilibrium there is a set of nonnegative social welfare weights, λ^i , $i = 1, \dots, n$, such that the equilibrium allocation of final consumption, x^i , $i = 1, \dots, n$, is the solution to the social welfare problem

$$\begin{aligned} & \max_{x^1, x^2, \dots} \sum_{i=1}^n \lambda^i u^i(x^i) \\ \text{s.t. } & \sum_{i=1}^n (e^i - x^i) = 0. \end{aligned} \quad (5.10.7)$$

In the Negishi approach to computing general equilibrium, we look for the set of social welfare weights, λ^i , $i = 1, \dots, n$, such that the solution to (5.10.7) is an equi-

3. Recall that if supply equals demand for all but one market, and all budget constraints are satisfied, then supply equals demand for the last market.

librium allocation. Since prices are proportional to marginal utilities, the equilibrium condition can be written $\sum_j u_j^i(x^i)(x_j^i - e_j^i) = 0$ for $i = 1, \dots, n$.

We proceed as follows: For any vector of social welfare weights, λ , we compute the allocation, $X(\lambda) \in R^{m \times n}$, which solves (5.10.7). We next compute the prices, constrained to lie in the unit simplex, implied by the allocation $X(\lambda)$. These prices are defined by⁴

$$p_j = \frac{u_{x_j^1}^1(X^1(\lambda))}{\sum_{l=1}^m u_{x_l^1}^1(X^1(\lambda))} \equiv P_j(\lambda).$$

We then ask, for each agent i if he can afford his allocation, $X^i(\lambda)$, at the prices $P(\lambda)$ by computing the excess wealth function $W_i(\lambda) \equiv P(\lambda) \cdot (e^i - X^i(\lambda))$. If $W_i(\lambda)$ is nonnegative, then agent i can afford $X^i(\lambda)$ at prices $P(\lambda)$ and have $W_i(\lambda)$ in “wealth” left over. If $W_i(\lambda) = 0$ for each i , the prices $P(\lambda)$ are equilibrium prices, and $X(\lambda)$ is the equilibrium final allocation. Therefore the Negishi approach boils down to solving the system of nonlinear equations

$$W_i(\lambda) = 0, \quad i = 1, \dots, n, \tag{5.10.8}$$

for λ , and we then compute $P(\lambda)$ at the solution to (5.10.8) to get the equilibrium prices.

The Negishi approach has substantial advantages if there are fewer agents than goods and individual demands are analytically computable, since the number of unknowns equal the number of agents. It may also be useful even if we must numerically compute demands. When we have a single representative agent, then this approach is particularly appropriate because the competitive equilibrium just maximizes the representative agent’s utility. This fact is heavily exploited in many aggregate models.

5.11 Software

There is some software available for nonlinear equations, but there is not nearly the variety and quantity as that available for optimization. HYBRD, a program available as part of MINPACK, combines Powell’s method for enhancing convergence with Broyden’s procedure for updating the Jacobian. The result is a generally reliable

4. We need only compute the relative prices for one agent since any solution to (5.11.7) will equate marginal rates of substitutions across agents.

algorithm for solving nonlinear equations. HOMPACK, available from Netlib, uses differential equation homotopy approaches. Aluffi-Pentini, F., V. Parisi, and F. Zirilli (1984) presents a differential equation approach to solving nonlinear equations. Allgower and Georg (1990) is a recent presentation of homotopy methods, and it offers readers software to implement both piecewise-continuous and piecewise-linear algorithms. The web site [http://www.mcs.anl.gov/home/otc/Guide/SoftwareGuide/ Categories/nonlinequ.html](http://www.mcs.anl.gov/home/otc/Guide/SoftwareGuide/Categories/nonlinequ.html) contains links to many nonlinear equation packages. Kalaba and Tesfatsion (1991) extend the homotopy method to include a complex tracking variable in order to maneuver around and over branch points. More and Wright (1993) provide references to some nonlinear equation packages.

5.12 Further Reading and Summary

This chapter has presented the basic methods for solving nonlinear equations. One-dimensional problems are easily solved, reliably by comparison methods and often very quickly by Newton's method. Nonlinear systems of nonlinear equations are more difficult. Solving systems of nonlinear equations reduces to an iterative search guided generally by the Jacobian or diagonal portions of the Jacobian. For small systems we generally use Newton's method because of its good local convergence properties. Large systems are generally solved by breaking the system into smaller systems as in Gauss-Jacobi and Gauss-Seidel methods and their block versions.

Newton and Gaussian methods need good initial guesses, but finding good initial guesses is often an art and usually conducted in an ad hoc fashion. Powell's hybrid method combines Newton's method with optimization ideas to create a more reliable algorithm, but it too may not converge. Homotopy methods generically converge from any initial guess, and they are necessary to use when we can't easily generate good initial conditions. Unfortunately, homotopy methods are slower, and their use is limited to systems of small to moderate size. Of course what we consider "small" and "moderate" grows as computer power increases.

Nonlinear equation solving presents problems not present with linear equations or optimization. In particular, the existence problem is much more difficult for nonlinear systems. Unless one has an existence proof in hand, a programmer must keep in mind that the absence of a solution may explain a program's failure to converge. Even if there exists a solution, all methods will do poorly if the problem is poorly conditioned near a solution. Transforming the problem will often improve performance. After computing a solution, the condition number of the Jacobian should be computed if possible to get an estimate of the error.

The literature on nonlinear equations and computation of economic equilibrium is substantial. Ortega and Rheinboldt (1970) is a classic introduction to nonlinear equations. There is a large literature on computational general equilibrium; Johansen (1960) and Scarf and Hansen (1973) are classics in the area, and Shoven and Whalley (1992) and Dixon and Parmenter (1996) are comprehensive surveys of the literature. Scarf's method is a historically important method. It was the first globally convergent algorithm for solving nonlinear equations, but it has seldom been a competitive method in practice.

The Kakutani fixed-point theorem generalizes the Brouwer fixed-point theorem to correspondences, which allows for a more general theorem; Eaves (1971) discusses methods to solve such problems. Brown, DeMarzo, and Eaves (1994) and Schmedders (1996) have developed algorithms for computing equilibrium in general equilibrium models with incomplete asset markets. Actual homotopy solution methods are more complicated and efficient than (5.9.4). See Eaves (1972), Algower and Georg (1990), and Garcia and Zangwill (1981) for discussions of homotopy methods.

The game theory literature contains much related material on computing Nash equilibrium. The numerical literature begins with Lemke and Howson's (1964) treatment of two-person games, and Wilson (1971) generalized this to n players. Wilson (1992) presents an algorithm for finding stable components of Nash equilibria. McElvey (1996) surveys this literature. Wilson (1996) presents methods for solving nonlinear pricing problems.

Variational inequalities are problems that encompass both optimization and nonlinear equations. Nagurney (1993) contains a clear introduction to this topic.

Exercises

1. Solve $\sin 2\pi x - 2x = 0$ using bisection, Newton's method, the secant method, and fixed-point iteration. For what values of the initial guess $x_0 \in [-2, 2]$ does each of these methods converge? Repeat for $\sin 2\pi x - x = 0$. Repeat for $\sin 2\pi x - .5x = 0$.
2. Compute and compare the domains of convergence of Newton's method applied to (5.7.2) and (5.7.3) for initial guesses in $[0, 2]^2$. Did the nonlinear transformation in (5.7.3) increase the domain of convergence? Also, is the average rate of convergence over these domains greater for (5.7.2) or (5.7.3)? To do this exercise, develop your own concept of the "average rate of convergence" over a domain. Also use a graphical presentation similar to that in figure 5.7 in section 5.4.
3. Consider the CGE problem in section 5.10 with

$$u^i(x) = \sum_{j=1}^m a_j^i x_j^{v_j^i + 1} (1 + v_j^i)^{-1}.$$

Assume that $a_j^i, e_j^i > 0 > v_j^i$, $i = 1, \dots, n, j = 1, \dots, m$. Write programs that take the a_j^i, v_j^i , and e_j^i coefficients as inputs, and compute the pure exchange equilibrium price vector $p_j, j = 1, \dots, m$, and consumption of good j for agent i , c_j^i . For purposes of testing, make random choices for $a_j^i \in [1, 10]$, $v_j^i \in [-0.5, -3]$, and $e_j^i \in [1, 5]$. Initially start with two goods and two agents. Then try larger systems. Can you handle $m = n = 5$? $m = n = 10$?

- a. Write a program using Newton's method applied to the first-order conditions. How often do you get convergence?
- b. Find a program using Powell's hybrid method (e.g., HYBRD in Minpack) applied to the first-order conditions. Compare the performance of Newton's method and the hybrid method.
- c. Write a program using the Negishi approach.
- d. Write a program using the function iteration approach applied to g defined in (5.10.2).
- e. If all agents are identical, then it is trivial to compute equilibrium. Write a program that uses simple continuation to move from an identical agent case to the inputted case. Does simple continuation usually work?

(For $v_j^i = -1$, we replace $x_j^{v_j^i+1} (1 + v_j^i)^{-1}$ with $\ln x_j$. If you do not want to program this special case, you can approximate $v_j^i = -1$ with, for example, $v_j^i = -1.01$.)

4. Assume there are n individual investors and m assets with asset j paying Z_s^j dollars with probability π_s^j , $s = 1, \dots, S$, in state s . Let e_l^j denote agent l 's endowment of asset j . Assume that asset 1 is safe, paying R dollars tomorrow per unit held today. Suppose that investor l maximizes expected utility, $E\{u_l(W)\}$, over final wealth, W ; let $u_l(W) = W^{1+\gamma_l}/(1 + \gamma_l)$ where $\gamma_l < 0$. Write a program that reads in data on tastes, returns, and endowments, and computes the equilibrium price of the assets and the expected utility of each investor. Begin with three agents ($n = 3$), one safe asset paying 1, and two risky assets ($m = 3$). Assume that one risky asset pays 0.8, 1.0, 1.1, or 1.2 with equal probability, that the other risky asset pays 0.1, 0.8, 1.1, or 1.3 with equal probability, and that the two risky assets are independent.
5. Repeat the duopoly example of section 5.4, but assume three firms, concave, increasing utility functions of the form $u(x, y, z) = (a + x^\alpha + y^\beta + z^\gamma)^\delta + M$, and unit cost equaling 1. Compute and compare the Bertrand and Cournot outcomes for a variety of combinations of $a \in [0.1, 2]$ and $\alpha, \beta, \gamma, \delta \in [0.1, 0.9]$. Compute the change in profits for firms x and y if they merge; compare the results for the Bertrand and Cournot cases.

6 Approximation Methods

Many computational problems require us to approximate functions. In some cases we have a slow and complex way to compute a function and want to replace it with a simpler, more efficient approximating function. To do this, we compute the value of the function at only a few points and make guesses about its values elsewhere. In many interesting cases the function of interest is unknown; we then approximate it by generating a finite amount of information about it and use the information to construct an approximating function. Choosing the best way to accomplish this is often an essential aspect of an efficient computational strategy.

We examine three types of approximation problems in this chapter, representing three different kinds of data and objectives. A *local approximation* takes as data the value of a function f and its derivatives at a point x_0 and constructs a function that matches those properties at x_0 and is (hopefully) a good approximation of f near x_0 . Taylor series and Padé expansions are the major local approximation methods.

An L^p approximation finds a “nice” function g that is “close to” a given function f over some interval in the sense of a L^p norm. To compute an L^p approximation of f , we ideally need the entire function, an informational requirement that is generally infeasible. *Interpolation* is any procedure that finds a “nice” function that goes through a collection of prescribed points. Interpolation and L^p approximation are similar except that the data in interpolation are a finite set of points; in fact interpolation uses n facts to fix n free parameters.

Regression lies between L^p approximation and interpolation in that it uses m facts to fix n parameters, $m > n$, producing an approximation that only nearly fits the data. Regression methods are more stable but use more information than interpolation methods, and they are less accurate in that they use less information than the full L^p approximation.

In this chapter we will present the formal implementations of these ideas. In all cases we need to formalize the notions of “nice” and “close to.” We introduce orthogonal polynomials, splines, and their approximation properties. Economists often deal with functions that they know to be nonnegative, or concave, for example; we present methods that preserve such information about shape. Much of approximation theory is based on linear methods. Flexible, nonlinear approximation is a difficult topic, but one promising approach is *neural networks*. We discuss neural networks and their approximation properties.

6.1 Local Approximation Methods

Local approximation methods use information about a function $f: R \rightarrow R$ only at a point, $x_0 \in R$, but that information may include several derivatives of f at x_0 as well

as $f(x_0)$. The objective is to produce locally good approximations with little effort; for many important classes of functions, these local procedures produce approximations that are also globally useful. The two most common such procedures are Taylor series and Padé approximation.

Taylor Series Approximation

The theoretical basis for all local approximations is Taylor's theorem, theorem 1.6.1. The basic point of Taylor's theorem is that we can take a n -times differentiable function, f , form the polynomial approximation

$$\begin{aligned} f(x) \doteq f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2}f''(x_0) + \dots \\ + \frac{(x - x_0)^n}{n!}f^{(n)}(x_0) \end{aligned} \tag{6.1.1}$$

and be assured that the error is $\mathcal{O}(|x - x_0|^{n+1})$ and hence asymptotically smaller than any of the terms. The *degree n Taylor series approximation to f* is the right-hand side of (6.1.1). Taylor series approximations of f are polynomials that agree with f to a high order near x_0 . Taylor's theorem only says that this approximation is good near x_0 ; its accuracy may decay rapidly as we move away from x_0 .

As we take $n \rightarrow \infty$ (assuming that $f \in C^\infty$), (6.1.1) is potentially a power series representation for f near x_0 . The key fact about power series is that if $\sum_{n=0}^{\infty} a_n z^n$ converges for z_0 , then it converges absolutely for all complex numbers z such that $|z| < |z_0|$. If we define

$$r = \sup \left\{ |z| : \left| \sum_{n=0}^{\infty} a_n z^n \right| < \infty \right\},$$

then r is called the *radius of convergence*, and $\sum_{n=0}^{\infty} a_n z^n$ converges for all $|z| < r$ and diverges for all $|z| > r$.

The properties of power series are studied extensively in complex analysis. We will give some basic definitions from complex analysis and state some basic theorems without proof. We provide them so that we can discuss various aspects of approximation in a precise fashion. First, let Ω be any nonempty, connected open set in the complex plane. A function $f: \Omega \subset C \rightarrow C$ on the complex plane C is *analytic* on Ω iff for every $a \in \Omega$, there is an r and a sequence c_k such that $f(z) = \sum_{k=0}^{\infty} c_k (z - a)^k$ whenever $\|z - a\| < r$. A *singularity of f* is any point a such that f is analytic on $\Omega - \{a\}$ but not on Ω . A point $a \in C$ is a *pole of f* iff a is a singularity of f , but

$f(z)(z - a)^m$ is analytic on Ω for some m . The pole has *multiplicity* m iff $f(z)(z - a)^m$ is analytic on Ω , but $f(z)(z - a)^{m-1}$ is not. A function $f: C \rightarrow C$ is *meromorphic* iff it is analytic on an open set in the complex plane outside of a nowhere dense set of poles.

The following theorem states a basic property of analytic functions:

THEOREM 6.1.2 Let f be analytic at $x \in C$. If f or any derivative of f has a singularity at $z \in C$, then the radius of convergence in the complex plane of the Taylor series based at x_0 , $\sum_{n=0}^{\infty} ((x - x_0)^n / n!) f^{(n)}(x_0)$, is bounded above by $\|x_0 - z\|$.

Theorem 6.1.2 tells us that the Taylor series at x_0 cannot reliably approximate $f(x)$ at any point farther away from x_0 than any singular point of f . This fact will be important for economic applications, since utility and production functions often satisfy an Inada condition, a singularity, at some point. Note that it is distance that matters, not direction. Therefore, if $f(x) = x^{-1/2}$, the singularity at $x = 0$ means that the Taylor series based at $x_0 = 1$ cannot be accurate for $x > 2$. Also even a singularity at a complex point affects convergence for real x .

We illustrate the quality of a Taylor expansion with a function familiar to economists. Suppose that $f(x) = x^\alpha$, where $0 < \alpha < 1$. This function has its only singularity at $x = 0$, implying that the radius of convergence for the Taylor series around $x = 1$ is only unity. The coefficients in the Taylor series decline slowly, since

$$a_k = \frac{1}{k!} \left. \frac{d^k}{dx^k} (x^\alpha) \right|_{x=1} = \frac{\alpha(\alpha - 1) \cdots (\alpha - k + 1)}{1 \cdot 2 \cdots k}.$$

For example, if $\alpha = 0.25$, then for $k < 50$, $|a_k| > 0.001$. Table 6.1 displays the error, $|x^{1/4} - \sum_{k=0}^N a_k (x - 1)^k|$, for various values of x and N .

Table 6.1 shows that the Taylor series is very good near $x = 1$ but that its quality falls sharply as x falls away from $x = 1$. Even a 50th-order Taylor series approximation achieves only two-digit accuracy at $x = 0.05$. Also note that the Taylor series is worthless for $x > 2$. In fact outside $[0, 2]$ the error grows as we increase the degree; for example, the error at $x = 3$ is of order 10^{12} for the 50th-order expansion.

Rational Approximation

The k th-degree Taylor series approximation is the degree k polynomial that matches f at x_0 up to the k th derivative. When stated this way, we are led to ask if we can do better by matching some other functional form to f and its derivatives at x_0 . *Padé approximation* is a powerful alternative that uses the local information to construct a *rational function*, a ratio of polynomials, to approximate f at x_0 .

Table 6.1
Taylor series approximation errors for $x^{1/4}$

x	N				$x^{1/4}$
	5	10	20	50	
3.0	5(-1)	8(1)	3(3)	1(12)	0.9457
2.0	1(-2)	5(-3)	2(-3)	8(-4)	0.9457
1.8	4(-3)	5(-4)	2(-4)	9(-9)	0.9457
1.5	2(-4)	3(-6)	1(-9)	0(-12)	0.9457
1.2	1(-6)	2(-10)	0(-12)	0(-12)	0.9457
0.80	2(-6)	3(-10)	0(-12)	0(-12)	0.9457
0.50	6(-4)	9(-6)	4(-9)	0(-12)	0.8409
0.25	1(-2)	1(-3)	4(-5)	3(-9)	0.7071
0.10	6(-2)	2(-2)	4(-3)	6(-5)	0.5623
0.05	1(-1)	5(-2)	2(-2)	2(-3)	0.4729

DEFINITION A (m, n) Padé approximant of f at x_0 is a rational function

$$r(x) = \frac{p(x)}{q(x)},$$

where $p(x)$ and $q(x)$ are polynomials, the degree of p is at most m , the degree of q is at most n , and

$$0 = \frac{d^k}{dx^k} (p - f q)(x_0), \quad k = 0, \dots, m+n.$$

One usually chooses $m = n$ or $m = n + 1$. The defining conditions can be rewritten as $q(x_0) = 1$ and

$$p^{(k)}(x_0) = (f q)^{(k)}(x_0), \quad k = 0, \dots, m+n, \tag{6.1.2}$$

which are linear in the $m+1$ unknown coefficients of p and the $n+1$ coefficients of q once the derivatives of f at x_0 are computed. The $m+n+1$ conditions in (6.1.2) together with the normalization $q(x_0) = 1$ form $m+n+2$ linear conditions on the $m+n+2$ coefficients of p and q . The linear system may be singular; in that case one reduces n or m by 1 and tries again.

In practice, we replace f by its degree $m+n$ Taylor series and equate the coefficients of like powers; that is, if $t(x)$ is the degree $m+n$ Taylor series of f based at x_0 , then we find $p(x)$ and $q(x)$ such that $p^{(k)}(x_0) = (t(x)q(x))^{(k)}(x_0)$ for all $k = 0, \dots, m+n$, which is equivalent to finding $p(x)$ and $q(x)$ such that the coefficients of

Table 6.2
Padé approximation errors for $x^{1/4}$

x	(m, n)				
	(3, 2)	(5, 5)	(10, 10)	(9, 1)	(4, 1)
3.0	7(-4)	1(-6)	2(-12)	1(-2)	6(-3)
2.0	5(-5)	6(-9)	0(-12)	2(-4)	2(-4)
1.8	2(-5)	1(-9)	0(-12)	2(-6)	8(-5)
1.5	2(-6)	2(-11)	0(-12)	2(-8)	8(-6)
1.2	1(-8)	0(-12)	0(-12)	2(-12)	5(-8)
0.80	4(-8)	0(-12)	0(-12)	5(-12)	1(-7)
0.50	3(-5)	4(-9)	0(-12)	3(-7)	7(-5)
0.25	1(-3)	5(-6)	1(-10)	1(-4)	2(-3)
0.10	1(-2)	6(-4)	8(-7)	4(-3)	2(-2)
0.05	4(-2)	4(-3)	5(-5)	2(-2)	5(-2)

$1, x, x^2, \dots, x^{m+n}$ in $p(x) - t(x)q(x)$ are all zero. To illustrate the method, we compute the $(2, 1)$ Padé approximation of $x^{1/4}$ at $x = 1$. We first construct the degree $2 + 1 = 3$ Taylor series for $x^{1/4}$ at $x = 1$:

$$t(x) = 1 + \frac{(x-1)}{4} - \frac{3(x-1)^2}{32} + \frac{7(x-1)^3}{128}.$$

If $p(x) \equiv p_0 + p_1(x-1) + p_2(x-1)^2$, and since $q_0 = 1$, $q(x) \equiv 1 + q_1(x-1)$, we are searching for p_0, p_1, p_2 , and q_1 such that

$$p_0 + p_1(x-1) + p_2(x-1)^2 - t(x)(1 + q_1(x-1)) = 0 \quad (6.1.3)$$

Combining coefficients of like powers in (6.1.3) implies a set of linear equations for p_0, p_1, p_2 , and q_1 that produces, after multiplying p and q by 40, the $(2, 1)$ Padé approximation

$$\frac{21 + 70x + 5x^2}{40 + 56x}. \quad (6.1.4)$$

In table 6.2 we report the errors of various Padé approximants to $x^{1/4}$ based at $x_0 = 1$. Comparing these results to those in table 6.1 shows that both are excellent for x close to x_0 but that the Padé approximant is much better for x farther from x_0 even for $x = 2$ and $x = 3$. The range of validity for the Padé approximant on $[1, \infty)$ is not limited by the singularity at $x = 0$.

The straightforward linear equation approach we used to solve for the Padé coefficients in (6.1.4) is satisfactory for some cases but may lead to poorly conditioned

linear systems if we use high-degree polynomials in the numerator and denominator. See Cuyt and Wuytack (1986) for better methods of constructing higher-order approximations.

The general experience is that Padé approximants of “well-behaved” functions are better global approximants than Taylor series approximations; that is, the error grows less rapidly as we move away from x_0 . The following theorem from Cuyt and Wuytack¹ covers some important cases.

THEOREM 6.1.3 Let $P_{m,n}$ be the (m,n) Padé approximant of $f(z)$, a meromorphic function on $B \equiv \{z \mid \|z\| \leq r\} \subset C$. Then for every $\delta, \varepsilon > 0$ there is a k such that for all $j > k$ the Lebesgue measure of $\{z : \|f(z) - P_{j,1}(z)\| > \varepsilon\}$ is less than δ . Furthermore, if the poles of f are finite in number and have total multiplicity of n , then the sequence $P_{j,n}$ converges uniformly as $j \rightarrow \infty$ to f on every closed and bounded subset of B outside of the poles.

These are useful results confirming the convergence of Padé approximants for important classes of functions. For example, it applies to $x^{1/4}$ on the right half of the complex plane; since its only pole is at the origin, the convergence of $P_{m,1}$ to $x^{1/4}$ as m increases is assured on each bounded interval on the positive real line. Table 6.2 shows that Padé approximation generally works well for $x^{1/4}$. We would like stronger results, such as $P_{m,m-1}$ converging on the real line, but the full convergence theory of Padé approximation is beyond the scope of this text; see Cuyt and Wuytack (1986) and Braess (1986) for treatments of these issues. In practice, we construct Padé expansions for a function and then check its quality, usually finding that it does well.

Log-Linearization, Log-Quadraticization, etc.

Economists often want to express results in unit-free form in terms of elasticities and shares. One way to accomplish this is to *log-linearize* an equation at a point. Suppose that we know a solution $f(x_0, \varepsilon_0) = 0$, and we want to compute the elasticity $(dx/x)/(d\varepsilon/\varepsilon)$ at $x = x_0$ and $\varepsilon = \varepsilon_0 \neq 0$. By implicit differentiation, $f_x dx + f_\varepsilon d\varepsilon = 0$, which implies that

$$\hat{x} = \frac{dx}{x} = -\frac{\varepsilon f_\varepsilon}{x f_x} \frac{d\varepsilon}{\varepsilon} = -\frac{\varepsilon f_\varepsilon}{x f_x} \hat{\varepsilon},$$

where we use the caret notation, $\hat{z} \equiv dz/z$. The term $\varepsilon f_\varepsilon / x f_x$ evaluated at $x = x_0$ and $\varepsilon = \varepsilon_0$ is unit-free, and generally can be decomposed into elasticities and shares.

1. See discussion in Cuyt and Wuytack (1986, pp. 97–99).

Since $\hat{x} = d(\ln x)$, log-linearization implies the log-linear approximation

$$\ln x - \ln x_0 \doteq -\frac{\varepsilon_0 f_\varepsilon(x_0, \varepsilon_0)}{x_0 f_x(x_0, \varepsilon_0)} (\ln \varepsilon - \ln \varepsilon_0). \quad (6.1.5)$$

The log-linear expression in turn implies that

$$x \doteq x_0 \exp\left(-\frac{\varepsilon_0 f_\varepsilon(x_0, \varepsilon_0)}{x_0 f_x(x_0, \varepsilon_0)} (\ln \varepsilon - \ln \varepsilon_0)\right), \quad (6.1.6)$$

which is a constant elasticity approximation of $x(\varepsilon)$ for ε near ε_0 . The expressions (6.1.5) and (6.1.6) can be compared to the linear approximation

$$x \doteq x_0 + (\varepsilon - \varepsilon_0) \left(-\frac{f_\varepsilon(x(\varepsilon_0), \varepsilon_0)}{f_x(x(\varepsilon_0), \varepsilon_0)}\right).$$

The log-linear approximation has the advantage of producing expressions based on shares and elasticities; this is useful when deriving qualitative formulas and in empirical work where one wants to express results in terms of shares and elasticities. However, there is no general reason for either the linear or log-linear approximation being a more accurate approximation than the other. They may appear to provide us with different information, but the differences concern higher-order information about f which is not contained in the first-order information used in (6.1.5) and (6.1.6).

It may appear difficult to extend the “log-linearized” first-order method to a “log-quadraticized” second-order approximation. However, once we realize that both methods are essentially the same, we see that we can arbitrarily generalize log-linearization.

The key observation is that we are just using a nonlinear change of variables. Suppose that we have $Y(X)$ implicitly defined by $f(Y(X), X) = 0$. Define $x = \ln X$ and $y = \ln Y$, then $y(x) = \ln Y(e^x)$. Furthermore the equation $f(Y(X), X) = 0$ is equivalent to $f(e^{y(x)}, e^x) \equiv g(y(x), x) = 0$. Implicit differentiation of $g(y(x), x) = 0$ will produce $y'(x) = (d \ln Y)/(d \ln X)$ and the log-linear approximation (6.1.5) applies. The expression $\ln Y(X) = y(x)$ suggests the second-order approximation

$$\ln Y(X) = y(x) \doteq y(x_0) + y'(x)(x - x_0) + y''(x_0) \frac{(x - x_0)^2}{2}, \quad (6.1.7)$$

which expresses $\ln Y$ in terms of a polynomial in $x = \ln X$. Again we have no reason to prefer (6.1.7) over a quadratic expansion in X . In any specific case they may differ; it may be sensible to compute both and then pick whichever one does better at satisfying the implicit relation $f(Y(X), X) = 0$.

These ideas can also be applied to constructing Padé expansions in terms of the logarithm. One could construct approximations that are ratios of polynomials in the logarithm. We also see that there is nothing special about log function. We could take any monotonic $h(\cdot)$, define $x = h(X)$ and $y = h(Y)$, and use the identity

$$\begin{aligned} f(Y, X) &= f(h^{-1}(h(Y)), h^{-1}(h(X))) \\ &= f(h^{-1}(y), h^{-1}(x)) \equiv g(y, x). \end{aligned}$$

The log expansion where $h(z) = \ln z$ is natural for economists, since it delivers expressions in terms of elasticities and shares. However, it is not the only possibility, and others may be better when we are using Taylor series expansions as numerical approximations.

6.2 Ordinary Regression as Approximation

Economists are familiar with one common form of global approximation, that being least squares regression. In this section we discuss ordinary regression as an approximation method, the statistical context with which we are most familiar, and show how it can be adjusted to form efficient approximation methods in numerical contexts.

Regression analysis studies the function $f(x) = E\{y|x\}$, the expectation of the dependent variable, $y \in R$, conditional on the variables, $x \in R^k$. The task of econometric analysis is to derive an approximation of f using n data points, $(y_i, x^i) \in R \times R^k$, $i = 1, \dots, n$, and $m < n$ functions, $\phi_j: R^k \rightarrow R$, $j = 1, \dots, m$. *Ordinary least squares regression* (OLS) fits the data with the functions ϕ_j by solving the problem

$$\min_{a_1, a_2, \dots, a_m} \sum_{i=1}^n (a_1 \phi_1(x^i) + a_2 \phi_2(x^i) + \dots + a_m \phi_m(x^i) - y_i)^2. \quad (6.2.1)$$

The a_j coefficients that solve (6.2.1) defines a function, $\hat{f}(x) \equiv \sum_{j=1}^m a_j \phi_j(x)$ such that $\hat{f}(x^i)$ is “close to” y_i for each i . Generally, $y_i \neq \hat{f}(x^i)$, but the coefficients are chosen to minimize the sum of squared errors. More generally, $\hat{f}(x)$ is a curve that hopefully approximates the true function $f(x)$ over all points x in some region of interest.

In numerical analysis we will also want to generate approximations, $\hat{f}(x)$, to unknown or known functions and, as with OLS, do so with only the finite amount of information represented by (y_i, x^i) points. We could use simple regression. However, we can generally do much better because the context of our numerical approximation problems differs substantially from the statistical context of econometric problems.

The first key difference is that the econometrician has no control over the x^i points in his data set, whereas the numerical analyst will be able to choose the x^i he uses. Second, the econometrician has to deal with significant error in the (y_i, x^i) data, since $y_i - f(x^i)$ will generally be nontrivial; in numerical analysis we will be able to control this error, often making it trivial.

These features allow us to do things in numerical analysis that are unacceptable in econometrics. For example, econometricians would never solve (6.2.1) with $m = n$; the resulting approximations would generally be silly and unreliable. However, we will see that when the error $y_i - f(x^i)$ is small, we can often construct reliable exact fits through *careful* choices of the x^i values.

In the rest of the chapter we describe several approximation methods available. Even in one dimension we find that they do significantly better than simple regression; in several dimensions the advantage increases exponentially. These more powerful methods build on the same basic ideas as ordinary least squares approximation; therefore much of this material will be familiar. We will see, however, that by pushing these ideas a bit and analyzing them in the context of numerical, not statistical, problems, we can develop much more efficient approximation methods.

6.3 Orthogonal Polynomials

We next use basic vector space ideas to construct representations of functions that lead to good approximations. Since the space of continuous functions is spanned by the monomials, x^n , $n = 0, 1, 2, \dots$, it is natural to think of the monomials as a basis for the space of continuous functions. However, good bases for vector spaces possess useful orthogonality properties. We will develop those orthogonality ideas to construct *orthogonal polynomials*.

DEFINITION A *weighting function*, $w(x)$, on $[a, b]$ is any function that is positive almost everywhere and has a finite integral on $[a, b]$. Given a weighting function $w(x)$, we define the inner product

$$\langle f, g \rangle = \int_a^b f(x) g(x) w(x) dx.$$

The family of polynomials $\{\varphi_n(x)\}$ is *mutually orthogonal* with respect to the weighting function $w(x)$ if and only if $\langle \varphi_n, \varphi_m \rangle = 0$ for $n \neq m$. They are *mutually orthonormal* if they are mutually orthogonal and $\langle \varphi_n, \varphi_n \rangle = 1$ for all n . The inner product defines a norm $\|f\|^2 = \langle f, f \rangle$ on the functions f for which $\langle f, f \rangle$ is finite.

Table 6.3
Common families of orthogonal polynomials

Family	$w(x)$	$[a, b]$	Definition
Legendre	1	$[-1, 1]$	$P_n(x) = \frac{(-1)^n}{2^n n!} \frac{d^n}{dx^n} [(1 - x^2)^n]$
Chebyshev	$(1 - x^2)^{-1/2}$	$[-1, 1]$	$T_n(x) = \cos(n \cos^{-1} x)$
General Chebyshev	$\left(1 - \left(\frac{2x - a - b}{b - a}\right)^2\right)^{-1/2}$	$[a, b]$	$T_n\left(\frac{2x - a - b}{b - a}\right)$
Laguerre	e^{-x}	$[0, \infty)$	$L_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n} (x^n e^{-x})$
Hermite	e^{-x^2}	$(-\infty, \infty)$	$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} (e^{-x^2})$

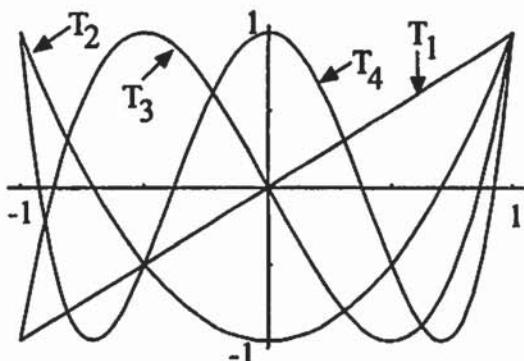


Figure 6.1
Chebyshev polynomials

We assume that all the nonnegative powers of x are integrable with respect to w . Since the resulting orthogonal polynomial families are orthogonal bases of the space of polynomials, they can be generated using the Gram-Schmidt procedure. Below we will find simpler ways to generate orthogonal polynomials.

There are several examples of orthogonal families of polynomials, each defined by a different weighting function. Some common ones are described in table 6.3.

Figures 6.1 through 6.4 display the degree 1, 2, 3, and 4 members of the first four of four orthogonal families. The first member of each family is the constant function $x = 1$; we do not display it in these figures. Note that each Chebyshev polynomial has a maximum of 1 and a minimum of -1 . Legendre polynomials are roughly similar to Chebyshev polynomials in appearance, but they do not bounce between 1 and -1 . Both the Laguerre and Hermite polynomials are defined over infinite domains and diverge as $|x| \rightarrow \infty$. This divergence is not important in their respective inner prod-

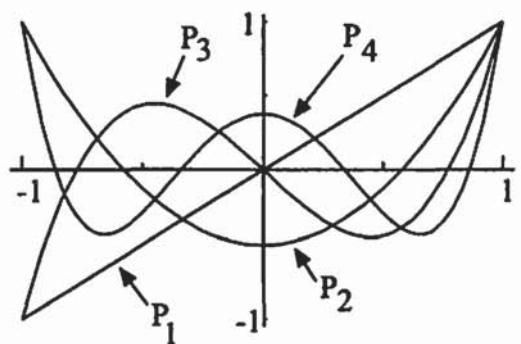


Figure 6.2
Legendre polynomials

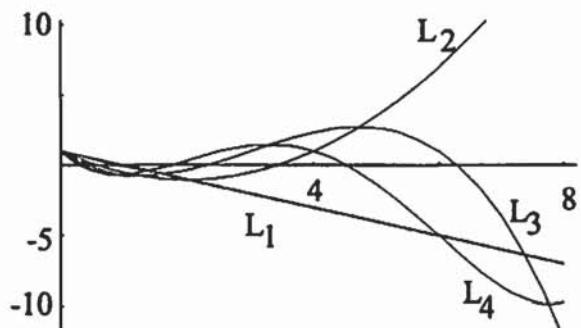


Figure 6.3
Laguerre polynomials

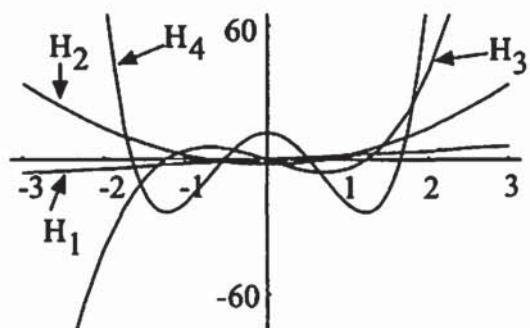


Figure 6.4
Hermite polynomials

ucts because the exponentially vanishing weighting functions will dominate any polynomial divergence.

Recursion Formulas

The formulas defining orthogonal families in table 6.3 are complex and too costly to compute; in fact some of the expressions in table 6.3 don't look like polynomials. Fortunately each family satisfies a simple recursive scheme. First note that $P_0(x) = T_0(x) = L_0(x) = H_0(x) = 1$, $P_1(x) = T_1(x) = x$, $L_1(x) = 1 - x$, and $H_1(x) = 2x$ are the first and second members of the Legendre, Chebyshev, Laguerre, and Hermite families. With these initial choices, these orthogonal polynomial families satisfy the following recursive schemes:

$$P_{n+1}(x) = \frac{2n+1}{n+1} x P_n(x) - \frac{n}{n+1} P_{n-1}(x), \quad (6.3.1)$$

$$L_{n+1}(x) = \frac{1}{n+1} (2n+1-x) L_n(x) - \frac{n}{n+1} L_{n-1}(x), \quad (6.3.2)$$

$$T_{n+1}(x) = 2x T_n(x) - T_{n-1}(x), \quad (6.3.3)$$

$$H_{n+1}(x) = 2x H_n(x) - 2n H_{n-1}(x). \quad (6.3.4)$$

Recursive formulas yield efficient ways of evaluating orthogonal polynomials which are faster than the direct definitions given in table 6.3. Algorithm 6.1 illustrates the procedure for evaluating a finite sum of Chebyshev polynomials, $\sum_{i=0}^n a_i T_i(x)$, at a single point, x .

Algorithm 6.1 Chebyshev Evaluation Algorithm

Objective: Evaluate $\sum_{i=0}^n a_i T_i(x)$.

Step 1. Evaluate the Chebyshev polynomials at x by recursion:

$$T_0(x) = 1; \quad T_1(x) = x,$$

$$T_{i+1}(x) = 2x T_i(x) - T_{i-1}(x), \quad i = 1, \dots, n.$$

Step 2. Construct the sum at x , $\sum_{i=0}^n a_i T_i(x)$.

This recursive approach is a relatively efficient way to compute $\sum_{i=0}^n a_i T_i(x)$. For example, if one wanted the value of the first twelve Chebyshev polynomials at x , only twenty multiplications and ten subtractions are needed. If one instead evaluated each polynomial separately using the trigonometric formula in table 6.3, the operation

count would be much worse at ten cosine and ten arc cosine evaluations. If one is to compute $\sum_{i=0}^n a_i T_i(x)$ for many values of x , it may seem better to compute the b_i coefficients such that $\sum_{i=0}^n a_i T_i(x) = \sum_{i=0}^n b_i x^i$. Computing $\sum_{i=0}^n a_i T_i(x)$ in the manner above is often more accurate because of round-off error problems, particularly if n is large. If that is not important, then using the $\sum_{i=0}^n b_i x^i$ representation is better because Horner's method can be used.

One can generally use the recursive schemes in (6.3.1)–(6.3.4) to accurately compute the polynomials in table 6.3. The recursive definitions for these orthogonal families are not special since recursion formulas exist for any orthogonal family, as demonstrated in theorem 6.3.1.

THEOREM 6.3.1 Let $\{\varphi_n(x)\}_{n=0}^\infty$ be an orthogonal family on $[a, b]$ relative to an inner product $\langle \cdot, \cdot \rangle$, $\varphi_0(x) = 1$, and $\varphi_1(x) = x - \langle x, 1 \rangle / \langle x, x \rangle$. Then $\{\varphi_n(x)\}_{n=0}^\infty$ satisfies the recursive scheme

$$\varphi_{n+1}(x) = (x - \delta_n) \varphi_n(x) - \gamma_n \varphi_{n-1}(x) \quad (6.3.5)$$

where δ_n and γ_n are defined by

$$\delta_n = \frac{\langle x \varphi_n, \varphi_n \rangle}{\langle \varphi_n, \varphi_n \rangle}, \quad \gamma_n = \frac{\langle \varphi_n, \varphi_n \rangle}{\langle \varphi_{n-1}, \varphi_{n-1} \rangle}.$$

Proof See Young and Gregory (1988, pp. 319–21). ■

If one were using a special purpose basis different from one of the standard families, then theorem 6.3.1 produces the corresponding recursion formulas.

6.4 Least Squares Orthogonal Polynomial Approximation

Before describing approximation algorithms, we first develop the critical concepts and theory. In the next two sections, we ask how well orthogonal polynomials can approximate functions. This material will give us a language with which we can discuss approximation ideas and also give us some idea of how well we can do.

Given $f(x)$ defined on $[a, b]$, one approximation concept is least squares approximation. That is, given $f(x)$, the *least squares polynomial approximation of f with respect to weighting function $w(x)$* is the degree n polynomial that solves

$$\min_{\deg(p) \leq n} \int_a^b (f(x) - p(x))^2 w(x) dx. \quad (6.4.1)$$

In this problem the weighting function $w(x)$ indicates how much we care about

approximation errors as a function of x . For example, if we have no preference over where the approximation is good (in a squared-error sense), then we take $w(x) = 1$.

The connections between orthogonal polynomials and least squares approximation are immediately apparent in solving for the coefficients of $p(x)$ in the least squares approximation problem. If $\{\varphi_n\}_{n=0}^{\infty}$ is an orthogonal sequence with respect to a weighting function on $[a, b]$, $w(x)$, the solution to (6.4.1) can be expressed

$$p(x) = \sum_{k=0}^n \frac{\langle f, \varphi_k \rangle}{\langle \varphi_k, \varphi_k \rangle} \varphi_k(x). \quad (6.4.2)$$

Note the similarity between least squares approximation and regression. The formula for $p(x)$ is essentially the same as regressing the function $f(x)$ on $n+1$ orthogonal regressors that are the $\varphi_k(x)$ functions; the coefficient of the k th regressor, $\varphi_k(x)$, equals the “covariance” between f and the k th regressor divided by the “variance” of the k th regressor. This is no accident, since regression is a least squares approximation. The difference in practice is that regression is limited to those observations of the regressors, the $\varphi_k(x)$, that real data give the analyst, whereas in L^p approximation (6.4.2) evaluates the regressors at all x .

Any polynomial can be easily expressed as a linear sum of orthogonal polynomials. Specifically, if $\{\varphi_n(x)\}_{n=0}^{\infty}$ is an orthogonal collection of polynomials with $\deg(\varphi_n) = n$, then for any polynomial $f(x)$ of degree n ,

$$f(x) = \sum_{k=0}^n \frac{\langle f, \varphi_k \rangle}{\langle \varphi_k, \varphi_k \rangle} \varphi_k(x). \quad (6.4.3)$$

Fourier Approximation

One common form of least squares approximation is Fourier approximation. Fourier approximation theorems tell us that if f is continuous on $[-\pi, \pi]$ and $f(-\pi) = f(\pi)$, then

$$f(\theta) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos(n\theta) + \sum_{n=1}^{\infty} b_n \sin(n\theta) \quad (6.4.4)$$

where the *Fourier coefficients* are defined by

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(\theta) \cos(n\theta) d\theta, \quad b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(\theta) \sin(n\theta) d\theta,$$

and convergence in (6.4.4) is uniform. Since the sine and cosine functions satisfy

the necessary orthogonality conditions on $[-\pi, \pi]$, trigonometric polynomials fit our orthogonal function framework. A *trigonometric polynomial* is any function of the form in (6.4.4).

Fourier approximation is particularly important when we are approximating a *periodic function*, that is, a function $f: R \rightarrow R$ such that for some ω , $f(x) = f(x + \omega)$ for all x , which is also smooth. Nonperiodic functions may also be approximated by their Fourier series, (6.4.4). However, convergence is not uniform, and many terms are needed for good fits.

Chebyshev Least Squares Approximation

Chebyshev polynomials are a special class of polynomials with many useful strong properties. One hint of their special position is given by the formula $T_n(x) = \cos(n \cos^{-1} x)$. Note that $T_n(\cos \theta) = \cos n\theta$ is an orthogonal sequence on $[0, 2\pi]$ with respect to $w(x) = 1$. This is closely related to Fourier analysis, the dominant way to approximate smooth periodic functions. Given this connection, it is not surprising that Chebyshev approximations are good for smooth nonperiodic functions. The next theorem states this precisely.

THEOREM 6.4.1 Assume $f \in C^k [-1, 1]$. Let

$$c_j \equiv \frac{2}{\pi} \int_{-1}^1 \frac{f(x) T_j(x)}{\sqrt{1 - x^2}} dx, \quad (6.4.5)$$

$$C_n(x) \equiv \frac{1}{2} c_0 + \sum_{j=1}^n c_j T_j(x). \quad (6.4.6)$$

Then there is a $B < \infty$ such that for all $n \geq 2$,

$$\|f - C_n\|_\infty \leq \frac{B \ln n}{n^k}. \quad (6.4.7)$$

Hence $C_n \rightarrow f$ uniformly as $n \rightarrow \infty$. We next present a useful theorem on Chebyshev coefficients.

THEOREM 6.4.2 If $f \in C^k [-1, 1]$ has a Chebyshev representation

$$f(x) = \frac{1}{2} c_0 + \sum_{j=1}^{\infty} c_j T_j(x), \quad (6.4.8)$$

then there is a constant c such that

$$|c_j| \leq \frac{c}{j^k}, \quad j \geq 1. \quad (6.4.9)$$

Hence the Chebyshev coefficients eventually drop off rapidly for smooth functions.

In practice, we never have the infinite series (6.4.8), and we never have a perfect way to calculate f . In practice, we must approximate f with a finite sum, $(1/2)c_0 + \sum_{j=1}^n c_j T_j(x)$ for some n . The growth condition on the coefficients (6.4.9) can be useful in choosing n . If the c_k coefficients, $k < n$, are rapidly falling and c_n is small, then we can be more confident that the ignored coefficients make a trivial contribution, and that the $(n+1)$ -term finite series is an acceptable approximation to f . If c_n is not much less than c_{n-1} , then we should extend the series beyond the T_n term. This is done even if c_n is small. The reason is that if the coefficients are not trailing off rapidly, the sequence of coefficients that are ignored, c_k , $k > n$, may add up to be important. We often will not know the character of the f that we are approximating; therefore, if the Chebyshev coefficients are not dropping off rapidly, that may indicate that f is not as smooth as required by theorem 6.4.2. In that case we should move to a different approximation approach, such as splines.

At this point, we should note that computing Chebyshev least squares coefficients is difficult because the integral in (6.4.5) generally does not have an analytic solution. The numerical solution of integrals is taken up in the next chapter.

Approximations over General Intervals

We usually need to find polynomial approximations over intervals other than $[-1, 1]$. Suppose that we have a function, $f(y)$, $f: [a, b] \rightarrow R$ and want to compute an orthogonal polynomial approximation over $[a, b]$ that corresponds to the orthogonal family over $x \in [-1, 1]$ with weighting function $w(x)$. To do this, we define the transformation $X(y) = -1 + 2(y-a)/(b-a) = (2y - a - b)/(b - a)$ and the inverse transform $Y(x) = ((x+1)(b-a))/2 + a$. We define the function $g(x) = f(Y(x))$. If $g(x) = \sum_{i=0}^{\infty} a_i \varphi_i(x)$ where the φ_i are orthogonal over $[-1, 1]$ with respect to $w(x)$, then

$$f(y) = \sum_{i=0}^{\infty} a_i \varphi_i(X(y)) \quad (6.4.10)$$

is an orthogonal representation with respect to the weight $w(X(y))$ over $[a, b]$ since $\int_a^b \varphi_i(X(y)) \varphi_j(X(y)) w(X(y)) dy = 0$ for $i \neq j$. The a_i coefficients in (6.4.10) are therefore defined by

$$\begin{aligned}
 a_i &= \frac{\langle g(x), \varphi_i(x) \rangle}{\langle \varphi_i(x), \varphi_i(x) \rangle} = \frac{\int_{-1}^1 f(Y(x)) \varphi_i(x) w(x) dx}{\int_{-1}^1 \varphi_i(x)^2 w(x) dx} \\
 &= \frac{\int_a^b f(y) \varphi_i(X(y)) w(X(y)) dy}{\int_a^b \varphi_i(X(y))^2 w(X(y)) dy}.
 \end{aligned} \tag{6.4.11}$$

In table 6.3 we include a line for the general case of Chebyshev polynomials on the general interval $[a, b]$. We could do the same for Legendre polynomials. Similar linear transformations of Laguerre and Hermite polynomials allow us to create orthogonal polynomial families with weighting function $e^{-(ax+b)}$ on the interval $[d, \infty)$ and families with weighting function $e^{-(ax^2+bx+c)}$ on $(-\infty, \infty)$.

6.5 Uniform Approximation

The previous sections showed how to approximate functions in L^2 norms. Unfortunately, convergence in L^2 puts no restriction on the approximations at any individual point. We often instead want to approximate f uniformly well with polynomials; that is, we want to find a sequence, $p_n(x)$, such that

$$\lim_{n \rightarrow \infty} \max_{x \in [a, b]} |f(x) - p_n(x)| = 0. \tag{6.5.1}$$

Uniform approximations are good approximations of f at each x , whereas least squares approximations attempt to have small total error. Uniform approximation is more demanding than least squares approximation in that it attempts to be a good pointwise approximation. The Weierstrass theorem tells us that there exists such a sequence of approximations for continuous functions.

THEOREM 6.5.1 (Weierstrass theorem) If $f \in C[a, b]$, then for all $\varepsilon > 0$, there exists a polynomial $p(x)$ such that

$$\forall x \in [a, b], |f(x) - p(x)| \leq \varepsilon. \tag{6.5.2}$$

Furthermore, if $f \in C^k[a, b]$ then there exists a sequence of polynomials, p_n , where the degree of p_n is n , such that

$$\lim_{n \rightarrow \infty} \max_{x \in [a, b]} |f^{(l)}(x) - p_n^{(l)}(x)| = 0 \tag{6.5.3}$$

for $l \leq k$.

One proof (see Powell 1981, ch. 6) constructs the *Bernstein polynomials* on $[0, 1]$,

$$p_n(x) \equiv \sum_{k=0}^n \binom{n}{k} f\left(\frac{k}{n}\right) x^k (1-x)^{n-k} \quad (6.5.4)$$

and, by noting the properties of the binomial distribution and the law of large numbers, shows that $\lim_{n \rightarrow \infty} p_n(x) = f(x)$ for all $x \in [0, 1]$. Furthermore the Bernstein polynomials also satisfy the stronger convergence conditions for the derivatives of f expressed in (6.5.3).

The Weierstrass theorem is conceptually valuable but not important from a practical point of view. For example, Bernstein polynomial approximations converge slowly. This is not surprising, since the derivatives of the Bernstein polynomial approximations also converge uniformly to the derivatives of f . We next turn to more efficient methods of generating uniformly convergent approximations of f .

Minimax Approximation

In *minimax approximation* we attempt to find that polynomial that best approximates a function uniformly. More precisely, we define $\rho_n(f)$ to be the infimum of the L^∞ errors of all degree n polynomial approximations of f :

$$\rho_n(f) \equiv \inf_{\deg(q) \leq n} \|f - q\|_\infty. \quad (6.5.5)$$

Theorem 6.5.2 tells us that there is a polynomial that achieves this lower bound.

THEOREM 6.5.2 (Equioscillation theorem) If $f \in C[a, b]$, then there is a unique polynomial of degree n , $q_n^*(x)$, such that $\|f - q_n^*\|_\infty = \rho_n(f)$. The polynomial q_n^* is also the unique polynomial for which there are at least $n + 2$ points $a \leq x_0 < x_1 < \dots < x_{n+1} \leq b$ such that for $m = 1$ or $m = -1$,

$$f(x_j) - q_n^*(x_j) = m(-1)^j \rho_n(f), \quad j = 0, \dots, n + 1. \quad (6.5.6)$$

The property (6.5.6) is called the *equioscillation property*. Geometrically it says that the maximum error of a cubic approximation, for example, should be achieved at least five times and that the sign of the error should alternate between these points. The equioscillation principle is illustrated in figure 6.5 where we plot the error of the minimax cubic polynomial approximation of $(x + 0.1)^{0.1}$ on $[0, 1]$. This picture is important to remember because it shows us the desired shape of the error when we want L^∞ approximation. If we plot the error and have a very different shape, we know that it is theoretically possible to do better. Knowing what the best possible error looks like helps us to know when to stop looking for something better.

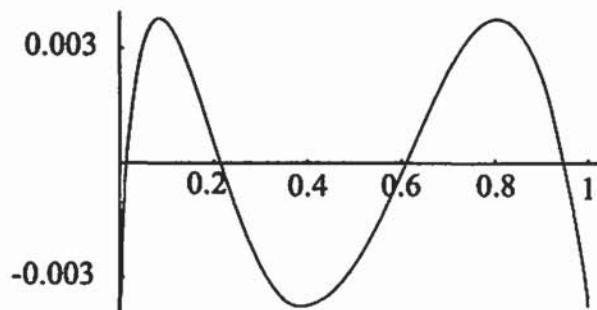


Figure 6.5
Equioscillation example

When using an L^∞ approximation of a function f , one should always keep in mind that it is only the values of f that are being approximated, not any of its derivatives. In fact the derivatives may be poorly approximated. This is illustrated in figure 6.5 where the slope of the approximation error, which equals the difference between the slopes of f and its approximation and is clearly greater and more volatile than the size of the approximation error.

The quality of the $q_n^*(x)$ approximation is addressed by Jackson's theorem.

THEOREM 6.5.3 (Jackson's theorem) For all k , if $f \in C^k[a, b]$, then for $n \geq k$,

$$\rho_n(f) \leq \frac{(n-k)!}{n!} \left(\frac{\pi}{2}\right)^k \left(\frac{b-a}{2}\right)^k \|f^{(k)}\|_\infty. \quad (6.5.7)$$

Proof This is implied in Powell (1981, p. 197, eq. 16.50). ■

Jackson's theorem says two important things. First, the quality of $q_n^*(x)$ improves polynomially as we increase n ; that is, use higher degree polynomials. Second, the polynomial rate of convergence is faster for smoother functions. Unfortunately, computing $q_n^*(x)$ can be difficult. Instead of developing these algorithms, we discuss a least squares approximation procedure that also does very well in creating uniform approximations.

Near-Minimax Approximation

Because minimax approximation is difficult, we commonly use Chebyshev least squares approximation to compute a good substitute. The special property of Chebyshev least squares approximation is that it is nearly the same as the minimax polynomial approximation.

THEOREM 6.5.4 If $C_n(x)$ is the n th degree Chebyshev least squares approximation to $f \in C^1[-1, 1]$, then

$$\rho_n(f) \leq \|f - C_n\|_\infty \leq \left(4 + \frac{4}{\pi^2} \ln n\right) \rho_n(f) \quad (6.5.8)$$

Proof See Rivlin (1990, p. 167, thm. 3.3). ■

By theorem 6.5.4, we conclude that least squares Chebyshev approximations are not much worse than the best polynomial approximation in the L^∞ norm. This is convenient, since constructing L^2 approximations involves only a few linear operations on the data. The fact that this can be done with Chebyshev polynomials is due to the Chebyshev weighting function, $w(x) = (1 - x^2)^{-1/2}$.

Other methods will also yield a good sequence of approximations but will tend to have nonuniform errors. For example, Legendre approximation implicitly uses a constant weighting function, and tends to have larger errors at the end points. The reasons are intuitive. With a constant loss function, making an approximation good at $x = 0$ also makes it good at nearby points to the right and left, whereas making an approximation good at $x = 1$ makes it good only at points to its left. Minimizing the sum of squared errors will therefore concentrate at making the approximation good for central values by sacrificing accuracy at the end points. The Chebyshev weighting function counters this tendency by penalizing errors at the end points more than it penalizes errors in the center.

The $\ln n$ factor is a troublesome term in (6.5.8). Fortunately $\rho_n(f)$ goes to zero fast by Jackson's theorem, yielding an important convergence theorem.

THEOREM 6.5.5 For all k , for all $f \in C^k[a, b]$,

$$\|f - C_n\|_\infty \leq \left(4 + \frac{4}{\pi^2} \ln n\right) \frac{(n-k)!}{n!} \left(\frac{\pi}{2}\right)^k \left(\frac{b-a}{2}\right)^k \|f^{(k)}\|_\infty, \quad (6.5.9)$$

and hence $\lim_{n \rightarrow \infty} \|f - C_n\|_\infty = 0$ uniformly.

The proof of theorem 6.5.5 follows from combining Jackson's theorem with the estimate of $\rho_n(f)$ above.

We illustrate these properties with some simple examples. Table 6.4 contains the Chebyshev coefficients of approximations to e^{x+1} , $(x+1)^{1/4}$, $\max[0, x^3]$, and $\min[\max[-1, 4(x-0.2)], 1]$ on $[-1, 1]$. Also next to the n th coefficient is the L^∞ error of the degree n Chebyshev approximation. Note that the patterns are consistent with the theory. For e^{x+1} the coefficients drop very rapidly, and low-order approximations

Table 6.4
Chebyshev coefficients

$f(x)$:	e^{x+1}	$(x + 1)^{1/4}$	$\max[0, x^3]$	$\min[\max[-1, 4(x - 0.2)], 1]$
c_0	6.88	1.81	0.424	-0.259
c_1	3.07	0.363	0.375	1.23
c_2	0.738	-0.121	0.255	0.241
c_3	0.121	0.065	0.125	-0.314
c_4	1.49(-2)	-0.042	0.036	-0.192
c_5	1.48(-3)	0.030	0	0.100
c_6	1.22(-4)	-0.023	-0.004	0.129
c_7	8.69(-6)	0.018	0	-0.015
c_8	5.42(-7)	-0.015	0.001	-0.068
c_9	3.00(-8)	0.012	0	-0.010
c_{10}	1.50(-9)	0.011	-4.2(-4)	0.025

do quite well. On the other hand, convergence for $\max[0, x^3]$ is much slower. The function $(x + 1)^{1/4}$ is an intermediate case. It is $C^\infty(0, 1]$, but all derivatives at $x = -1$ are infinite, making our error estimates useless.

Chebyshev Economization

Suppose that $q(x)$ is an m th degree polynomial. Then q can be written

$$q(x) = \sum_{k=0}^m a_k T_k(x). \quad \text{This is in fact the Chebyshev expansion of } f$$

Since it is a polynomial, computing the Chebyshev least squares approximation is easy. In fact, once we have the Chebyshev polynomial expansion of q , the n th degree least squares approximation for $n < m$ can be written

$$p_n(x) = \sum_{k=0}^n a_k T_k(x).$$

This is called the n th degree *Chebyshev economization* of q . Furthermore $p_n(x)$ is the minmax degree n polynomial approximation of q . In this way we can approximate a high-degree polynomial very well with a lower-degree polynomial.

While these results are of interest, they do not yet yield a reliable and simple approximation method since computing the Chebyshev least squares approximation is costly, involving the evaluation of several integrals. Since almost all approximations are ultimately based on only a finite number of evaluations of $f(x)$, we next

consider interpolation as an approximation scheme. The past two sections have described basic approximation theory. With these ideas in mind, we will now move to actual methods constructing approximations.

6.6 Interpolation

Interpolation is any method that takes a finite set of conditions and constructs a “nice” function that satisfies these conditions. In this section we will interpolate with polynomials, our collection of “nice functions,” but it will be clear that many of the ideas will also work with other families of “nice” functions.

Lagrange Interpolation

The most demanding but feasible version of one-dimensional polynomial interpolation is to take a collection of n points in R^2 , $D = \{(x_i, y_i) | i = 1, \dots, n\}$, where the x_i are distinct, and find a degree $n - 1$ polynomial, $p(x)$, such that $y_i = p(x_i)$, $i = 1, \dots, n$. The collection D is called the *Lagrange data*, and this is called the *Lagrange interpolation problem*.

We can explicitly construct the interpolating polynomial. Define

$$l_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}. \quad (6.6.1)$$

Note that $l_i(x)$ is unity at $x = x_i$ and zero at $x = x_j$ for $j \neq i$. This property implies that the *Lagrange interpolation polynomial*

$$p(x) = \sum_{i=1}^n y_i l_i(x) \quad (6.6.2)$$

interpolates the data, that is, $y_i = p(x_i)$, $i = 1, \dots, n$.

The Lagrange interpolation formula is not practical in general. The reason for computing the interpolant p is to evaluate p at $x \neq x_i$, but to evaluate (6.6.2) at an arbitrary x would require a large number of operations. The Lagrange formula uses (roughly) $3n$ additions, $2n$ divisions, and $2n$ multiplications even when done most efficiently. Recall that Horner’s method shows that evaluating a degree $n - 1$ polynomial needs only $n - 2$ additions and $n - 1$ multiplications. To make the Lagrange formula useful, we need to compute the coefficients of the interpolating polynomial, that is, to find the a_i such that $p(x) = \sum_{i=0}^{n-1} a_i x^i$. This is easily accomplished using symbolic math software.

A theoretically useful way to “compute” the coefficients of the interpolating polynomial is to solve the linear system

$$a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_{n-1} x_i^{n-1} = y_i, \quad i = 1, \dots, n. \quad (6.6.3)$$

If we define the *Vandermonde matrix* for x_i , $i = 1, \dots, n$, to be

$$V = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix}, \quad (6.6.4)$$

then the coefficient vector $a = (a_0, a_1, \dots, a_{n-1})^\top$ of $p(x)$ solves $Va = y$.

THEOREM 6.6.1 If the points x_1, \dots, x_n are distinct, then there is a unique solution to the Lagrange interpolation problem.

Proof The Lagrange formula implies existence. To show uniqueness, suppose that $\hat{p}(x)$ is another polynomial of degree at most $n - 1$ that interpolates the n data points. Then $p(x) - \hat{p}(x)$ is a polynomial of at most degree $n - 1$ and is zero at the n interpolation nodes. Since the only degree $n - 1$ polynomial with n distinct zeros is the zero polynomial, $p(x) - \hat{p}(x) = 0$ and $p(x) = \hat{p}(x)$. Note that this shows V is non-singular iff the x_i are distinct. ■

Using (6.6.3) to directly compute the coefficients of the interpolating polynomial is not practical for the reason that the condition number of V is generally large. In any case, solving the linear equation $Va = y$ has a time cost of $O(n^3)$. The simplest way to compute the a_i coefficients is to compute the coefficients of the Lagrange polynomial in (6.6.2), and then combine the coefficients of like powers of x .

Hermite Interpolation

We may want to find a polynomial p that fits slope as well as level requirements. Suppose that we have data

$$p(x_i) = y_i, \quad p'(x_i) = y'_i, \quad i = 1, \dots, n, \quad (6.6.5)$$

where the x_i are distinct. Since we have $2n$ conditions, we are looking for a degree $2n - 1$ polynomial that satisfies these conditions; this is the *Hermite interpolation problem*.

We construct the unique solution, $p(x)$. First define the functions

$$\begin{aligned}\tilde{h}_i(x) &= (x - x_i) l_i(x)^2, \\ h_i(x) &= (1 - 2l'_i(x_i)(x - x_i)) l_i(x)^2.\end{aligned}\tag{6.6.6}$$

The critical properties of h_i and \tilde{h}_i are

$$\begin{aligned}h'_i(x_j) = \tilde{h}'_i(x_j) &= 0, \quad 1 \leq i, j \leq n, \\ h_i(x_j) = \tilde{h}_i(x_j) &= \begin{cases} 0, & i \neq j, \\ 1, & i = j. \end{cases}\end{aligned}\tag{6.6.7}$$

Hence h_i is a function that is zero at all interpolation nodes except x_i , where it is unity, and its derivative is zero at all $x_j, j = 1, \dots, n$; the reverse is true for $\tilde{h}_i(x)$. The properties (6.6.7) allow us to simply construct the *Hermite interpolating polynomial* as a weighted sum of the h_i and \tilde{h}_i functions with the weights being the prescribed values for $p(x)$ and $p'(x)$ at the interpolation nodes:

$$p(x) = \sum_{i=1}^n y_i h_i(x) + \sum_{i=1}^n y'_i \tilde{h}_i(x)\tag{6.6.8}$$

As with Lagrange interpolation, the formulas generated by Hermite interpolation are inefficient. If one were to use these formulas, one would simplify (6.6.8) by combining like powers.

Cardinal Function Interpolation

Lagrange and Hermite interpolation are special cases of a general approach to interpolation. Suppose that we have a set of interpolation nodes, x_i , and linearly independent functions $\phi_i(x)$, $i = 1, \dots, n$, that satisfy

$$\phi_i(x_j) = \delta_i^j \equiv \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}\tag{6.6.9}$$

Such a collection is called a *cardinal function basis*. With such a basis, we can directly write an interpolating function with Lagrange data as

$$p(x) = \sum_i y_i \phi_i(x).\tag{6.6.10}$$

In the case of Lagrange interpolation, the functions $\phi_i(x) = \prod_{j \neq i} (x - x_j)/(x_i - x_j)$

comprise the cardinal basis. The advantages of cardinal bases is that the data are the coefficients. The disadvantages are that the basis functions may be costly to compute and that any evaluation of $p(x)$ requires the evaluation of each of the $\phi_i(x)$.

6.7 Approximation through Interpolation and Regression

We often have some qualitative information about a function, such as smoothness properties, and can evaluate the function at many points, but at a nontrivial cost per point. In such a case, computing the least squares approximation (6.4.2) is not possible because the critical integrals cannot be evaluated exactly. In this section we turn to methods that use a finite number of points to compute an approximation to $f(x)$.

We could use Lagrange interpolation. If the result is to be a good approximation of f , we also want some assurance that the interpolating function agrees with the function more generally. This is a much more difficult problem because it requires some information about the function globally.

Unfortunately, interpolation does not always work even on well-behaved, simple functions. Examine figure 6.6. The solid line is a graph of the function $f(x) = \frac{1}{1+x^2}$ over the interval $[-5, 5]$, whereas the dotted line is the tenth-degree polynomial Lagrange interpolation of f at the 11 uniformly spaced nodes, including the end points. Note that the interpolating polynomial does a very poor job of approximating f . This is not an aberration; in fact the degree $n - 1$ interpolation at n uniformly spaced points, $p_n(x)$, gets worse as we use more points, since, for $|x| > 3.64$, $\limsup_{n \rightarrow \infty} |f(x) - p_n(x)| = \infty$. Therefore, for a seemingly well-behaved C^∞ function, interpolation at the uniformly spaced nodes does not improve as we use more

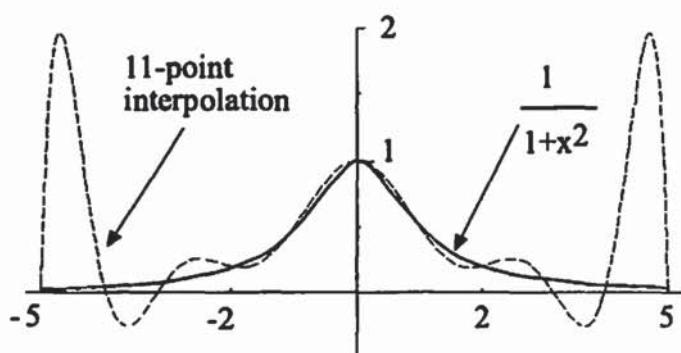


Figure 6.6
Nonconvergence of interpolation example

points.² This example shows that this interpolation scheme may not work; we will next ask whether there are any dependable interpolation schemes.

Interpolation Error

The last example may discourage one from approximating a function through interpolation. While it does indicate that caution is necessary, there are some procedures that reduce the likelihood of perverse behavior by interpolants. Recall that we defined $l_i(x) = \prod_{j \neq i} (x - x_j)/(x_i - x_j)$, and that the Lagrange polynomial interpolating f at points x_i is $p_n(x) = \sum_{i=1}^n f(x_i)l_i(x)$. Define

$$\Psi(x; x_1, \dots, x_n) = \prod_{k=1}^n (x - x_k).$$

The following theorem uses smoothness conditions to compute a bound on the error of the Lagrange interpolation polynomial:

THEOREM 6.7.1 Assume that $a = x_1 < \dots < x_n = b$. Then for all $x \in [a, b]$ there is some $\xi \in [a, b]$ such that

$$f(x) - \sum_{j=1}^n f(x_j) l_j(x) = \Psi(x; x_1, \dots, x_n) \frac{f^{(n)}(\xi)}{n!}.$$

Proof See Atkinson (1989, p. 134, thm. 3.2). ■

Theorem 6.7.1 implies that if $p_n(x) \equiv \sum_{j=1}^n f(x_j) l_j(x)$, then

$$\sup_{x \in [a, b]} |f(x) - p_n(x)| \leq \frac{\|f^{(n)}\|_\infty}{n!} \sup_{x \in [a, b]} \Psi(x; x_1, \dots, x_n) \quad (6.7.1)$$

is an upper bound on the error of the interpolant, $p_n(x)$, through x_1, \dots, x_n . We want $p_n(x)$ to be an increasingly good approximation for $f(x)$ as $n \rightarrow \infty$. Suppose that we interpolate at uniformly distributed points. The example in figure 6.6 showed that convergence may fail with a uniform grid, implying that (6.7.1) does not shrink to zero. Using Hermite interpolation does not improve matters. If $p(x)$ is a Hermite interpolant given by (6.6.8), the Hermite interpolation error is bounded above by

$$|f(x) - p(x)| \leq (\Psi(x; x_1, \dots, x_n))^2 \frac{|f^{(2n)}(\xi)|}{(2n)!} \quad (6.7.2)$$

2. This failure may not be so surprising if we take into account some elementary complex analysis: $(1 + x^2)^{-1}$ has poles at $\pm i$, implying that it has no power series representation on $[-5, 5]$.

for some ξ between x and x_1 . Since Ψ again is the critical piece, this error bound will not shrink to zero if we use uniform grids, even for $f \in C^\infty$.

Chebyshev Minimax Property

The challenge of approximation through interpolation is to choose the interpolation nodes, x_i , so as to be assured of a small interpolation error. The key result is the minimax property of Chebyshev polynomials.

THEOREM 6.7.2 For $n > 0$, consider

$$s_n = \inf_p \left[\max_{-1 \leq x \leq 1} |x^n + p(x)| \right]$$

$$\text{s.t. } \deg(p) \leq n - 1.$$

Then $s_n = 2^{1-n}$ and is attained when $x^n + p(x) = (1/2)^{n-1} T_n(x)$.

Proof See Powell (1981, p. 78, thm. 7.3). ■

Hence $(1/2)^{n-1} T_n(x)$ is the polynomial with a leading term equal to x^n which has the smallest maximum magnitude.

Chebyshev Interpolation

We will next combine the interpolation error formula with the Chebyshev minimax property to determine a good collection of interpolation nodes. Equation (6.7.1) demonstrates that this error is no more than

$$\left(\max_{x \in [a, b]} \Psi(x; x_1, \dots, x_n) \right) \|f^{(n)}\|_\infty (n!)^{-1}.$$

Note the three components. The term $n!$ is a function of only n , and $\|f^{(n)}\|_\infty$ depends only on f . Both of these are independent of the interpolation points. Our choice of $\{x_i\}_{i=1}^n$ affects only the maximum value of $\Psi(x)$, which in turn does not depend on f . So if we want to choose interpolation points so as to minimize their contribution to (6.7.1), the problem is

$$\min_{x_1, \dots, x_n} \max_{x \in [a, b]} \prod_{k=1}^n (x - x_k). \quad (6.7.3)$$

Note that the zeroes of $\Psi(x; x_1, \dots, x_n)$ are the x_k and that the leading term is x^n . Hence the minimax property of Chebyshev polynomials tells us that the choice of $\Psi(x)$ must be $T_n(x)/2^{n-1}$. Furthermore we conclude that the x_k , the zeros of

$\Psi(x; x_1, \dots, x_n)$, must be the zeros of T_n . Hence

$$x_k = \cos\left(\frac{2k-1}{2n}\pi\right), \quad k = 1, \dots, n, \quad (6.7.4)$$

are the interpolation nodes that minimize the error bound computed above.

The next question is whether interpolation at the zeros of Chebyshev polynomials leads to asymptotically valid approximations of f .

THEOREM 6.7.3 Suppose that function $f: [-1, 1] \rightarrow R$ is C^k for some $k \geq 1$, and let I_n be the degree n polynomial interpolation of f based at the zeros of $T_n(x)$. Then

$$\|f - I_n\|_{\infty} \leq \left(\frac{2}{\pi} \log(n+1) + 1\right) \frac{(n-k)!}{n!} \left(\frac{\pi}{2}\right)^k \left(\frac{b-a}{2}\right)^k \|f^{(k)}\|_{\infty} \quad (6.7.5)$$

Proof See Rivlin (1990, p. 14).

Theorem 6.7.3 shows that Chebyshev interpolation will work for C^1 smooth functions. This result is not quite as good as the Fourier theory. For periodic functions Fourier series are used, and one gets pointwise convergence almost everywhere for any piecewise C^0 function.

It may appear unnatural not to include the end points in an interpolation scheme for $[-1, 1]$. In fact for x near -1 and 1 the Chebyshev approximation is really an extrapolation of the data, not an interpolation. For such points the Chebyshev interpolant often does poorly. If one wants to include -1 and 1 as interpolating nodes, then let

$$x_i = \sec\left(\frac{\pi}{2n}\right) \cos\left(\frac{2i-1}{2}\frac{\pi}{n}\right), \quad i = 1, \dots, n, \quad (6.7.6)$$

be the set of n points used. This is called the *expanded Chebyshev array* and is practically as good as the Chebyshev array (see Rivlin 1990, p. 23).

Approximation and Regression

When the number of data points substantially exceeds the number of unknown coefficients in a representation, then least squares methods can be used to find an approximation. These are essentially regression methods. In our approximation problems, regression lies between the pure least squares approximation method, as solved in (6.4.2), and interpolation. Least squares approximation, (6.4.2), implicitly uses all the value of $f(x)$ at all x , whereas interpolation methods use only n points to find n parameters. In regression methods we use m points to find $n < m$ parameters

in some approximation. We will generally underfit the data, resulting in some approximation error. However, using more points than free parameters makes the approximation method better behaved. Furthermore the convergence results in theorem 6.7.3 hold also for regression.

We can adapt Chebyshev least squares approximation and Chebyshev interpolation ideas to come up with a general *Chebyshev regression* algorithm. We compute the degree $m - 1$ interpolation formula but use only the degree n truncation, essentially dropping the degree $n + 1$ through $m - 1$ terms. The omitted terms are high-degree polynomials that may produce undesirable oscillations; this is an application of Chebyshev economization. The result is a smoother function that approximates the data. The Algorithm 6.2 summarizes the procedure for constructing a degree n polynomial that approximately computes the function f for $x \in [a, b]$ using $m > n$ points. If $m = n + 1$, the algorithm reduces to Chebyshev interpolation.

Algorithm 6.2 Chebyshev Regression Algorithm

Objective: Choose m nodes and use them to construct a degree $n < m$ polynomial approximation of $f(x)$ on $[a, b]$.

Step 1. Compute the $m \geq n + 1$ Chebyshev interpolation nodes on $[-1, 1]$:

$$z_k = -\cos\left(\frac{2k-1}{2m}\pi\right), \quad k = 1, \dots, m.$$

Step 2. Adjust the nodes to the $[a, b]$ interval:

$$x_k = (z_k + 1)\left(\frac{b-a}{2}\right) + a, \quad k = 1, \dots, m.$$

Step 3. Evaluate f at the approximation nodes:

$$y_k = f(x_k), \quad k = 1, \dots, m.$$

Step 4. Compute Chebyshev coefficients, $a_i, i = 0, \dots, n$:

$$a_i = \frac{\sum_{k=1}^m y_k T_i(z_k)}{\sum_{k=1}^m T_i(z_k)^2}$$

to arrive at the approximation for $f(x), x \in [a, b]$:

$$\hat{f}(x) = \sum_{i=0}^n a_i T_i\left(2\frac{x-a}{b-a} - 1\right).$$

zk

6.8 Piecewise Polynomial Interpolation

Lagrange interpolation computes a C^∞ function to interpolate the given data. An alternative is to construct a function that is only piecewise smooth. This section presents two simple methods.

Piecewise Linear Interpolation

The simplest way to interpolate is the kindergarten procedure of “connecting the dots,” that is, draw straight lines between successive data points of the Lagrange data. If we have Lagrange data (x_i, y_i) , then the *piecewise linear interpolant* $\hat{f}(x)$ is given by the formula

$$\hat{f}(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i} (y_{i+1} - y_i), \quad x \in [x_i, x_{i+1}]. \quad (6.8.1)$$

Piecewise linear interpolants are spanned by B-splines, which we discuss below.

Hermite Interpolation Polynomials

Suppose that we want to interpolate both level and slope information at x_1, \dots, x_n , but we only use piecewise cubic polynomials. *Piecewise Hermite polynomial interpolation* will do this. First, for an arbitrary interval $[a, b]$, define the four cubic polynomials

$$\begin{aligned} \varphi_1(t) &= 1 - 3t^2 + 2t^3, & \varphi_2(t) &= (b-a)t(t-1)^2, \\ \varphi_3(t) &= t^2(3-2t), & \varphi_4(t) &= (b-a)t^2(t-1), \end{aligned} \quad (6.8.2)$$

where $t \equiv (x - a)/(b - a)$. The polynomial

$$p(x) = y_1\varphi_1(x) + y_2\varphi_3(x) + y'_1\varphi_2(x) + y'_2\varphi_4(x) \quad (6.8.3)$$

interpolates the data

$$y_1 = p(a), \quad y_2 = p(b), \quad y'_1 = p'(a), \quad y'_2 = p'(b).$$

Next, suppose that we have this level and slope information at several points. Within each interval we construct the polynomial in (6.8.3). The collection of interval-specific interpolations constitute our piecewise polynomial approximation. The resulting function is a cubic polynomial almost everywhere and is C^1 everywhere.

6.9 Splines

A particularly powerful and widely used piecewise polynomial interpolation scheme uses *splines*. A spline is any smooth function that is piecewise polynomial but also smooth where the polynomial pieces connect. Formally a function $s(x)$ on $[a, b]$ is a *spline of order n* if s is C^{n-2} on $[a, b]$, and there is a grid of points (called nodes) $a = x_0 < x_1 < \dots < x_m = b$ such that $s(x)$ is a polynomial of degree $n - 1$ on each subinterval $[x_i, x_{i+1}]$, $i = 0, \dots, m - 1$. Note that an order 2 spline is just the common piecewise linear interpolant in equation (6.8.1).

Cubic Splines

Since the *cubic spline* (a spline of order 4) is the most popular, we will describe in detail one way to construct a cubic spline interpolant; the procedure can be applied analogously to compute a spline of any order. Suppose that the Lagrange data set is $\{(x_i, y_i) | i = 0, \dots, n\}$. The x_i will be the nodes of the spline, and we want to construct a spline, $s(x)$, such that $s(x_i) = y_i$, $i = 0, \dots, n$. On each interval $[x_{i-1}, x_i]$, $s(x)$ will be a cubic $a_i + b_i x + c_i x^2 + d_i x^3$. A cubic spline is represented in a computer as a list of the a_i, b_i, c_i , and d_i coefficients along with a list of the x_i nodes. We therefore have n intervals, $n + 1$ data points, and $4n$ unknown coefficients, $a_i, b_i, c_i, d_i, i = 1, \dots, n$.

The interpolating conditions plus continuity at the interior nodes implies $2n$ conditions on the coefficients:

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3, \quad i = 1, \dots, n, \quad (6.9.1)$$

$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3, \quad i = 0, \dots, n - 1. \quad (6.9.2)$$

Since the approximation is to be C^2 at the interior nodes, the first and second derivatives must agree at the nodes, implying $2n - 2$ more conditions:

$$b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2, \quad i = 1, \dots, n - 1, \quad (6.9.3)$$

$$2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i, \quad i = 1, 2, \dots, n - 1. \quad (6.9.4)$$

Note that (6.9.1)–(6.9.4) are linear equations in the unknown parameters, a , b , c , and d .

This exhausts the interpolation and smoothness conditions, leaving us two conditions short of fixing the unknown coefficients. Various splines are differentiated by the two additional conditions imposed. One way to fix the spline is to pin down $s'(x_0)$ and $s'(x_n)$. For example, the *natural spline* imposes $s'(x_0) = 0 = s'(x_n)$. A natural

spline is of special interest because it minimizes the total curvature, defined to be $\int_{x_0}^{x_n} s''(x)^2 dx$, of all cubic splines that interpolate the data.

Another way to fix $s(x)$ is to make the interpolation good in some other sense. For example, we may know, or have a good guess, about the slope at x_0 and x_n . For example, we may reject the natural spline if we want a concave $s(x)$. If we know the true slopes for the function being approximated are y'_0 and y'_n at the end points, we can then impose $s'(x_0) = y'_0$ and $s'(x_n) = y'_n$; the resulting spline is called the *Hermite spline*. If we want to use a slope approach but do not have the true values, we can use the slopes of the secant lines over $[x_0, x_1]$ and $[x_{n-1}, x_n]$; that is, we choose $s(x)$ to make $s'(x_0) = (s(x_1) - s(x_0))/(x_1 - x_0)$ and $s'(x_n) = (s(x_n) - s(x_{n-1}))/(x_n - x_{n-1})$. We call this the *secant Hermite spline*. With natural, secant, and Hermite splines, the two extra conditions are linear. To solve for the coefficients, we just add two slope conditions to the system (6.9.1)–(6.9.4), and solve the combined system of linear equations.

Theorem 6.9.1 displays some useful error formulas that indicate the quality of spline approximation.

THEOREM 6.9.1 If $f \in C^4[x_0, x_n]$ and s is the Hermite cubic spline approximation to f on $\{x_0, x_1, \dots, x_n\}$ and $h \geq \max_i\{x_i - x_{i-1}\}$, then

$$\|f - s\|_\infty \leq \frac{5}{384} \|f^{(4)}\|_\infty h^4$$

and

$$\|f' - s'\|_\infty \leq \left[\frac{\sqrt{3}}{216} + \frac{1}{24} \right] \|f^{(4)}\|_\infty h^3.$$

Proof See Prenter (1989, p. 112). ■

These error bounds indicate the rapid rate of convergence of spline interpolation. In general, order $k+2$ splines will yield $\mathcal{O}(n^{-(k+1)})$ convergence for $f \in C^{k+1}[a, b]$.

Splines are excellent for approximations for two general reasons. First, evaluation is cheap, since splines are locally cubic. To evaluate a spline at x one must first find which interval $[x_i, x_{i+1}]$ contains x , look at some table to find the coefficients for the particular polynomial used over $[x_i, x_{i+1}]$, and evaluate that polynomial at x . The only possible difficulty is determining which interval $[x_i, x_{i+1}]$ contains x . If there are n intervals, then it will take at most $\lceil \log_2 n \rceil$ comparisons to determine which one contains x . Even that cost can be avoided if the nodes are carefully chosen. For example, if the nodes are evenly spaced, that is, $x_i = a + ih$ for some h , then find-

ing the relevant interval for x uses the fact that $x \in [x_i, x_{i+1}]$ if and only if $i = \lfloor (x - a)/h \rfloor$. In general, if the nodes are created according to some invertible formula, the inverse can be used to quickly determine the interval to which x belongs.

The second reason for using splines is that good fits are possible even for functions that are not C^∞ or have regions of large higher-order derivatives. This is indicated for cubic splines by the error formulas above, which depend only on $f^{(4)}(x)$ implying that badly behaved $f^{(5)}(x)$ will not adversely affect the performance of a cubic spline.

B-Splines

The discussion above assumes that a spline is represented by the collection of nodes, x_i , and of interval specific coefficients, a_i, b_i, c_i and d_i , $i = 0, \dots, n$. The space of splines with nodes on a prescribed grid comprise a finite vector space. The B -splines form a basis for splines.

Suppose that we have a grid of knots at $x_{-k} < \dots < x_{-1} < x_0 < \dots < x_{n+k}$. Order 1 splines implement step function interpolation and are spanned by the B^0 -splines. The typical B^0 -spline is

$$B_i^0(x) = \begin{cases} 0, & x < x_i, \\ 1, & x_i \leq x < x_{i+1}, \\ 0, & x_{i+1} \leq x, \end{cases} \quad (6.9.5)$$

for $i = -k, \dots, n$. Note that the B_i^0 are right-continuous step functions. Linear splines implement piecewise linear interpolation and are spanned by the B^1 -splines, with a typical B^1 -spline being

$$B_i^1(x) = \begin{cases} 0, & x \leq x_i \text{ or } x \geq x_{i+2}, \\ \frac{x - x_i}{x_{i+1} - x_i}, & x_i \leq x \leq x_{i+1}, \\ \frac{x_{i+2} - x}{x_{i+2} - x_{i+1}}, & x_{i+1} \leq x \leq x_{i+2}. \end{cases} \quad (6.9.6)$$

The B_i^1 -spline is the tent function with peak at x_{i+1} and is zero for $x \leq x_i$ and $x \geq x_{i+2}$. Both B^0 - and B^1 -splines form cardinal bases for interpolation at the x_i 's.

Higher-order B -splines are defined by the recursive relation

$$B_i^k(x) = \left(\frac{x - x_i}{x_{i+k} - x_i} \right) B_i^{k-1}(x) + \left(\frac{x_{i+k+1} - x}{x_{i+k+1} - x_{i+1}} \right) B_{i+1}^{k-1}(x). \quad (6.9.7)$$

These B -spline families can be used to construct arbitrary splines.

THEOREM 6.9.2 Let S_n^k be the space of all order $k + 1$ spline functions on $[x_0, x_n]$ with knots at $\{x_0, x_1, \dots, x_n\}$. Then 1–4 follow:

1. The set

$$\{B_i^k|_{[x_0, x_n]} : -k \leq i \leq n - 1\}$$

forms a linearly independent basis for S_n^k , which has dimension $n + k$.

2. $B_i^k(x) \geq 0$, and the support of $B_i^k(x)$ is (x_i, x_{i+k+1}) .
3. $(d/dx)B_i^k(x) = (k/x_{i+k} - x_i)B_i^{k-1}(x) - (k/x_{i+k+1} - x_{i+1})B_{i+1}^{k-1}(x)$.
4. If we have Lagrange interpolation data, $(z_i, y_i), i = 1, \dots, n + k$, and

$$\max[x_0, x_{i-k-1}] < z_i < \min[x_i, x_n], \quad 1 \leq i \leq n + k,$$

then there is an interpolant S in S_n^k such that $y_i = S(z_i), i = 1, \dots, n + k$.

These properties form the basis for computations using B -splines. The general theory of B -splines is laid out in de Boor (1978), and the necessary programs are also presented there.

6.10 Examples

We next illustrate various approximation methods for some simple functions. We examine each of these functions on the interval $[-1, 1]$. To make the comparisons “fair,” we will use methods which have five free parameters. Therefore we explore fourth-degree least squares approximations and interpolations at five different points. The spline approximations are all cubic splines where we use the secant Hermite method to fix the free end points.

First, we examine e^{2x+2} . This is a C^∞ function; in fact, it is analytic on the complex plane, a very well-behaved function. Figure 6.7 displays the approximation error for four different approximation methods: Legendre least squares, Chebyshev interpolation, spline, and polynomial interpolation on a uniform grid. Over $[-1, 1]$, e^{2x+2} ranges in value from 1 to 54, and its derivative ranges from 1 to 109. Despite this substantial variation, all methods did so well in approximating e^{2x+2} that graphs of e^{2x+2} and its approximations are indistinguishable. Figure 6.7 displays the approximation errors. The Chebyshev interpolation approximation had the smallest maximum error and most nearly satisfied the equioscillation property, as predicted by our theorems.

Next figure 6.8 displays the spline, Chebyshev, and uniform interpolation approximations of $(x + 1)^{1/4}$ on $[-1, 1]$. This function has a singularity at $x = -1$. Note

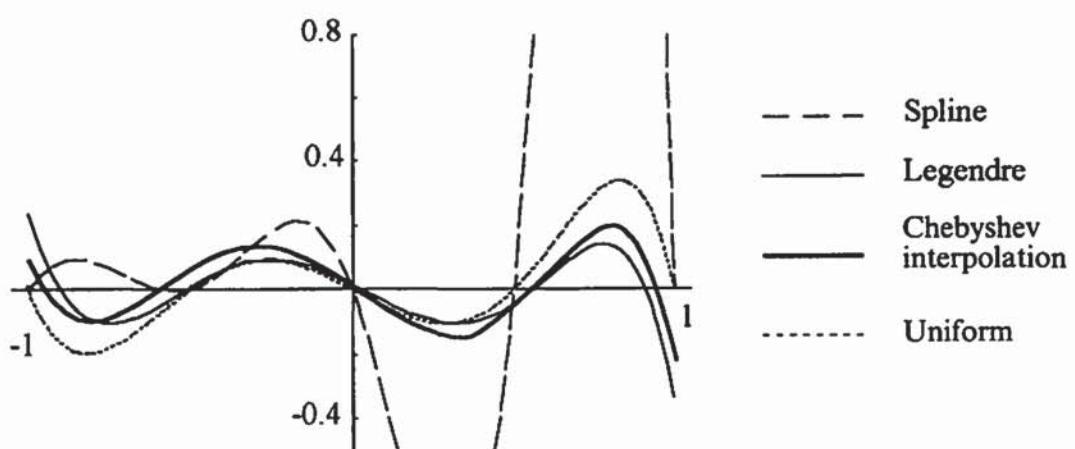


Figure 6.7
Approximation errors for e^{2x+2} approximations

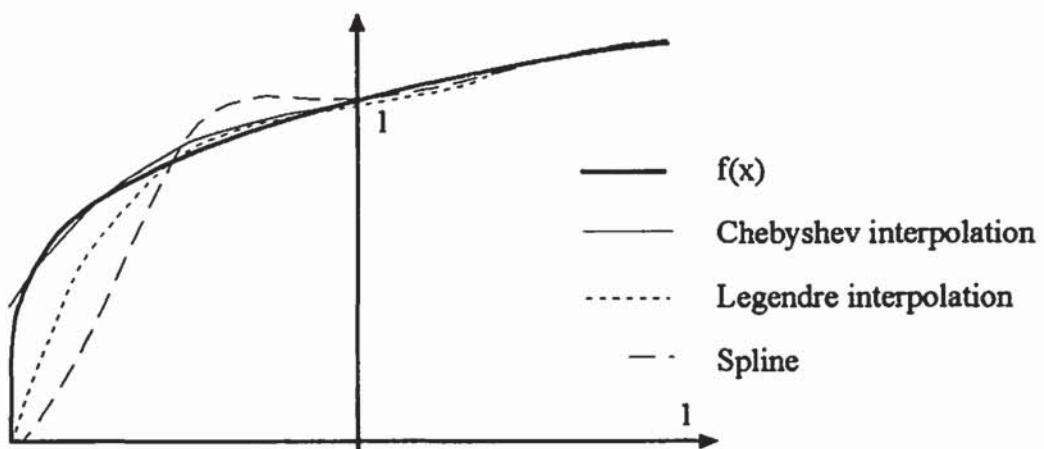


Figure 6.8
Approximations for $(x + 1)^{0.25}$

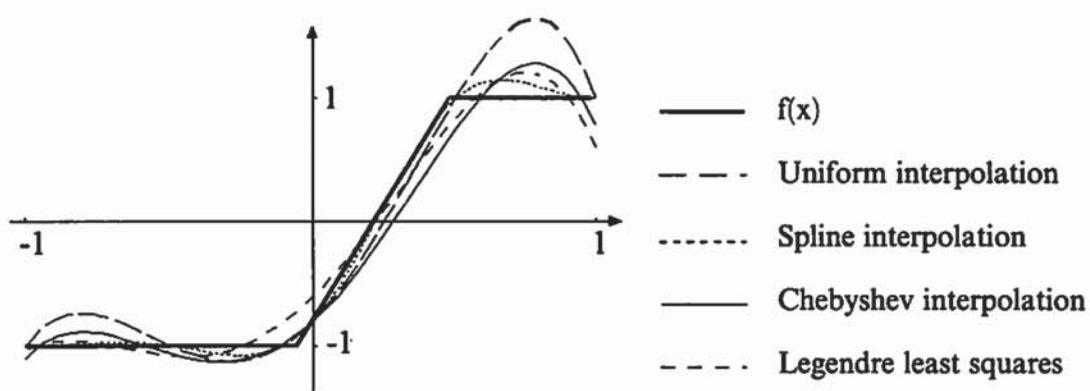


Figure 6.9
Approximations of $\min[\max[-1, 4(x - 0.2)], 1]$

that the approximations do well except near the singular point, $x = -1$. This is expected, since all the derivatives of $(x + 1)^{1/4}$ are infinite at $x = -1$. Since the spline and uniform approximants interpolate $(x + 1)^{1/4}$ at $x = -1$, they do poorly on $[-1, -0.5]$ because $(x + 1)^{1/4}$ initially has infinite slope, a feature that no low-order polynomial can match. In an effort to catch up with the fast-rising $(x + 1)^{1/4}$ the spline and uniform interpolants overshoot it at $x = -0.5$. In contrast, the Chebyshev interpolant ignores the value at $x = -1$, avoiding the high curvature there by instead interpolating at a point to the right of $x = -1$, and it does well almost everywhere else. Even though $(x + 1)^{1/4}$ is C^∞ and locally analytic, the singularity at $x = -1$ makes approximation more difficult. Note how the spline and uniform interpolant approximations have convex regions even though $(x + 1)^{1/4}$ is concave over $[-1, 1]$. This shows how our methods will fail at preserving qualitative properties, such as concavity, even while producing good L^∞ approximations. Also one should be very careful using any of these methods near singularities.

Figure 6.9 displays the results for $f(x) = \min[\max[-1, 4(x - 0.2)], 1]$, a much more difficult case. The problem in approximating this function lies in trying to approximate the two kink points with polynomials.³ The key point is that all of these approximations are bothered by the kinks. Since they can use only low-order polynomials, they have difficulty in “turning the corners,” which are essentially points of infinite curvature. In order to make any turn through a kink, all of the approximations oscillate around $f(x)$ on both sides of the kink. Even the spline does not do well, though the piecewise polynomial structure of splines would seem to make them

3. The first kink was put at $x = -0.05$ so that it would not be an interpolation point.

better for dealing with kinks. The problem here is that cubic splines are C^2 , which is much smoother than the function being approximated, and five free parameters are not enough for the flexibility of splines to be exploited. In this case, though, it is surprising that any of the methods, particularly the interpolation methods, do as well as they do.

6.11 Shape-Preserving Approximation

All of the procedures above find an approximation that is close to f in some norm. Sometimes we may be less concerned with closeness and more interested in other properties. In particular, we may care more about getting a visually pleasing approximation, one that looks right. For example, in figure 6.8, we had L^2 approximations to a concave function that were not themselves concave. Instead, they fluctuated around the function being approximated. In fact the equioscillation property of L^∞ approximation methods shows that methods that are good in the L^∞ sense necessarily yield such oscillations around the true function. The spline approximations can also fail to produce concave approximations in those cases. Not only can these approximations fail to preserve curvature, but it is possible that an increasing function is approximated by a function that is decreasing in some places. We refer to the curvature and monotonicity properties of a function as aspects of its shape. In some cases it is important to construct approximations that preserve shape as well as are accurate. We now turn to some basic methods for preserving shape.

Piecewise-Linear Interpolation

Piecewise-linear interpolation obviously preserves positivity, monotonicity, and concavity. However, it is not differentiable. This violates our desire to use smooth functions to approximate smooth functions.

Shape-Preserving Quadratic Spline Interpolation

We next present the shape-preserving quadratic spline of Schumaker (1983) which produces a smooth function which both interpolates data and preserves some shape. We first examine the Hermite interpolation version and then discuss the Lagrange version.

Let us consider the shape-preservation problem on an interval. The basic Hermite problem on the single interval $[t_1, t_2]$ takes the data z_1, z_2, s_1, s_2 , and constructs a piecewise-quadratic function $s \in C^1[t_1, t_2]$ such that

$$s(t_i) = z_i, \quad s'(t_i) = s_i, \quad i = 1, 2. \tag{6.11.1}$$

We first examine the nongeneric case where a quadratic polynomial works.

LEMMA 6.11.1 If $(s_1 + s_2)/2 = (z_2 - z_1)/(t_2 - t_1)$, then the quadratic function

$$s(t) = z_1 + s_1(t - t_1) + \frac{(s_2 - s_1)(t - t_1)^2}{2(t_2 - t_1)} \quad (6.11.2)$$

satisfies (6.11.1).

The shape-preserving properties of this construction are clear. First, if $s_1 \cdot s_2 \geq 0$, then $s'(t)$ has the same sign as s_1 and s_2 throughout $[t_1, t_2]$. Therefore, if the data indicate a monotone increasing (decreasing) function on $[t_1, t_2]$, then $s(t)$ is similarly monotone. Second, $s_1 < s_2$ indicates a convex function, which $s(t)$ satisfies since $s''(t) = (s_2 - s_1)/(t_2 - t_1)$. Similarly, if $s_1 > s_2$, then $s(t)$ and the data are concave.

In general, lemma 6.11.1 does not apply. The typical strategy is to add a knot to the interval $[a, b]$ and then construct a spline with the desired properties. Lemma 6.11.2 first constructs a quadratic spline that satisfies (6.11.1).

LEMMA 6.11.2 For every $\xi \in (t_1, t_2)$, there is a unique quadratic spline solving (6.11.1) with a knot at ξ . It is

$$s(t) = \begin{cases} A_1 + B_1(t - t_1) + C_1(t - t_1)^2, & t \in [t_1, \xi], \\ A_2 + B_2(t - \xi) + C_2(t - \xi)^2, & t \in [\xi, t_2], \end{cases} \quad (6.11.3)$$

$$A_1 = z_1, \quad B_1 = s_1, \quad C_1 = \frac{\bar{s} - s_1}{2\alpha},$$

$$A_2 = A_1 + \alpha B_1 + \alpha^2 C_1, \quad B_2 = \bar{s}, \quad C_2 = \frac{s_2 - \bar{s}}{2\beta},$$

$$\bar{s} = \frac{2(z_2 - z_1) - (\alpha s_1 + \beta s_2)}{t_2 - t_1},$$

$$\alpha = \xi - t_1, \quad \beta = t_2 - \xi.$$

Lemma 6.11.2 constructs a quadratic spline once we choose ξ . Note that ξ is a free node in that we do not know $s(\xi)$; $s(\xi)$ is allowed to be whatever is convenient.

We will now choose ξ to satisfy desirable shape properties. First, if $s_1, s_2 \geq 0$, then $s(t)$ is monotone if and only if $s_1 \bar{s} \geq 0$, which is equivalent to

$$2(z_2 - z_1) \geq (\xi - t_1)s_1 + (t_2 - \xi)s_2 \quad \text{if } s_1, s_2 \geq 0,$$

$$2(z_2 - z_1) \leq (\xi - t_1)s_1 + (t_2 - \xi)s_2 \quad \text{if } s_1, s_2 \leq 0.$$

To deal with curvature, we compute $\delta = (z_2 - z_1)/(t_2 - t_1)$, the average slope between t_1 and t_2 . If $(s_2 - \delta)(s_1 - \delta) \geq 0$, there must be an inflection point in $[t_1, t_2]$, and we can have neither a concave nor convex interpolant. If $|s_2 - \delta| < |s_1 - \delta|$, and ξ satisfies

$$t_1 < \xi \leq \bar{\xi} \equiv t_1 + \frac{2(t_2 - t_1)(s_2 - \delta)}{s_2 - s_1}, \quad (6.11.4)$$

then $s(t)$ in (6.11.3) is convex (concave) if $s_1 < s_2$ ($s_1 > s_2$). Furthermore, if $s_1 s_2 > 0$, it is also monotone. If $|s_2 - \delta| > |s_1 - \delta|$ and

$$t_2 + \frac{2(t_2 - t_1)(s_1 - \delta)}{s_2 - s_1} \equiv \underline{\xi} \leq \xi < t_2, \quad (6.11.5)$$

then $s(t)$ in (6.11.3) is convex (concave) if $s_1 < s_2$ ($s_1 > s_2$), and if $s_1 s_2 > 0$, s is monotone. Inequalities (6.11.4) and (6.11.5) give us a range of ξ that can be used to preserve shape conditions. These facts imply the following algorithm for finding a value for ξ such that $s(t)$ defined in (6.11.3) is a shape preserving interpolant on the interval $[t_1, t_2]$ of the data in (6.11.1):

Algorithm 6.3 Schumaker Shape-Preserving Interpolation for (6.11.1)

Initialization. Check if lemma 6.11.1 applies; if so, $\xi = t_2$, (6.11.3) gives $s(t)$, and STOP.

Step 1. Compute $\delta = (z_2 - z_1)/(t_2 - t_1)$.

Step 2. If $(s_1 - \delta)(s_2 - \delta) \geq 0$, set $\xi = (1/2)(t_1 + t_2)$ and STOP else, if $|s_2 - \delta| < |s_1 - \delta|$, then compute $\bar{\xi}$ using (6.11.4), set $\xi = (1/2)(t_1 + \bar{\xi})$, and STOP; if $|s_2 - \delta| \geq |s_1 - \delta|$, then compute $\underline{\xi}$ using (6.11.5), set $\xi = (1/2)(t_2 + \underline{\xi})$, and STOP.

This algorithm for ξ solves the problem for a single interval, $[t_1, t_2]$. We next consider a general interpolation problem. If we have Hermite data, namely we have $\{(z_i, s_i, t_i) | i = 1, \dots, n\}$, we then apply the shape-preserving interpolant algorithm to each interval to find $\xi_i \in [t_i, t_{i+1}]$. If we have Lagrange data, $\{(z_i, t_i) | i = 1, \dots, n\}$, we must first add estimates of the slopes and then proceed as we do with Hermite data. Schumaker suggests the following formulas for estimating slopes s_1 through s_n :

$$L_i = \left[(t_{i+1} - t_i)^2 + (z_{i+1} - z_i)^2 \right]^{1/2}, \quad i = 1, \dots, n-1,$$

$$\delta_i = \frac{z_{i+1} - z_i}{t_{i+1} - t_i}, \quad i = 1, \dots, n-1,$$

$$s_i = \begin{cases} \frac{L_{i-1}\delta_{i-1} + L_i\delta_i}{L_{i-1} + L_i} & \text{if } \delta_{i-1}\delta_i > 0 \\ 0, & \text{if } \delta_{i-1}\delta_i \leq 0 \end{cases}, \quad i = 2, \dots, n-1, \quad (6.11.6)$$

$$s_1 = -\frac{3\delta_1 - s_2}{2}, \quad s_n = \frac{3\delta_{n-1} - s_{n-1}}{2}.$$

Once we have the level and slope information data, the latter either given or computed, we can then proceed locally, choosing ξ_i which causes (6.11.3) to be a shape-preserving spline on each interval $[t_i, t_{i+1}] \equiv I_i$. The result of this scheme is a quadratic spline that is globally monotone if the data are monotone. Similarly for convexity and concavity. Furthermore the resulting spline is *co-monotone*, that is, s is increasing (decreasing) on I_i iff $z_i < z_{i+1}$ ($z_i > z_{i+1}$). This property follows from the local monotonicity of (6.11.3) and our imposing $s_i = 0$ wherever the slope changes. Preserving local curvature is also accomplished. If $\delta_i < \delta_{i+1} < \delta_{i+2} < \delta_{i+3}$ the data appear to be convex on I_{i+1} , and our construction is convex on I_{i+1} , since $s_{i+1} < s_{i+2}$ by construction; hence the Shumaker scheme is *co-convex*.

In table 6.5 we display the results of applying the Schumaker procedure to some familiar functions. We use uniformly spaced nodes on the indicated intervals. Note the small errors in approximating both f and f' . The first four columns display L^∞ and L^2 norms for the error in the approximation to f , first using estimated slopes from (6.11.6) and then using the true slopes. The last two columns display error

Table 6.5
Shape-preserving interpolation

Function	Nodes	f errors				f' errors	
		Estimated slopes		True slopes		True slopes	
		L^∞	L^2	L^∞	L^2	L^∞	L^2
\sqrt{x} on $[1, 2]$	3	6(-3)	2(-3)	3(-5)	1(-5)	1(-3)	5(-5)
	4	4(-3)	1(-3)	1(-5)	6(-6)	7(-4)	3(-5)
	5	2(-3)	7(-4)	8(-6)	3(-6)	5(-4)	2(-5)
\sqrt{x} on $[1, 20]$	3	3(-1)	1(-1)	2(-2)	5(-3)	3(-2)	1(-3)
	4	2(-1)	8(-2)	1(-2)	3(-3)	3(-2)	8(-4)
	5	2(-1)	5(-2)	8(-3)	2(-3)	2(-2)	6(-4)
$\sin x$ on $[0, 2\pi]$	5	3(-1)	9(-2)	2(-2)	8(-3)	7(-2)	1(-3)
	7	1(-1)	4(-1)	7(-3)	3(-3)	4(-2)	7(-4)
	10	5(-2)	1(-1)	3(-3)	1(-3)	2(-2)	4(-4)

norms for the implied estimate of f' where we use the true slopes of f . Using the true slopes helps substantially, and the resulting approximations are very good at approximating both f and f' .

6.12 Multidimensional Approximation

Most problems require us to approximate multidimensional functions. When we move beyond one dimension, several difficulties present themselves. We will discuss multidimensional interpolation and approximation methods, first by generalizing the one-dimensional methods via product formulations and then by constructing inherently multidimensional schemes.

General Theory of Multidimensional Approximation

We next present two basic theorems in multidimensional approximation. We first present Stone's theorem. Suppose that X is a compact metric space. The space of real-valued continuous functions, $C[X]$, is a linear vector space. It is also an algebra when we add multiplication, $*$; that is, the operations $+$ and $*$ over $C[X]$ satisfy distributivity and associativity, and for scalar $a \in R$, $(a * f) * g = a * (f * g)$. We say that $S \subset C[X]$ is a *subalgebra* if it is closed under $+$ and $*$; furthermore a subalgebra S separates points if for all distinct $x, y \in X$, there is $f \in S$ such that $f(x) \neq f(y)$.

THEOREM 6.12.1 (Stone's theorem) If X is a compact metric space and $(S, +, *)$ is a subalgebra of $(C[X], +, *)$ which contains the function $f(x) = 1$ and separates points in X , then S is dense in $C[X]$.

Proof See Cheney (1966, pp. 191–92). ■

The Weierstrass theorem is a special case of Stone's theorem, since the ordinary polynomials form a subalgebra of $C[R]$ and the linear monomial $f(x) = x$ separates any two points in R . Since any two points in R^n can be separated by some linear monomial, Stone's theorem implies that the ordinary multivariate polynomials are dense in $C[R^n]$, as are the tensor product bases of orthogonal polynomials. Stone's theorem can be used to show completeness of many linear approximation schemes, including ones built from nonpolynomial functions.

While linear approximation theory is very well developed, there are nonlinear schemes of approximation. We saw how Padé approximations effectively used rational approximations. A totally different approach to approximation is the subject of the remarkable Kolmogorov's theorem.

THEOREM 6.12.2 (Kolmogorov's theorem) For any n there exists an $\alpha \in [0, 1]^n$ and $2n + 1$ strictly monotone Lipschitz functions, $\varphi_l : [0, 1] \rightarrow R$, $l = 1, \dots, 2n + 1$, such that for any $f \in C[0, 1]^n$ there is a $\psi \in C[0, n]$ such that

$$f(x) = \sum_{l=1}^{2n+1} \psi \left(\sum_{i=1}^n \alpha_i \varphi_l(x_i) \right). \quad (6.12.1)$$

This is a striking theorem. It breaks down a multivariate function into a special sequence of addition, multiplication, and univariate operations. The whole approach is fundamentally different from linear procedures, focusing not on a sequence of nonlinear but on univariate function evaluations. One key difference between (6.12.1) and an orthogonal polynomial approach is that division is a permissible operation in ψ and φ_l , whereas it is not used when evaluating any polynomial. In practice, this is the key difference, for the computer only does division, multiplication, and addition. Therefore (6.12.1) is operationally equivalent to a special class of rational functions.

Unfortunately, implementation of Kolmogorov's theorem is difficult; the φ 's are nondifferentiable functions and very difficult to compute. While the direct practical usefulness of Kolmogorov's theorem is questionable, it does give us reason to examine alternative approaches to approximation. One such alternative are artificial neural networks, which we examine in a later section.

Lagrange Interpolation

Multidimensional Lagrange interpolation takes a finite set of conditions (the Lagrange data), $D \equiv \{(x_i, z_i)\}_{i=1}^N \subset R^{n+m}$, where $x_i \in R^n$ and $z_i \in R^m$, and finds a function $f: R^n \rightarrow R^m$ such that $z_i = f(x_i)$. Section 6.6 examined the case where $n = m = 1$. The multidimensional problem is much more complex, and occasionally impossible. For example, suppose that data at the four points are $\{P_1, P_2, P_3, P_4\} \equiv \{(1, 0), (-1, 0), (0, 1), (0, -1)\}$ in R^2 , and we want to interpolate using linear combinations of the functions $\{1, x, y, xy\}$. Let $z_i = f(P_i)$, $i = 1, 2, 3, 4$. The linear set of conditions for a polynomial $f(x, y) = a + bx + cy + dxy$ to satisfy $f(P_i) = z_i$ is

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix},$$

which is a singular system due to the column of zeros. Hence interpolation over these four points using $\{1, x, y, xy\}$ is not generally possible.

If we want to interpolate in R^n , then we must proceed carefully. Interpolation is possible on many grids. To determine that, we form the linear system as we did above, and check if it is nonsingular. By standard genericity logic, interpolation is possible on most grids. That is not good enough. We also want grids that do a good job and produce well-conditioned procedures.

Tensor Product Bases

Linear approximations begin with a basis of functions. We can use *tensor products* of univariate functions to form bases of multivariate functions. If A and B are sets of functions over $x \in R^n$, $y \in R^m$, their tensor product is

$$A \otimes B = \{\varphi(x)\psi(y) | \varphi \in A, \psi \in B\}. \quad (6.12.2)$$

Given a basis for functions of the single variable x_i , $\Phi^i = \{\varphi_k^i(x_i)\}_{k=0}^\infty$, we can build the *n-fold tensor product* basis for functions of n variables (x_1, x_2, \dots, x_n) by taking all possible n -term products of φ_k^i . The resulting basis is

$$\Phi = \left\{ \prod_{i=1}^n \varphi_{k_i}^i(x_i) \mid k_i = 0, 1, \dots, i = 1, \dots, n \right\} \quad (6.12.3)$$

One problem with tensor product bases is their size. We will only want to use finite subsets of the full tensor product basis. It will be natural to take the first m elements (typically those of degree less than m) of each univariate basis and construct the tensor product of these subbases. If Φ is a basis for univariate functions and $\Psi \subset \Phi$ has m elements, then the n -dimensional tensor product of Ψ , has m^n elements. This exponential growth in dimension makes it quite costly to use the full tensor product subbasis.

Multidimensional versions of splines can also be constructed through tensor products; here *B*-splines would be useful. There are more sophisticated methods available, but the details are more complex. We refer the interested reader to Nurnberger (1989).

Our discussion of orthogonal polynomials and least squares approximation generalizes directly to the multivariate case. To compute n -dimensional least squares approximation, we need only note that the basis Φ from (6.12.3) inherits the orthogonality properties of the individual Φ^i . Hence, if the elements of Φ^i are orthogonal with respect to the weighting function $w_i(x_i)$ over the interval $[a_i, b_i]$, then the least squares approximation of $f(x_1, \dots, x_n)$ in Φ is

$$\sum_{\varphi \in \Phi} \frac{\langle \varphi, f \rangle}{\langle \varphi, \varphi \rangle} \varphi, \quad (6.12.4)$$

where the product weighting function

$$W(x_1, x_2, \dots, x_n) = \prod_{i=1}^n w_i(x_i) \quad (6.12.5)$$

defines $\langle \cdot, \cdot \rangle$ over $D = \prod_i [a_i, b_i]$ in

$$\langle f(x), g(x) \rangle = \int_D f(x)g(x) W(x) dx. \quad (6.12.6)$$

The following procedure for Chebyshev approximation in R^2 illustrates how one adapts one-dimensional methods to higher dimensions:

Algorithm 6.4 Chebyshev Approximation Algorithm in R^2

Objective: Given a function $f(x, y)$ defined on $[a, b] \times [c, d]$, find its Chebyshev polynomial approximation $p(x, y)$.

Step 1. Compute the $m \geq n + 1$ Chebyshev interpolation nodes on $[-1, 1]$:

$$z_k = -\cos\left(\frac{2k-1}{2m}\pi\right), \quad k = 1, \dots, m.$$

Step 2. Adjust the nodes to the $[a, b]$ and $[c, d]$ intervals:

$$x_k = (z_k + 1)\left(\frac{b-a}{2}\right) + a, \quad k = 1, \dots, m,$$

$$y_k = (z_k + 1)\left(\frac{d-c}{2}\right) + c, \quad k = 1, \dots, m.$$

Step 3. Evaluate f at the approximation nodes:

$$w_{k,l} = f(x_k, y_l), \quad k = 1, \dots, m, \quad l = 1, \dots, m.$$

Step 4. Compute Chebyshev coefficients, $a_{ij}, i, j = 0, \dots, n$:

$$a_{ij} = \frac{\sum_{k=1}^m \sum_{l=1}^m w_{k,l} T_i(z_k) T_j(z_l)}{(\sum_{k=1}^m T_i(z_k)^2)(\sum_{l=1}^m T_j(z_l)^2)}$$

to arrive at the approximation for $f(x, y), x \in [a, b]; y \in [c, d]$:

$$p(x, y) = \sum_{i=0}^n \sum_{j=0}^n a_{ij} T_i\left(2\frac{x-a}{b-a} - 1\right) T_j\left(2\frac{y-c}{d-c} - 1\right).$$

Complete Polynomials

Tensor product collections have the disadvantage of growing exponentially as the dimension increases. We will now turn to bases that grow only polynomially as the dimension increases and see why they can be quite good even though they have far fewer members. Recall that Taylor's theorem for many dimensions produces the approximation

$$\begin{aligned} f(x) &\doteq f(x^0) + \sum_{i=1}^n \frac{\partial f}{\partial x_i}(x^0)(x_i - x_i^0) \\ &\quad \vdots \\ &+ \frac{1}{k!} \sum_{i_1=1}^n \cdots \sum_{i_k=1}^n \frac{\partial^k f}{\partial x_{i_1} \cdots \partial x_{i_k}}(x^0)(x_{i_1} - x_{i_1}^0) \cdots (x_{i_k} - x_{i_k}^0). \end{aligned}$$

Notice the terms used in the k th-degree Taylor series expansion. For $k = 1$, Taylor's theorem for n dimensions used the linear functions $\mathcal{P}_1^n \equiv \{1, x_1, x_2, \dots, x_n\}$. For $k = 2$, Taylor's theorem used

$$\mathcal{P}_2^n \equiv \mathcal{P}_1^n \cup \{x_1^2, \dots, x_n^2, x_1 x_2, x_1 x_3, \dots, x_{n-1} x_n\}.$$

\mathcal{P}_2^n contains some product terms, but not all; for example, $x_1 x_2 x_3$ is not in \mathcal{P}_2^n . In general, the k th-degree expansion of n dimensional functions uses functions in

$$\mathcal{P}_k^n \equiv \left\{ x_1^{i_1} \cdots x_n^{i_n} \mid \sum_{l=1}^n i_l \leq k, 0 \leq i_1, \dots, i_n \right\}.$$

The set \mathcal{P}_k^n is called the *complete set of polynomials of total degree k in n variables*.

Using complete sets of polynomials will give us a way of constructing bases for multivariate approximation that are often more efficient than tensor products. If $\Psi_k \equiv \{1, x, \dots, x^k\}$ is the collection of k th-degree basis functions, then its n -dimensional tensor product contains many more elements than \mathcal{P}_k . For example, table 6.6 compares the size of \mathcal{P}_k^n to the tensor product.

The key difference is that the tensor product basis grows exponentially in dimension n for fixed k , but the set of complete polynomials grows polynomially. However, from the perspective of the rate of convergence, many of the elements in a tensor product are excessive. Taylor's theorem tells us that the elements of \mathcal{P}_k^n will yield an approximation near x^0 which has k th-degree convergence asymptotically. The n -fold tensor product of Ψ_k can give us only k th-degree convergence, since it does not

Table 6.6
Sizes of alternative bases

Degree k	\mathcal{P}_k^n	Tensor product Ψ_k^n
2	$1 + n + \frac{n(n+1)}{2}$	3^n
3	$1 + n + \frac{n(n+1)}{2} + n^2 + \frac{n(n-1)(n-2)}{6}$	4^n

contain all terms of total degree $k + 1$. Therefore, in terms of asymptotic convergence, the complete polynomials will give us as good an approximation as the tensor product with far fewer elements. We use complete polynomials of finite degree in the hope that this asymptotic behavior holds for polynomials of practical degree.

6.13 Finite Element Approximations

A finite element approach to approximation uses basis functions that are zero over most of the domain. It is a local approach to approximation, in contrast to orthogonal polynomial approach where the basis elements are each nonzero everywhere. We present two two-dimensional examples.

Bilinear Interpolation

Bilinear interpolation constructs an approximation that interpolates the data linearly in both coordinate directions. Suppose that we have the values of $f(x, y)$ at $(x, y) = (\pm 1, \pm 1)$. Then the following four functions form a cardinal interpolation basis on $[-1, 1]^2$:

$$\begin{aligned}\varphi_1(x, y) &= \frac{1}{4}(1-x)(1-y), & \varphi_2(x, y) &= \frac{1}{4}(1+x)(1-y), \\ \varphi_3(x, y) &= \frac{1}{4}(1+x)(1+y), & \varphi_4(x, y) &= \frac{1}{4}(1-x)(1+y).\end{aligned}\tag{6.13.1}$$

Notice that each of these functions is zero at all but one of the points $(\pm 1, \pm 1)$. Therefore the approximation to f on the square $[-1, 1] \times [-1, 1]$ is

$$\begin{aligned}f(-1, -1)\varphi_1(x, y) + f(1, -1)\varphi_2(x, y) \\ + f(1, 1)\varphi_3(x, y) + f(-1, 1)\varphi_4(x, y)\end{aligned}\tag{6.13.2}$$

The basis function φ_1 is graphed in figure 6.10; note the mixed curvature of the surface. In particular, it is linear only on the edges, and has a saddle point curvature on

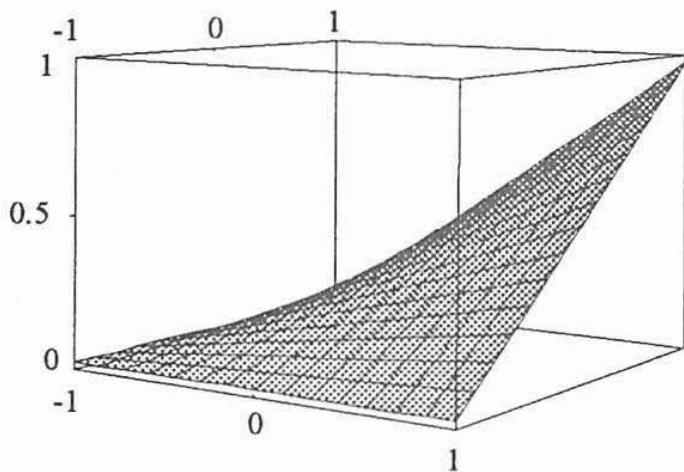


Figure 6.10
Typical bilinear element

the interior; that is, it is convex in the $(-1, -1)$ to $(1, 1)$ direction, as can be seen in figure 6.10, and concave in the $(-1, 1)$ to $(1, -1)$ direction. This is true of each of the basis functions in (6.13.1).

If we have data at the vertices of a rectangle other than $[-1, 1] \times [-1, 1]$ we can interpolate using the φ_i in (6.13.1) after a linear change of variables. If the data are at the vertices $[a, b] \times [c, d]$, then the linear map

$$l(x, y) = \left(-1 + 2 \frac{x-a}{b-a}, -1 + 2 \frac{y-c}{d-c} \right)$$

maps $[a, b] \times [c, d]$ onto $[-1, 1] \times [-1, 1]$, and the functions $\psi_i(x, y) \equiv \varphi_i(l(x, y))$, $i = 1, 2, 3, 4$, are cardinal basis functions for bilinear interpolation over $[a, b] \times [c, d]$ based on $f(a, c)$, $f(a, d)$, $f(b, c)$, and $f(b, d)$.

If we have Lagrange data on a two-dimensional lattice, we compute the interpolants on each square and piece them together. While this is similar to how we constructed piecewise-polynomial approximations on R , we must be more careful here because of the extra difficulties created by the higher dimensionality. Since we are trying to compute a continuous interpolant, we must be sure that the individual pieces meet continuously at common edges. In the piecewise-polynomial case, this was ensured by the fact that the intervals had common nodes. That is not enough, in general, in higher dimensions because adjacent regions share faces as well as vertices. Therefore interpolants in neighboring regions must agree along all common faces, edges, and vertices. In bilinear interpolation this will happen because any two approximations overlap only at the edges of rectangles, and on those edges the

approximation is the linear interpolant between the common end points. Higher-order interpolation generates much more complex problems, which we will not cover here.

An equivalent approach to bilinear interpolation is to take the linear interpolation basis in each dimension, the tent functions, and construct the tensor product, which will also be a cardinal function basis. This fact is useful but misleading, since finite element approximation does not generally reduce to a tensor product formulation. In either case the result has a pockmarked shape, like a golf ball, because of the mixed curvature inside each rectangle.

Simplicial 2-D Linear Interpolation

While bilinear interpolation is used sometimes, many prefer simplicial triangular elements. We start by triangulating the x - y plane. Let $P_1P_2P_3$ be the canonical linear triangular element, where $P_1 = (0, 0)$, $P_2 = (0, 1)$, and $P_3 = (1, 0)$. We are looking for linear functions $\varphi_i(x, y)$ such that $\varphi_i(P_i) = 1$ and $\varphi_i(P_j) = 0$ for $i \neq j$, $i, j = 1, 2, 3$. The functions $\varphi_1(x, y) = 1 - x - y$, $\varphi_2(x, y) = y$, and $\varphi_3(x, y) = x$ satisfy the cardinal interpolation conditions on the set $\{P_1, P_2, P_3\}$. If we examine $P_2P_3P_4$ where $P_4 = (1, 1)$, then $\varphi_4(x, y) = 1 - x$, $\varphi_5(x, y) = 1 - y$, and $\varphi_6(x, y) = x + y - 1$ are cardinal basis functions on the set $\{P_2, P_3, P_4\}$. Therefore, on the square $P_1P_2P_4P_3$, the interpolant, \hat{f} , is

$$\hat{f}(x, y) = \begin{cases} f(0, 0)(1 - x - y) + f(0, 1)y + f(1, 0)x & \text{if } x + y \leq 1, \\ f(0, 1)(1 - x) + f(1, 0)(1 - y) + f(1, 1)(x + y - 1), & \text{if } x + y \geq 1. \end{cases} \quad (6.13.3)$$

Elementary considerations show that piecewise-planar approximation is no worse than bilinear approximation. Since bilinear interpolation uses only $1, x, y$, and xy , the highest degree of completeness is 1. Therefore, as the rectangles get smaller, we only get linear convergence, the same as linear triangular elements.

Multidimensional Linear Interpolation

Both the bilinear and simplicial interpolation methods can be extended to arbitrarily high dimensions. Weiser and Zarantonello (1988) develop simple recursive formulas for both methods. We describe both methods.

Without loss of generality, we can assume that we are working with the hypercube $\mathcal{C} \equiv \{x \in R^n | 0 \leq x_i \leq 1, i = 1, \dots, n\}$. We also assume that we have Lagrange data on the vertices of \mathcal{C} , $V \equiv \{x \in \mathcal{C} | x_i \in \{0, 1\}, i = 1, \dots, n\}$. Therefore we know $f(x)$ for $x \in V$ and want to compute an interpolating function $f(x)$ for $x \in \mathcal{C}$.

The multilinear interpolant, $\hat{f}(x)$ is defined recursively by a sequence of linear interpolations in successive dimensions. The first step is to define f by linear interpolation in dimension n :

$$\begin{aligned}\hat{f}(x_1, \dots, x_n) &= \hat{f}(x_1, \dots, x_{n-1}, 0) \\ &\quad + x_n(\hat{f}(x_1, \dots, x_{n-1}, 1) - \hat{f}(x_1, \dots, x_{n-1}, 0)),\end{aligned}$$

where the elements on the lefthand side of (6.13.4) are now $(n-1)$ -dimensional interpolants. These in turn are inductively defined by successive interpolation on the next lower dimension:

$$\begin{aligned}\hat{f}(x_1, \dots, x_l, j_{l+1}, \dots, j_n) &= \hat{f}(x_1, \dots, x_{l-1}, 0, j_{l+1}, \dots, j_n) \\ &\quad + x_l(\hat{f}(x_1, \dots, x_{l-1}, 1, j_{l+1}, \dots, j_n) - \hat{f}(x_1, \dots, x_{l-1}, 0, j_{l+1}, \dots, j_n))\end{aligned}$$

where the $j_i = 0, 1$ for $i = l+1, \dots, n$. This scheme ends when each component of \hat{f} is either 0 or 1, in which case the point is in V , and we know the value from the Lagrange interpolation data. This scheme is rather costly, requiring roughly 2^n multiplications to evaluate the interpolation at just one point. The general curvature is also complex and difficult to perceive. These considerations make this a poor method.

We next generalize the simplicial 2-D linear interpolation method above to R^n . The idea is to pick a simplicial subdivision of \mathcal{C} and compute the linear interpolation on each simplex. This sounds difficult, but the following simple scheme computes a piecewise-linear simplicial interpolant.

Suppose that we want to compute $f(x)$ where $x \in [0, 1]^n$. We first need to determine the simplex which contains x . To do this, we find the permutation π of the indexes such that

$$0 \leq x_{\pi(1)} \leq x_{\pi(2)} \leq \dots \leq x_{\pi(n)} \leq 1. \quad (6.13.6)$$

This is done by sorting the components of x according to the value of each component. The complete procedure is summarized in algorithm 6.5, where e^i denotes the vector with 1 in component i and zero elsewhere:

Algorithm 6.5 Multidimensional Simplicial Interpolation

Objective: Given x on V , compute $f(x)$ for $x \in \mathcal{C}$.

Initialization. Construct the permutation π satisfying (6.13.4). Let $s^0 = (1, \dots, 1)$, $y_0 = f(s^0)$.

Step 1. For $i = 1, \dots, n$, $s^i = s^{i-1} - e^{\pi(i)}$ and $y_i = y_{i-1} + (1 - x_{\pi(i)})(f(s^i) - f(s^{i-1}))$.

Step 2. Report y_n as the value of $f(x)$ and STOP.

This procedure can be adapted, through linear transformations, to compute the simplicial interpolant at any point x in any hypercube. Because of the linearity of the approximation, interpolants of different hypercubes will match at common points, resulting in a continuous interpolation globally. The simplicial interpolation method is less costly than multilinear interpolation and has simpler curvature.

6.14 Neural Networks

The previous approximation procedures are based on linear combinations of polynomial and trigonometric functions. Neural networks use simple but powerful non-linear approximation schemes. Below we state the important definitions and results.

Neural Network Functional Form

For our purposes, a neural network is a particular type of functional form used in approximation. A *single-layer* neural network is a function of the form

$$F(x; \beta) \equiv h\left(\sum_{i=1}^n \beta_i g(x_i)\right), \quad (6.14.1)$$

where $x \in R^n$ is the vector of inputs and h and g are scalar functions. A common form assumes that $g(x) = x$, reducing (6.14.1) to the form $h(\beta^T x)$. A *single hidden-layer feedforward* network has the form

$$F(x; \beta, \gamma) \equiv f\left(\sum_{j=1}^m \gamma_j h\left(\sum_{i=1}^n \beta_i^j g(x_i)\right)\right), \quad (6.14.2)$$

where h is called the *hidden-layer activation function*.

The source of this terminology is displayed in figure 6.11. Figure 6.11b displays (6.14.2). Each input is processed by a node, where each node is some function. The output of each node is then fed to one more node for further processing that produces the output. In figure 6.11a we illustrate the single hidden-layer structure.

Neural Network Approximation

The data for a neural network consists of (y_i, x^i) pairs such that y_i is supposed to be the output of a neural network if x^i is the input. This requirement imposes conditions on the parameter matrix β in (6.14.1) and β and γ in (6.14.2). Indeed, when we use

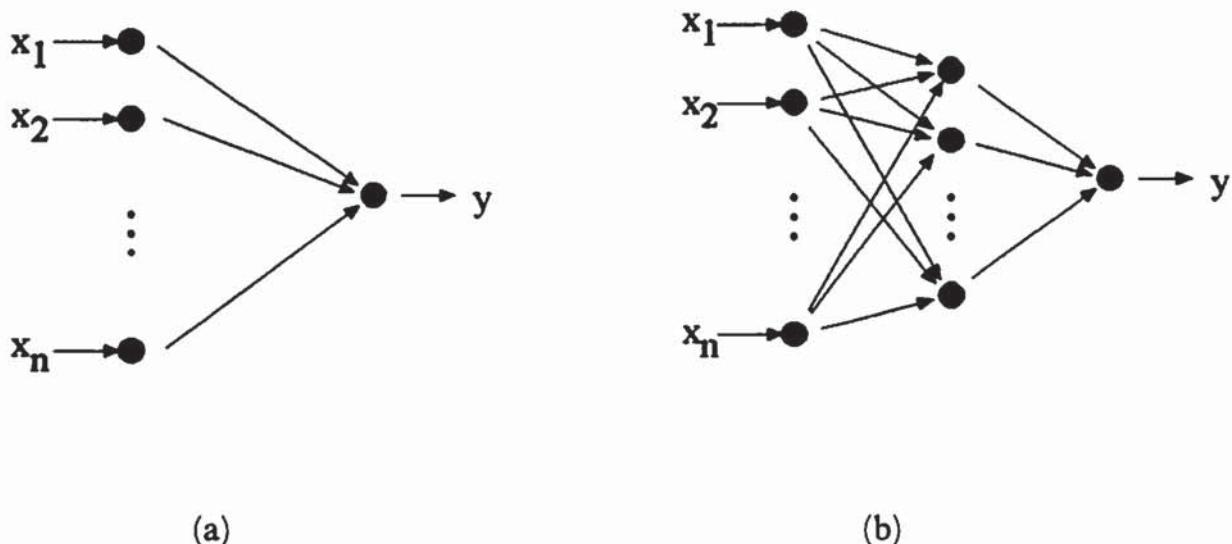


Figure 6.11
Neural networks

single-layer neural networks, the objective is to find β to solve

$$\min_{\beta} \sum_j (y_j - F(x^j; \beta))^2,$$

and when we use a single hidden-layer feedforward network, our objective is to find β and γ to solve

$$\min_{\beta, \gamma} \sum_j (y_j - F(x^j; \beta, \gamma))^2.$$

These objectives are just instances of nonlinear least squares fitting. Because of the nonlinearity, we do not have simple formula for the β and γ coefficients. Instead, we must use general minimization methods. Experience also informs us that there often are several local minima.

Neural networks are useful for approximation since they are flexible enough to fit general functions. The power of neural network approximation is indicated by the following theorem of Horni, Stinchcombe and White (1989).

THEOREM 6.14.1 Let G be a continuous function, $G: R \rightarrow R$, such that either (1) $\int_{-\infty}^{\infty} G(x)dx$ is finite and nonzero and G is L^p for $1 \leq p < \infty$ or (2) G is a *squashing function* $G: R \rightarrow [0, 1]$; that is, G is nondecreasing, $\lim_{x \rightarrow \infty} G(x) = 1$, and $\lim_{x \rightarrow -\infty} G(x) = 0$. Define

$$\Sigma^n(G) = \left\{ g: R^n \rightarrow R \mid g(x) = \sum_{j=1}^m \beta_j G(w^j \cdot x + b_j), \beta_j, b_j \in R, \right. \\ \left. w^j \in R^n, w^j \neq 0, m = 1, 2, \dots \right\}$$

to be the set of all possible single hidden-layer feedforward neural networks, using G as the hidden layer activation function. Let $f: R^n \rightarrow R$ be continuous. Then for all $\varepsilon > 0$, probability measures μ , and compact sets $K \subset R^n$, there is a $g \in \Sigma^n(G)$ such that

$$\sup_{x \in K} |f(x) - g(x)| \leq \varepsilon$$

and

$$\int_K |f(x) - g(x)| d\mu \leq \varepsilon.$$

The class of functions covered by theorem 6.14.1 is quite broad. The functions that fit (1) in theorem 6.14.1 include all probability density functions with compact support. Note that any squashing function is a cumulative distribution function, and vice versa. A common choice for G is the *sigmoid function*, $G(x) = (1 + e^{-x})^{-1}$. Also note that the form of the functions in $\Sigma^n(G)$ is a simple case of (6.14.2).

Theorem 6.14.1 is a universal approximation result that justifies the use of neural network approximation and helps to explain its success. Note the simplicity of the functional forms; this simplicity makes neural network approximations easy to evaluate. Barron (1993) shows that neural networks are efficient functional forms for approximating multidimensional functions. It is unclear if this efficiency is sufficient compensation for the operational problems of finding a fit. The theoretical development of neural networks is proceeding but is inherently difficult because of the non-linearity of this approach.

Note the strong similarities between neural networks and the universal approximators in Kolmogorov's theorem. Both proceed by successive application of addition, multiplication, and scalar functions. In both cases the innermost one-dimensional functions are monotonic. However, there are differences. In neural networks the innermost function, $g(x)$, is fixed, whereas there are $2n + 1$ innermost functions in Kolmogorov's representation. Neural networks fit functions by different choices of weights γ_j and β_i^j , whereas the Kolmogorov approximation alters only the $2n + 1$ innermost weights and the $\psi(x)$ function. However, these distinctions are small from

a practical point of view, and both methods are quite different from the approach implicit in the Stone-Weierstrass theorem.

6.15 Further Reading and Summary

This chapter has presented basic methods for constructing approximations of functions. These methods use various kinds of data including function values, derivative values, and *a priori* shape information. Many approximation methods resemble statistical techniques; however, we can do much better than statistical methods typically do because we have control over the values of the independent variables. The major software libraries, such as IMSL and NAG, contain many programs for constructing polynomial, spline, and rational approximations.

Many of the interpolation ideas described above, such as polynomial interpolation, were presented in simple ways which are often ill-conditioned and inefficient. More efficient methods are presented in most numerical analysis texts, such as Atkinson (1989), Rice (1983), and de Boor (1978).

There is a large literature on finite element approximations of multidimensional functions. Burnett (1987) gives a very readable presentation of the basic methods. Because the finite element method is largely aimed at engineers, it is most highly developed for one-, two-, and three-dimensional problems. For higher dimensions, economists will have to adapt existing methods, but the basic ideas will still hold: low-order approximations within elements, simplicial or rectangular, with smoothness conditions across the boundaries of elements.

There is a large literature on shape-preserving splines, much of it spurred on by the demand for efficient computer graphics methods. Rasch and Williamson (1990) present a general discussion and comparison of several shape-preserving schemes in the one-dimensional case. There has also been success in producing shape-preserving interpolants for data over R^2 ; Costantini and Fontanella (1990) is a recent example. Again this literature is concerned with at most three-dimensional problems, presumably because of the focus on “visually appealing” approximations. Economists need shape-preserving approximation methods for higher dimensions as well.

The sections above have relied on polynomials to approximate functions over an interval. We saw that rational functions were useful in constructing approximations based at a point and had some advantages over polynomial approximations. We can also compute minimax rational approximations of functions over compact intervals. As with Padé approximations rational approximants are often superior for a given number of free parameters. An important difficulty is that rational approximations are not linear functions of the data. The absence of a vector space formulation

makes them more cumbersome to compute and make the theory much more difficult. For these reasons rational approximation methods have not been used as much as orthogonal polynomial and piecewise polynomial approximations.

The neural network literature is growing rapidly, but the potential value of neural networks is not clear. White (1992) presents the basic ideas, and universal approximation results, such as those of Barron (1993), Gallant and White (1988), and Hornik et al. (1989, 1990), indicate that neural network approximations can produce efficient approximations. Unfortunately, solving the least squares solution to the fitting problem is quite difficult because of multiple local minima. Golomb (1959) discusses Kolmogorov's theorem.

Exercises

1. Compute the degrees 1, 2, and 3 Taylor expansions for $f(x) = (x^{1/2} + 1)^{2/3}$ near $x_0 = 1$. Compute the log-linear approximation, the log-quadratic approximation, and the log-cubic approximation of f near $x_0 = 1$. Compute the $(1, 1)$ Padé approximation based at $x_0 = 1$. Compare the quality of these approximations for $x \in [0.2, 3]$.
2. Compute the condition number of the Vandermonde matrix for $x_i = i/n$ for $n = 3, 5, 10, 20, 50$.
3. Write a program to compute the coefficients of the Lagrange interpolating polynomial $p(x)$ given $\{y_i, x_i\}_{i=1}^n$ data. Write another program that uses those coefficients and computes $p(x)$. Use these programs to interpolate the values of $f(x) = x^{1/3}$ on the set $X = \{i/5 | i = 1, 2, 3, 4, 5\}$, and compare $x^{1/3}$ and this interpolant on $[0.1, 0.5]$.
4. Write a program to compute the coefficients of the Hermite interpolating polynomial $p(x)$ given $\{y_i, y'_i, x_i\}_{i=1}^n$ data. Write another program that uses those coefficients and computes $p(x)$. Apply this to the values of $x^{1/3}$ and its derivatives on $[0.1, 0.5]$ to construct the degree 9 Hermite interpolant on the set $X = \{i/5 | i = 1, 2, 3, 4, 5\}$. Compare $x^{1/3}$, this Hermite interpolant on $[0.1, 0.5]$, and the Lagrange interpolant constructed in exercise 3.
5. Compute the n -term Legendre approximation for $f(x)$ on $[-1, 1]$ for the following cases:
 - a. $f(x) = (x + 2)^5$, $n = 3, 4, 5$
 - b. $f(x) = (x + 2)^{-2}$, $n = 3, 4, 5$
 In both cases compute the L^∞ and L^2 errors over $[-1, 1]$.

6. Express the orthogonal polynomial family with weighting function 1 on the interval $[a, b]$ in terms of Legendre polynomials. Express the orthogonal polynomial family with weighting function $e^{-(ax+b)}$ on the interval $[d, \infty)$ in terms of Laguerre polynomials. Express the orthogonal polynomial family with weighting function $e^{-(ax^2+bx+c)}$ on $(-\infty, \infty)$ in terms of Hermite polynomials.
7. Compute the linear minmax approximation to e^x on $[0, 1]$. Compare it to the Legendre approximation and the Chebyshev approximation.

8. Compute the upper bound for $\rho_n(f)$ given by Jackson's theorem for $f(x) = x^{1/2}$ on the interval $[0.1, 2.0]$. Compare it to the bound on the interval $[0, 2.0]$.
9. Compute the natural cubic spline approximation to $x^{1/4}$ on $[1, 10]$. Plot L^2 and L^∞ errors against the number of nodes used.
10. Compute tables 6.1 and 6.2 for x^γ for $\gamma = -0.5, -2$, and -5 . Also use neural networks to approximate these functions over $[1, 10]$. Plot the errors using equi-spaced Lagrange data.
11. Newton's method uses a sequence of linear approximations to converge to a zero of $f(x) = 0, x \in R$. Develop an algorithm that uses a sequence of (1,1) Padé approximations to solve $f(x) = 0$. Compare the methods' performances for solving $x^2 - 2$ with the initial guess $x = 0$. What are the apparent rates of convergence?
12. Write a program to implement Schumaker's quadratic shape-preserving approximation. Apply it to the function $V(k) = k^\alpha$ on $[0, 10]$ for $\alpha = 0.01, 0.25, 0.80$. Compare the results when you use slope information to the results when you just approximate the slopes. Compare the results with Chebyshev interpolation.
13. Write a program that takes an increasing function $f(x)$ and computes the inverse function. Use Lagrange, Hermite, and shape-preserving interpolation schemes, and test the program on the functions x^3 , $\sin \pi x/2$, and e^x on $[0, 2]$ using the points $\{0, 0.5, 1.0, 1.5, 2.0\}$.
14. Let $f(k, l) = (k^{1/2} + l^{1/2})^2$. Compute its quadratic Taylor expansion around $(1, 1)$. Plot the error over $[0, 2.5] \times [0, 2.5]$.
15. Construct an interpolant of the function $V(x, y) = (x^\alpha + y^\alpha)^{\gamma+1}/(\gamma+1)$ on $[a, b] \times [c, d]$ using the triangular element approach. Let $\alpha = 0.1, 0.3, 0.5, 0.9$, $\gamma = -0.5, -1.1, -3, -5$, $a, c = 0.1, 1$, $b = 2a$, $d = 2c, 10c$.
16. Let
$$V(k_1, k_2, k_3) = (k_1 + k_2 + k_3)^{1/4} - (k_1 - k_2)^2 - (k_1 - 3k_3)^2 - (k_2 - 2k_3)^2.$$
Compute $V(k)$ on the uniform grid of 11^3 points on $[1, 3]^3$. Use these points to compute neural network approximations and Chebyshev approximations for $V(k)$. Which type of approximation does best (smallest error in supremum norm) per floating operation?
17. Consider a two-person, two-good exchange economy with total endowment of each good being $(1, 1)$. Assume agent one's utility function is $x^{1/3}y^{2/3}$ and agent two's utility function is $\sqrt{x} + 2\sqrt{y}$. Approximate the contract curve with (a) a Chebyshev polynomial, (b) a cubic spline, and (c) a Schumaker quadratic shape-preserving spline.

7

Numerical Integration and Differentiation

Numerical evaluation of a definite integral is a frequent problem encountered in economic modeling. If a firm pays a continuous stream of dividends, $d(t)$, and the interest rate is r , then the present value of the dividends equals $\int_0^\infty e^{-rt}d(t)dt$. If the random variable X is distributed $N(0, 1)$, then the expectation of $f(X)$ is $(2\pi)^{-1/2} \int_{-\infty}^{\infty} f(x)e^{-x^2/2}dx$. In Bayesian statistics, if one's prior belief over the parameter space, Θ , has density $f(\theta)$, if the data are X , and if $g(X|\theta)$ is the density of X conditional on θ , then the posterior mean belief is $(\int \theta g(X|\theta)f(\theta)d\theta)/(\int g(X|\theta)f(\theta)d\theta)$. Not only do integrals arise naturally in formulations of economic and econometric problems, but we will often introduce them as part of our numerical procedures. For example, computing the coefficients of orthogonal polynomial approximations in chapter 6 involves computing integrals.

The general problem in numerical integration, also called *quadrature* or *cubature*, is to compute $\int_D f(x)dx$ where $f: R^n \rightarrow R$ is an integrable function over the domain $D \subset R^n$. All numerical integration formulas use a finite number of evaluations of the integrand, f , and use a weighted sum of those values to approximate $\int_D f(x)dx$.

We examine several methods in this chapter and the next two. The reason for our studying quadrature so intensively is that analysts typically will have to choose among several alternatives. The most generally applicable methods are slow. Alternative methods are very efficient, but they require strong conditions on the integrand. Knowing when one can use the efficient methods can improve running times by orders of magnitude. Also analysts will frequently code the integration methods themselves. Since so many decisions must be made by the analyst, he must be aware of the critical mathematical factors. Of course one could economize on thinking and choose a simple method; however, the returns to careful analysis are particularly high in the case of quadrature.

Quadrature methods differ in how they choose where to evaluate the integrand and how to use the evaluations. The first quadrature methods we examine use ideas from approximation and interpolation theory. The second class of methods are called sampling methods and are examined in the next two chapters.

7.1 Newton-Cotes Formulas

The *Newton-Cotes quadrature formulas* use ideas from piecewise-polynomial approximation theory. They evaluate f at a finite number of points, use this information to construct a piecewise-polynomial approximation of f , and then integrate this approximation of f to approximate $\int_D f(x)dx$. This section will present various Newton-Cotes formulas and their error properties.

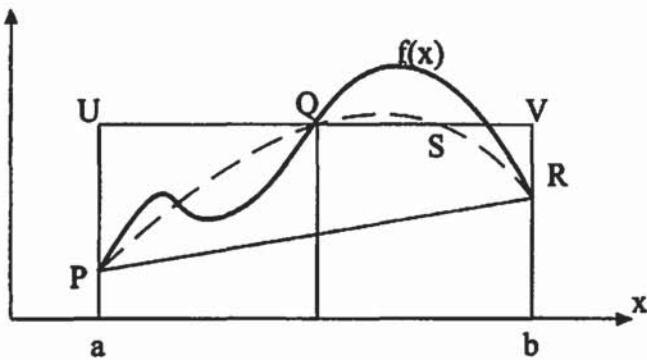


Figure 7.1
Newton-Cotes rules

Consider the graph in figure 7.1. Suppose that f is the solid curve through the points P, Q , and R . The integral $\int_a^b f(x) dx$ is the area under the function f and above the horizontal axis. Three approximations are immediately apparent. The box $aUQVb$ approximates f with a constant function equaling f at Q , which is the midpoint of $[a, b]$. The trapezoid $aPRb$ approximates f with a straight line through points P and R . The area under the broken curve $PQSR$ approximates f with a parabola through P, Q , and R . These approximations are based on one, two, and three evaluations of f , respectively, which are used to compute interpolating polynomials of degree one, two, and three. This approach yields *Newton-Cotes quadrature formulas*. We will now examine particular cases.

Midpoint Rule

The simplest quadrature formula is the *midpoint rule*, implied by (7.1.1):

$$\int_a^b f(x) dx = (b - a)f\left(\frac{a + b}{2}\right) + \frac{(b - a)^3}{24}f''(\xi) \quad (7.1.1)$$

for some $\xi \in [a, b]$. We will express many integration formulas in the fashion of (7.1.1), where the first terms comprise the integration rule and the last term is the error of the integration rule. Hence the midpoint rule is the first term on the RHS of (7.1.1), and the second term is its error term. Equation (7.1.1) is proved by applying Taylor's theorem and the intermediate value theorem. The midpoint rule is the simplest example of an *open rule*, which is a rule that does not use the end points.

This approximation is too coarse to be of value generally. Instead, we break the interval $[a, b]$ into smaller intervals, approximate the integral over each of the smaller intervals, and add those approximations. The result is a *composite rule*. Let $n \geq 1$

be the number of intervals, $h = (b - a)/n$, and $x_j = a + (j - \frac{1}{2})h$, $j = 1, 2, \dots, n$. Then the *composite midpoint rule* derives from the equation

$$\int_a^b f(x) dx = h \sum_{j=1}^n f(x_j) + \frac{h^2(b-a)}{24} f''(\xi) \quad (7.1.2)$$

for some $\xi \in [a, b]$. Notice that the error is proportional to h^2 ; doubling the number of quadrature nodes will halve the step size h and reduce the error by about 75 percent. Therefore the composite midpoint rule¹ converges quadratically for $f \in C^2$.

Trapezoid Rule

The trapezoid rule is based on the linear approximation of f using only the value of f at the endpoints of $[a, b]$. The trapezoid rule is

$$\int_a^b f(x) dx = \frac{b-a}{2} [f(a) + f(b)] - \frac{(b-a)^3}{12} f''(\xi) \quad (7.1.3)$$

for some $\xi \in [a, b]$. The trapezoid rule is the simplest example of a *closed rule*, which is a rule that uses the end points. Let $h = (b - a)/n$, and $x_i = a + ih$, and let f_j denote $f(x_j)$; the composite trapezoid rule is

$$\int_a^b f(x) dx = \frac{h}{2} [f_0 + 2f_1 + \dots + 2f_{n-1} + f_n] - \frac{h^2(b-a)}{12} f''(\xi) \quad (7.1.4)$$

for some $\xi \in [a, b]$,

Simpson's Rule

Piecewise-linear approximation of f in the composite trapezoid rule is unnecessarily coarse if f is smooth. An alternative is to use a piecewise-quadratic approximation of f which uses the value of f at a , b , and the midpoint, $\frac{1}{2}(a+b)$. The result is Simpson's rule over the interval $[a, b]$. Simpson's rule is

$$\int_a^b f(x) dx = \left(\frac{b-a}{6} \right) \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right] - \frac{(b-a)^5}{2880} f^{(4)}(\xi) \quad (7.1.5)$$

for some $\xi \in [a, b]$.

We next construct corresponding $(n+1)$ -point composite rule over $[a, b]$. Let $n \geq 2$ be an even number of intervals; then $h = (b - a)/n$, $x_j = a + jh$, $j = 0, \dots, n$,

1. The term "composite" is often dropped, and the term "midpoint rule" includes the composite version. The same is true for the rules below.

Table 7.1
Some simple integrals

Rule	Number of points	$\int_0^1 x^{1/4} dx$	$\int_1^{10} x^{-2} dx$	$\int_0^1 e^x dx$	$\int_1^{-1} (x + 0.05)^+ dx$
Trapezoid	4	0.7212	1.7637	1.7342	0.6056
	7	0.7664	1.1922	1.7223	0.5583
	10	0.7797	1.0448	1.7200	0.5562
	13	0.7858	0.9857	1.7193	0.5542
Simpson	3	0.6496	1.3008	1.4662	0.4037
	7	0.7816	1.0017	1.7183	0.5426
	11	0.7524	0.9338	1.6232	0.4844
	15	0.7922	0.9169	1.7183	0.5528
Gauss-Legendre	4	0.8023	0.8563	1.7183	0.5713
	7	0.8006	0.8985	1.7183	0.5457
	10	0.8003	0.9000	1.7183	0.5538
	13	0.8001	0.9000	1.7183	0.5513
Truth		0.80000	0.90000	1.7183	0.55125

and the composite Simpson's rule is

$$S_n(f) = \frac{h}{3} [f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 4f_{n-1} + f_n] - \frac{h^4(b-a)}{180} f^{(4)}(\xi) \quad (7.1.6)$$

for some $\xi \in [a, b]$. The composite Simpson's rule essentially takes three consecutive x_j nodes, uses the interpolating quadratic function to approximate f , and integrates the interpolating quadratic to approximate the integral over that interval.

Notice that by using a locally quadratic approximation to f we have an error of order h^4 , whereas the locally linear approximation yields the trapezoidal rule which has error of order h^2 . As with any local approximation of smooth functions, higher-order approximations yield asymptotically smaller error.

We illustrate these rules by applying them to integrals with known values. Table 7.1 displays Newton-Cotes approximations for $\int_0^1 x^{1/4} dx$, $\int_1^{10} x^{-2} dx$, $\int_0^1 e^{-x} dx$, and $\int_1^{-1} \max[0, x + 0.05] dx$.

The trapezoid and Simpson rules do fairly well for $\int_0^1 e^x dx$, but thirteen points are needed to get five-digit accuracy for the Simpson rule. Both rules have difficulty with $\int_1^{10} x^{-2} dx$ because the integrand's curvature varies greatly within $[1, 10]$, but both converge with Simpson doing so more rapidly. Similar problems arise with $\int_0^1 x^{1/4} dx$ where the integrand is singular at $x = 0$, making the error bounds infinite and hence useless. Both the trapezoid and Simpson rules have difficulty with the integral in the last column because of the kink at $x = -0.05$. If instead the kink were at $x = 0$ as in $\int_1^{-1} \max(0, x) dx$, then any trapezoid rule with an odd number of points would com-

pute the integral exactly. This, however, would be an accident, and the case in table 7.1 is more typical of real life. In general, Simpson's rule does converge in table 7.1 as predicted by theory, but convergence may be quite poor if too few points are used.

Change of Variables Formula and Infinite Integration Domains

The integrals above were defined over finite domains. We will often want to compute integrals with infinite domains. We can adapt Newton-Cotes rules via the change of variables formula to derive integration formulas for some integrals with infinite domains. Since the resulting integral approximations vary substantially in quality, we develop some ideas concerning good choices for the change of variables.

Suppose that we want to integrate $\int_0^\infty f(x) dx$. First, we need to be sure that such an integral exists. Such improper integrals are defined by

$$\int_0^\infty f(x) dx \equiv \lim_{b \rightarrow \infty} \int_0^b f(x) dx \quad (7.1.7)$$

whenever the limit exists. The limit may fail to exist because of divergence, as in the case of $\int_0^\infty 1 \cdot dx$, or because of oscillations, as in the case of $\int_0^\infty \sin x dx$. The doubly infinite integral is defined by

$$\int_{-\infty}^\infty f(x) dx = \lim_{\substack{a \rightarrow -\infty \\ b \rightarrow \infty}} \int_a^b f(x) dx,$$

where the limit is defined if it does not depend on how a and b diverge to $-\infty$ and ∞ .

If (7.1.7) exists, then $f(x) \rightarrow 0$ as $x \rightarrow \infty$. This indicates that we can approximate (7.1.7) by computing $\int_0^b f(x) dx$ as long as b is large enough. We can similarly use $\int_a^b f(x) dx$ to approximate $\int_{-\infty}^\infty f(x) dx$ if b is large and positive, and a large and negative. These are direct, but slow, ways to approximate integrals with infinite domains.

One way to integrate an improper integral is to transform it into an integral with finite bounds. The following theorem facilitates this.

THEOREM 7.1.2 (Change of variables) If $\phi: R \rightarrow R$ is a monotonically increasing, C^1 function on the (possibly infinite) interval $[a, b]$, then for any integrable $g(x)$ on $[a, b]$,

$$\int_a^b g(y) dy = \int_{\phi^{-1}(a)}^{\phi^{-1}(b)} g(\phi(x))\phi'(x) dx. \quad (7.1.8)$$

Equation (7.1.8) is called the *change of variables formula*, since it converts an integral in the variable y into an equivalent one with variable x where y and x are related

by the nonlinear relation $y = \phi(x)$. When we refer to a function $\phi(x)$ as a “change of variables,” we mean that it satisfies the necessary conditions of theorem 7.1.2.

We can use (7.1.8) to approximate integrals of the form (7.1.7). Suppose that we introduce a change of variables, $x(z)$. If $x(0) = 0$ and $x(1) = \infty$, then theorem 7.1.2 implies

$$\int_0^\infty f(x) dx = \int_0^1 f(x(z)) x'(z) dz. \quad (7.1.9)$$

Once we have the new integral defined on $[0, 1]$ we can use any Newton-Cotes formula on this new, equivalent integral.

There are many changes of variables which can be used. The objective is to come up with a $x(z)$ function such that $\int_0^1 f(x(z)) x'(z) dz$ can be easily and accurately computed. One such map is $x(z) = z/(1-z)$ with derivative $x'(z) = 1/(1-z)^2$, implying that

$$\int_0^\infty f(x) dx = \int_0^1 f\left(\frac{z}{1-z}\right) (1-z)^{-2} dz. \quad (7.1.10)$$

In this form we see that we may have a problem. As $z \rightarrow 1$, the term $(1-z)^{-2}$ diverges. The divergent terms make it unclear if the error bounds of the Newton-Cotes quadrature rules imply that the Newton-Cotes formulas applied to (7.1.10) converge to the true value of (7.1.10). In general, if we choose the $x(z) = z/(1-z)$ transformation to approximate (7.1.7), f must be such that $f(x(z))x'(z)$ has some bounded derivatives.

Even with these considerations in mind, $x(z) = z/(1-z)$ may be a good choice for interesting integrands $f(x)$. Consider the integral $\int_0^\infty e^{-t} t^2 dt$. The transformation $t = z/(1-z)$ results in the integral

$$\int_0^1 e^{-z/(1-z)} \left(\frac{z}{1-z}\right)^2 (1-z)^{-2} dz. \quad (7.1.11)$$

The derivative of the integrand in (7.1.11) is

$$-e^{-z/(1-z)} (1-z)^{-6} z (2z^2 + z - 2), \quad (7.1.12)$$

which converges to 0 as $z \rightarrow 1$. Similar calculations show that all derivatives of the integrand of (7.1.11) are bounded on $[0, 1]$. Hence any Newton-Cotes error bound formula applies.

Another example would be the integral

$$\int_{-\infty}^\infty e^{-x^2} f(x) dx, \quad (7.1.13)$$

which equals $\sqrt{\pi}E\{f(X)\}$ if $X \sim N(0, \frac{1}{2})$. In this case we need a monotonic, smooth transformation $x: (0, 1) \rightarrow (-\infty, \infty)$ such that $x(0) = -\infty$ and $x(1) = \infty$. One candidate for this is $x(z) = \ln(z/(1-z))$. With its derivative $x'(z) = (z(1-z))^{-1}$, it replaces (7.1.13) with

$$\begin{aligned} & \int_0^1 e^{-(\ln(z/(1-z)))^2} f\left(\ln\left(\frac{z}{1-z}\right)\right) \frac{dz}{(1-z)z} \\ &= \int_0^1 \left(\frac{1-z}{z}\right)^{\ln(z/(1-z))} f(\ln(z/(1-z))) \frac{dz}{z(1-z)}. \end{aligned} \quad (7.1.14)$$

Again, as long as f is exponentially bounded in its growth, all the derivatives of the integrand in (7.1.14) are bounded.

Not every transformation is appropriate. For example, the map $x(z) = (\ln z/(1-z))^{1/3}$ also maps $(0, 1)$ onto $(-\infty, \infty)$. However, using it when applying (7.1.8) to (7.1.13) will result in an integral whose integrand has unbounded derivatives.

The change of variable strategy is commonly used to convert a difficult integral into a more manageable one. The best change of variables depends on the context, but the basic objective is always the same: The new form of the integral should be suitable for the application of some integration method.

7.2 Gaussian Formulas

Newton-Cotes formulas use a collection of low-order-polynomial approximations on small intervals to derive piecewise-polynomial approximations to f . Gaussian quadrature instead builds on the orthogonal polynomial approach to functional approximation. All Newton-Cotes rules are of the form

$$\int_a^b f(x) dx \doteq \sum_{i=1}^n \omega_i f(x_i) \quad (7.2.1)$$

for some *quadrature nodes* $x_i \in [a, b]$ and *quadrature weights* ω_i . The key feature of Newton-Cotes formulas is that the x_i points are chosen arbitrarily, usually being the uniformly spaced nodes on $[a, b]$, and the ω_i weights are chosen so that if f is locally a low-degree polynomial then the approximation (7.2.1) will be correct. In contrast, Gaussian formulas are constructed by efficient choices of both the nodes and weights. In general, the Gaussian approach is to find points $\{x_i: i = 1, \dots, n\}$ and weights $\{\omega_i: i = 1, \dots, n\}$ so as to make the approximation (7.2.1) of $\int f$ a “good” one.

In order to accomplish this, we must define what we mean by a “good” quadrature formula. The criterion we use is *exact integration* for a finite-dimensional collection of functions. More specifically, we choose the weights and nodes so that the approximation is exactly correct when f is a low-order polynomial. The remarkable feature of Gaussian quadrature is that it accomplishes this for spaces of degree $2n - 1$ polynomials using only n nodes and n weights.

Furthermore Gaussian quadrature is more general than (7.2.1). For any fixed nonnegative weighting function $w(x)$ Gaussian quadrature creates approximations of the form

$$\int_a^b f(x) w(x) dx \doteq \sum_{i=1}^n \omega_i f(x_i) \quad (7.2.2)$$

for some nodes $x_i \in [a, b]$ and positive weights ω_i , and the approximation (7.2.2) is exact whenever f is a degree $2n - 1$ polynomial. Specifically, given a nonnegative function $w(x)$ and the $2n$ -dimensional family \mathcal{F}_{2n-1} of degree $2n - 1$ polynomials, we can find n points $\{x_i\}_{i=1}^n \subset [a, b]$, and n nonnegative weights $\{\omega_i\}_{i=1}^n$ such that

$$\int_a^b f(x) w(x) dx = \sum_{i=1}^n \omega_i f(x_i) \quad (7.2.3)$$

for all $f \in \mathcal{F}_{2n-1}$. The following theorem summarizes the discussion in Davis and Rabinowitz (1984).

THEOREM 7.2.1 Suppose that $\{\varphi_k(x)\}_{k=0}^\infty$ is an orthonormal family of polynomials with respect to $w(x)$ on $[a, b]$. Furthermore define q_k so that $\varphi_k(x) = q_k x^k + \dots$. Let x_i , $i = 1, \dots, n$ be the n zeros of $\varphi_n(x)$. Then $a < x_1 < x_2 < \dots < x_n < b$, and if $f \in C^{(2n)}[a, b]$, then

$$\int_a^b w(x) f(x) dx = \sum_{i=1}^n \omega_i f(x_i) + \frac{f^{(2n)}(\xi)}{q_n^2 (2n)!}$$

for some $\xi \in [a, b]$, where

$$\omega_i = -\frac{q_{n+1}/q_n}{\varphi'_n(x_i) \varphi_{n+1}(x_i)} > 0.$$

Furthermore the formula $\sum_{i=1}^n \omega_i f(x_i)$ is the unique Gaussian integration formula on n nodes that exactly integrates $\int_a^b f(x) w(x) dx$ for all polynomials in \mathcal{F}_{2n-1} .

We can develop a Gaussian quadrature scheme over any interval $[a, b]$ using any weighting function. A key substantive result in theorem 7.2.1 is that Gaussian qua-

drature uses the zeros of the orthogonal polynomials and that they lie in the interval $[a, b]$. Furthermore the ω_i weights are always positive, avoiding the precision problems of high-order Newton-Cotes formulas. The formulas in theorem 7.2.1 tell us how to compute the necessary nodes and weights. Typically one does not do the computation indicated in theorem 7.2.1. Instead, there are some Gaussian quadrature formulas that are particularly useful, and the values of the nodes and weights are kept in tables.

Gauss-Chebyshev Quadrature

[see my tryouts file in matlab](#)

Integrals of the form $\int_{-1}^1 f(x)(1-x^2)^{-1/2} dx$ have the weighting function $(1-x^2)^{-1/2}$, which is the weighting function defining Chebyshev polynomials. To evaluate such integrals, we use the *Gauss-Chebyshev quadrature formula*, defined by the formula,

$$\int_{-1}^1 f(x)(1-x^2)^{-1/2} dx = \frac{\pi}{n} \sum_{i=1}^n f(x_i) + \frac{\pi}{2^{2n-1}} \frac{f^{(2n)}(\xi)}{(2n)!} \quad (7.2.4)$$

for some $\xi \in [-1, 1]$, where the quadrature nodes are

$$x_i = \cos\left(\frac{2i-1}{2n}\pi\right), \quad i = 1, \dots, n. \quad (7.2.5)$$

The Gauss-Chebyshev rule is particularly easy because of the constant weight, π/n , for each node and the easy formula for the quadrature nodes. We will see below that integrals of this form arise naturally when solving various functional equations.

We generally will not be computing integrals of the form $\int_{-1}^1 f(x)(1-x^2)^{-1/2} dx$; instead, we will often need to compute integrals of the form $\int_a^b f(x) dx$ where the range of integration is $[a, b]$ rather than $[-1, 1]$, and where the weight function, $(1-x^2)^{-1/2}$, is missing in the integrand. To apply Gauss-Chebyshev quadrature, we use the linear change of variables $x = -1 + 2(y-a)/(b-a)$ to convert the range of integration to $[-1, 1]$, and multiply the integrand by $(1-x^2)^{-1/2}/(1-x^2)^{-1/2}$. This identity implies that

$$\int_a^b f(y) dy = \frac{b-a}{2} \int_{-1}^1 f\left(\frac{(x+1)(b-a)}{2} + a\right) \frac{(1-x^2)^{1/2}}{(1-x^2)^{1/2}} dx. \quad (7.2.6)$$

We then use Gauss-Chebyshev quadrature to evaluate the RHS of (7.2.6) producing the approximation

$$\int_a^b f(y) dy \doteq \frac{\pi(b-a)}{2n} \sum_{i=1}^n f\left(\frac{(x_i+1)(b-a)}{2} + a\right) (1-x_i^2)^{1/2}, \quad (7.2.7)$$

where the x_i are the Gauss-Chebyshev quadrature nodes over $[-1, 1]$.

Gauss-Legendre Quadrature

Integrals over $[-1, 1]$ could use the trivial weighting function, $w(x) = 1$, resulting in *Gauss-Legendre quadrature* formula

$$\int_{-1}^1 f(x) dx = \sum_{i=1}^n \omega_i f(x_i) + \frac{2^{2n+1}(n!)^4}{(2n+1)!(2n)!} \cdot \frac{f^{(2n)}(\xi)}{(2n)!} \quad (7.2.8)$$

for some $-1 \leq \xi \leq 1$. The n -point Gauss-Legendre weights, ω_i , and nodes, x_i , are listed in table 7.2 for various n .

A linear change of variables is necessary to apply Gauss-Legendre quadrature to general integrals. In general,

$$\int_a^b f(x) dx \doteq \frac{b-a}{2} \sum_{i=1}^n \omega_i f\left(\frac{(x_i+1)(b-a)}{2} + a\right), \quad (7.2.9)$$

where the ω_i and x_i are the Gauss-Legendre quadrature weights and nodes over $[-1, 1]$.

The Gauss-Legendre formula is typical of the rapid convergence of Gaussian quadrature schemes. Applying Stirling's formula, $n! \doteq e^{-n-1} n^{n+(1/2)} \sqrt{2\pi n}$, to the error term in (7.2.8), we find that the error is bounded above by $\pi 4^{-n} M$, where

Table 7.2
Gauss—Legendre quadrature

N	x_i	ω_i	N	x_i	ω_i
2	0.5773502691	0.1000000000(1)	7	0.9491079123	0.1294849661
3	0.7745966692	0.5555555555		0.7415311855	0.2797053914
	0.0000000000	0.8888888888		0.4058451513	0.3818300505
4	0.8611363115	0.3478548451		0.0000000000	0.4179591836
	0.3399810435	0.6521451548	10	0.9739065285	0.6667134430(-1)
5	0.9061798459	0.2369268850		0.8650633666	0.1494513491
	0.5384693101	0.4786286704		0.6794095682	0.2190863625
	0.0000000000	0.5688888888		0.4333953941	0.2692667193
				0.1488743389	0.2955242247

Source: Stroud and Secrest (1966).

Note: $a(k)$ means $a \times 10^k$. An (x, ω) entry for N means that $\pm x$ are quadrature nodes in the N -point formula, and each gets weight ω .

$$M = \sup_m \left[\max_{-1 \leq x \leq 1} \frac{f^{(m)}(x)}{m!} \right].$$

For many functions, such as analytic functions, M is finite. This bound shows that if M is finite, the convergence of Gauss-Legendre quadrature is of exponential order as the number of quadrature nodes goes to infinity. Since Newton-Cotes formulas are only polynomial in convergence, Gauss-Legendre quadrature is much better when f is C^∞ and its derivatives are tame. The same considerations show that the Gauss-Chebyshev error is also proportional to $4^{-n}M$.

Gauss-Legendre integration can be used to compute discounted sums over finite horizons. For example, suppose that consumption at time t equals $c(t) = 1 + t/5 - 7(t/50)^2$, where $0 \leq t \leq 50$. The discounted utility is $\int_0^{50} e^{-\rho t} u(c(t)) dt$, where $u(c)$ is the utility function and ρ is the pure rate of time preference. Let $\rho = 0.05$ and $u(c) = c^{1+\gamma}/(1+\gamma)$. We can approximate the discounted utility with (7.2.9). Table 7.3 displays the result for these methods.

Gauss-Hermite Quadrature

Gauss-Hermite quadrature arises naturally because Normal random variables are used often in economic problems. To evaluate $\int_{-\infty}^{\infty} f(x) e^{-x^2} dx$ using n points, the Gauss-Hermite quadrature rule uses the weights, ω_i , and nodes, $x_i, i = 1, \dots, n$, indicated in table 7.4, and is defined by

$$\int_{-\infty}^{\infty} f(x) e^{-x^2} dx = \sum_{i=1}^n \omega_i f(x_i) + \frac{n! \sqrt{\pi}}{2^n} \cdot \frac{f^{(2n)}(\xi)}{(2n)!}$$

for some $\xi \in (-\infty, \infty)$.

Gauss-Hermite quadrature will be used in connection with Normal random variables. In particular, if Y is distributed $N(\mu, \sigma^2)$, then

Table 7.3
Errors in computing $-\int_0^{50} e^{-0.05t} \left(1 + \frac{t}{5} - 7\left(\frac{t}{50}\right)^2\right)^{1-\gamma} dt$ with Gauss-Legendre rule

γ :	0.5	1.1	3	10
Truth:	1.24431	0.664537	0.149431	0.0246177
Nodes: 3	5(-3)	2(-3)	3(-2)	2(-2)
5	1(-4)	8(-5)	5(-3)	2(-2)
10	1(-7)	1(-7)	2(-5)	2(-3)
15	1(-10)	2(-10)	9(-8)	4(-5)
20	7(-13)	9(-13)	3(-10)	6(-7)

Table 7.4
Gauss-Hermite quadrature

N	x_i	ω_i	N	x_i	ω_i
2	0.7071067811	0.8862269254	7	0.2651961356(1)	0.9717812450(-3)
3	0.1224744871(1)	0.2954089751		0.1673551628(1)	0.5451558281(-1)
	0.0000000000	0.1181635900(1)		0.8162878828	0.4256072526
4	0.1650680123(1)	0.8131283544(-1)		0.0000000000	0.8102646175
	0.5246476232	0.8049140900	10	0.3436159118(1)	0.7640432855(-5)
5	0.2020182870(1)	0.1995324205(-1)		0.2532731674(1)	0.1343645746(-2)
	0.9585724646	0.3936193231		0.1756683649(1)	0.3387439445(-1)
	0.0000000000	0.9453087204		0.1036610829(1)	0.2401386110
				0.3429013272	0.6108626337

Source: Stroud and Secrest (1966).

Note: $a(k)$ means $a \times 10^k$. An (x, ω) entry for N means that $\pm x$ are quadrature nodes in the N -point formula, and each gets weight ω .

$$E\{f(Y)\} = (2\pi\sigma^2)^{-1/2} \int_{-\infty}^{\infty} f(y) e^{-(y-\mu)^2/2\sigma^2} dy.$$

However, one must remember that to use Gauss-Hermite quadrature to compute such expectations, it is necessary to use the linear change of variables, $x = (y - \mu)/\sqrt{2}\sigma$, and use the identity

$$\int_{-\infty}^{\infty} f(y) e^{-(y-\mu)^2/(2\sigma^2)} dy = \int_{-\infty}^{\infty} f(\sqrt{2}\sigma x + \mu) e^{-x^2} \sqrt{2}\sigma dx. \quad (7.2.10)$$

Hence the general Gauss-Hermite quadrature rule for expectations of functions of a normal random variable is

$$\begin{aligned} E\{f(Y)\} &= (2\pi\sigma^2)^{-1/2} \int_{-\infty}^{\infty} f(y) e^{-(y-\mu)^2/(2\sigma^2)} dy \\ &\doteq \pi^{-1/2} \sum_{i=1}^n \omega_i f(\sqrt{2}\sigma x_i + \mu), \end{aligned} \quad (7.2.11)$$

where the ω_i and x_i are the Gauss-Hermite quadrature weights and nodes over $[-\infty, \infty]$.

Examples of using Gauss-Hermite quadrature to compute expectations arise naturally in portfolio theory. For example, suppose that an investor holds one bond which will be worth 1 in the future and equity whose value is Z , where $\ln Z \sim N(\mu, \sigma^2)$. If he consumes his portfolio at a future date and his future utility is $u(c)$, then his expected utility is

Table 7.5
Errors in computing the certainty equivalent of (7.2.12) with Gauss-Hermite rule

Nodes	γ				
	-0.5	-1.1	-2.0	-5.0	-10.0
2	1(-4)	2(-4)	3(-4)	6(-3)	3(-2)
3	1(-6)	3(-6)	9(-7)	7(-5)	9(-5)
4	2(-8)	7(-8)	4(-7)	7(-6)	1(-4)
7	3(-10)	2(-10)	3(-11)	3(-9)	1(-9)
13	3(-10)	2(-10)	3(-11)	5(-14)	2(-13)

$$U = (2\pi\sigma^2)^{-1/2} \int_{-\infty}^{\infty} u(1 + e^z) e^{-(z-\mu)^2/2\sigma^2} dz \quad (7.2.12)$$

and the certainty equivalent of (7.2.12) is $u^{-1}(U)$.

Table 7.5 displays the errors of various methods applied to (7.2.12) with $\mu = 0.15$ and $\sigma = 0.25$, and $u(c) = c^{1+\gamma}/(1 + \gamma)$ for various values of γ . The error that we display is the difference of the certainty equivalent of the approximation and the certainty equivalent of the true value of (7.2.12). We do this so that the error is expressed in economically meaningful terms. Since the certainty equivalent of (7.2.12) with $\mu = 0.15$ and $\sigma = 0.25$ is $2.34\dots$, the errors in table 7.5 are also roughly half the relative errors.

Gauss-Laguerre Quadrature

Exponentially discounted sums are used often in economic problems. To approximate integrals of the form $I = \int_0^\infty f(x)e^{-x}dx$, we use Gauss-Laguerre quadrature. In this case $w(x) = e^{-x}$, and the appropriate weights, ω_i , and nodes, x_i , $i = 1, \dots, n$, to use in (7.2.2) are listed in table 7.6 for various choices of n . The Gauss-Laguerre formulas are defined by

$$\int_0^\infty f(x)e^{-x} dx = \sum_{i=1}^n \omega_i f(x_i) + (n!)^2 \frac{f^{(2n)}(\xi)}{(2n)!}$$

for some $\xi \in [0, \infty)$.

To compute the more general integral, $\int_a^\infty f(y)e^{-ry}dy$, we must use the linear change of variables $x = r(y - a)$, implying that

$$\int_a^\infty e^{-ry}f(y)dy \doteq \frac{e^{-ra}}{r} \sum_{i=1}^n \omega_i f\left(\frac{x_i}{r} + a\right), \quad (7.2.13)$$

Table 7.6
Gauss-Laguerre quadrature

N	x_i	ω_i	N	x_i	ω_i
2	0.5857864376	0.8535533905	7	0.1930436765	0.4093189517
	0.3414213562(1)	0.1464466094		0.1026664895(1)	0.4218312778
3	0.4157745567	0.7110930099	10	0.2567876744(1)	0.1471263486
	0.2294280360(1)	0.2785177335		0.4900353084(1)	0.2063351446(−1)
	0.6289945082(1)	0.1038925650(−1)		0.8182153444(1)	0.1074010143(−2)
4	0.3225476896	0.6031541043	10	0.1273418029(2)	0.1586546434(−4)
	0.1745761101(1)	0.3574186924		0.1939572786(2)	0.3170315478(−7)
	0.4536620296(1)	0.3888790851(−1)		0.1377934705	0.3084411157
	0.9395070912(1)	0.5392947055(−3)		0.7294545495	0.4011199291
5	0.2635603197	0.5217556105	10	0.1808342901(1)	0.2180682876
	0.1413403059(1)	0.3986668110		0.3401433697(1)	0.6208745609(−1)
	0.3596425771(1)	0.7594244968(−1)		0.5552496140(1)	0.9501516975(−2)
	0.7085810005(1)	0.3611758679(−2)		0.8330152746(1)	0.7530083885(−3)
	0.1264080084(2)	0.2336997238(−4)		0.1184378583(2)	0.2825923349(−4)
				0.1627925783(2)	0.4249313984(−6)
				0.2199658581(2)	0.1839564823(−8)
				0.2992069701(2)	0.9911827219(−12)

Source: Stroud and Secrest (1966).

Note: $a(k)$ means $a \times 10^k$. An (x, ω) entry for N means that x is a quadrature node in the N -point formula, weight ω .

where the ω_i and x_i are the Gauss-Laguerre quadrature weights and nodes over $[0, \infty]$.

Gauss-Laguerre quadrature can be used to compute the present value of infinitely long streams of utility or profits. For example, suppose that a monopolist faces a demand curve $D(p) = p^{-\eta}$, $\eta > 1$, and has unit cost of $m(t)$ at time t . If unit costs change over time, say $m(t) = a + be^{-\lambda t}$, and the interest rate is r , then discounted profits equal

$$\eta \left(\frac{\eta - 1}{\eta} \right)^{\eta-1} \int_0^\infty e^{-rt} m(t)^{1-\eta} dt. \quad (7.2.14)$$

In table 7.7 we display the errors of several rules in computing (7.2.14) with $a = 2$, $b = -1$, and $\eta = 0.8$.

The errors in table 7.7 follow the expected pattern. Since we are using the Laguerre formula, the critical piece of the integrand is $m(t)^{1-\eta}$. Gauss-Laguerre integration implicitly assumes that $m(t)^{1-\eta}$ is a polynomial. When $\lambda = 0.05$, $m(t)$ is nearly constant, but when $\lambda = 0.20$, $m(t)^{1-\eta}$ is not constant and becomes less polynomial-like. Therefore it is not surprising that the errors are much larger when $\lambda = 0.20$.

Table 7.7
Errors in computing (7.2.14) with Gauss-Laguerre rule

	$r = 0.05$ $\lambda = 0.05$	$r = 0.10$ $\lambda = 0.05$	$r = 0.05$ $\lambda = 0.20$
Truth:	49.7472	20.3923	74.4005
Nodes: 4	3(-1)	4(-2)	6(0)
5	7(-3)	7(-4)	3(0)
10	3(-3)	6(-5)	2(-1)
15	6(-5)	3(-7)	6(-2)
20	3(-6)	8(-9)	1(-2)

General Applicability of Gaussian Quadrature

If $f(x)$ is not C^∞ , we can still use Gaussian quadrature. Even when the asymptotic rate of convergence for Gaussian quadrature is no better than the comparable Newton-Cotes formula, experience shows that Gaussian formulas often outperform the alternative Newton-Cotes formula. For general integrable functions, we have theorem 7.2.2, proved in Davis and Rabinowitz (1984).

THEOREM 7.2.2 (Gaussian quadrature convergence) If f is Riemann Integrable on $[a, b]$, the error in the n -point Gauss-Legendre rule applied to $\int_a^b f(x) dx$ goes to 0 as $n \rightarrow \infty$.

We now illustrate Gauss-Legendre integration by applying it to the integrals computed by the Newton-Cotes formulas in table 7.1. We find that the Gauss-Legendre approximations are much better than either the trapezoid or Simpson rules. This is the case even for $\int_{-1}^1 \max(x + .05, 0) dx$ where the integrand has a kink at $x = -0.05$ and is not even C^1 . The four-point formula computes $\int_0^1 e^x dx$ with five-digit accuracy. This is not surprising, for e^x is a very well-behaved integrand.

Interpolatory Rules

The formulas investigated above used values of $f(x)$, the integrand. We next investigate rules that also use derivatives of $f(x)$ to approximate $\int_a^b f(x) dx$. These rules, known as *interpolatory quadrature rules*, take the form

$$\int_a^b f(x) w(x) dx \doteq \sum_{i=1}^n \sum_{j=0}^m \omega_{ij} f^{(j)}(x_i), \quad (7.2.15)$$

where the x_i are the nodes and the ω_{ij} are the weights for the various derivatives. This

generalizes the standard Gaussian form where $m = 0$ implicitly. We generally choose the nodes and weights so that the rule correctly integrates low-order polynomials. A formula of form (7.2.15) is a *degree l interpolatory rule* if it exactly integrates all polynomials of degree l and less. To determine these rules, we need to solve a system of nonlinear equations that expresses these exactness requirements.

For example, suppose that we want to derive a degree 11 interpolatory rule for $[-1, 1]$ which uses three points. We examine a formula of the form

$$\begin{aligned}\hat{I}(f) \equiv & \omega_0 f(0) + \omega_1(f(-x) + f(x)) \\ & + \omega_2(f'(-x) - f'(x)) \\ & + \omega_3(f''(-x) + f''(x)) \\ & + \omega_4 f''(0),\end{aligned}\tag{7.2.16}$$

where the six unknown parameters are the nodal distance, x , and the five weights, $\omega_i, i = 0, \dots, 4$. Note that $\hat{I}(x^m) = 0 = \int_{-1}^1 x^m dx$ for any odd m and for any values of the ω parameters. To determine the six parameters in (7.2.16), we impose the six conditions $\hat{I}(x^m) = I(x^m)$ for $m = 0, 2, 4, 6, 8, 10$, resulting in six nonlinear equations in six unknowns. Fortunately there is a real-valued solution, which is

$$\begin{aligned}x &= 0.81444, \quad \omega_0 = 0.93337, \quad \omega_1 = 0.53332, \\ \omega_2 &= -0.03116, \quad \omega_3 = 0.00411, \quad \omega_4 = 0.02213.\end{aligned}\tag{7.2.17}$$

To test the quality of the resulting formula, we apply it to three functions. Table 7.8 compares the error of our interpolatory formula with the errors of the five point (degree 9) and six-point (degree 11) Legendre formulas applied to integrals of $\cos 3x$, e^{3x} , and $(x + 1.1)^{1/2}$. We see that the degree 11 interpolatory rule defined by (7.2.16) and (7.2.17) is better than the degree 9 Legendre rule and close to the degree 11 Legendre rule, even though the interpolatory rule uses f at only 3 points.

At first, derivative rules may not appear to offer any advantages, since computing derivatives often involve substantial extra computation. In fact, if one uses finite dif-

Table 7.8
Interpolatory rule errors

$f(x)$	$\cos 3x$	e^{3x}	$(x + 1.1)^{1/2}$
Interpolatory rule	1.9(-6)	2.6(-6)	1.2(-4)
Five-point Legendre rule	4.0(-5)	5.7(-5)	3.6(-4)
Six-point Legendre rule	7.0(-7)	9.6(-7)	4.7(-5)

ferences to evaluate derivatives, then interpolatory quadrature rules are not advantageous. However, computing a derivative may be much cheaper than computing a function value, particularly, if we can use automatic differentiation. Interpolatory rules are not used much, but as automatic differentiation software becomes more common, these rules may gain in popularity.

7.3 Singular Integrals

We have focused on integrals with bounded integrands and used error formulas that assume that the integrand's derivatives are also bounded. In this section we consider $\int_0^1 f(x) dx$ where either f or f' is unbounded on $[0, 1]$. Without loss of generality we can assume that the singular point of f is $x = 1$; otherwise, if there is a singularity at $a \in (0, 1)$, we separately consider terms in the equivalent sum $\int_0^a f(x) dx + \int_a^1 f(x) dx$. In all cases we assume that $\int_0^1 f(x) dx$ exists and is finite, facts that one must establish independent of any numerical computations.

Ignore the Singularities

We can often just use the standard Newton-Cotes or Gaussian rules. The following theorem makes a precise statement concerning this approach.

THEOREM 7.3.1 If there is a continuous monotonically increasing $g: [0, 1] \rightarrow \mathbb{R}$ such that $\int_0^1 g(x) dx < \infty$ and $|f(x)| \leq g(x)$ on $[0, 1]$, then Newton-Cotes rules (in which we set $f(1) = 0$ to avoid the singular value) and the Gauss-Legendre quadrature rule converge to $\int_0^1 f(x) dx$ as $n \rightarrow \infty$.

Proof See Davis and Rabinowitz (1984, pp. 180–82), and citations therein. ■

Our examples in table 7.1 included the integral $\int_0^1 x^{1/4} dx$ which is singular at $x = 0$. The trapezoid and Simpson rules do fairly well, but not as well as for non-singular integrals. The Gauss-Legendre approximations are very good even for handling the singularity in $x^{1/4}$ at 0 and the high curvature of x^{-2} over $[1, 10]$.

The major problem with using standard methods is that the convergence is generally much slower. In fact the standard error formulas are worthless, since $\|f^{(k)}\|_\infty$ is infinite for $k \geq 1$ for singular $f(x)$. Hence, if we want to have useful error bounds, we must use methods tailored to deal with the singularity.

Standard Methods Adapted to the Singularity

We can take standard methods of interpolation and orthogonal polynomials and adapt them to a particular singularity. Suppose that we have an integral of the form

$$\int_0^1 f(x)x^{-1/2} dx. \quad (7.3.1)$$

If $0 < x_0 < x_1 < \dots < x_n \leq 1$ are the $n+1$ fixed quadrature nodes, we can derive rules that will exactly integrate any degree n polynomial. For $n=1$, $x_0 = 1/4$, and $x_1 = 3/4$, we are looking for weights w_1, w_2 such that

$$w_1 + w_2 = \int_0^1 x^{-1/2} \cdot 1 dx = 2, \quad (7.3.2)$$

$$\frac{1}{4}w_1 + \frac{3}{4}w_2 = \int_0^1 x^{-1/2} \cdot x dx = \frac{2}{3}.$$

This linear system has the solution $w_1 = 5/3$ and $w_2 = 1/3$, which implies the rule

$$\int_0^1 f(x)x^{-1/2} dx \doteq \frac{5}{3}f\left(\frac{1}{4}\right) + \frac{1}{3}f\left(\frac{3}{4}\right). \quad (7.3.3)$$

In general, this approach will lead to extensions of Newton-Cotes to this singularity. Similarly one could view $x^{-1/2}$ as a weighting function over $[0, 1]$ and construct an orthogonal polynomial family and corresponding Gaussian quadrature rules. This approach has been applied to several kinds of singularities; see the references in Davis and Rabinowitz (1984). The weakness of this strategy is that one must derive the quadrature formulas anew for each different singularity. However, this would be a fixed, one-time cost, well worth the effort if it reduced the marginal cost of evaluating a common singular integral.

Change of Variables

A singularity can sometimes be eliminated by a change of variable. For example,

$$\int_0^1 x^{-1/p} f(x) dx = p \int_0^1 y^{p-2} f(y^p) dy \quad (7.3.4)$$

eliminates the singularity at $x=0$ with the change of variables $y^p = x$. Note that the second integral has no singularity for $p \geq 2$ as long as f has no singularities. Also the change of variables $y^q = x$ implies that

$$\int_0^1 x^{p/q} f(x) dx = q \int_0^1 y^{p+q-1} f(y^q) dy, \quad (7.3.5)$$

which eliminates the singular contribution of $x^{p/q}$ at zero if $p+q-1 > 0$.

One example of this approach is the computation of Chebyshev coefficients. It is convenient to use the change of variables $x = \cos \theta$, which leads to the identity

$$\int_{-1}^1 \frac{f(x)T_n(x)}{\sqrt{1-x^2}} dx = \int_0^\pi f(\cos \theta) \cos n\theta d\theta. \quad (7.3.6)$$

The first integral's integrand diverges at both $x = -1$ and $x = 1$. The second integral in (7.3.6) is much better behaved since it has no singularities, and this is the preferred way to compute Chebyshev coefficients.

These devices will often deal satisfactorily with singularities in integrals. There are no general rules for singular integrals; one should always be looking for clever transformations that eliminate or tame the singularity.

7.4 Adaptive Quadrature

The fact that a quadrature formula will converge for large N is comforting, but in actual problems the analyst must decide on how many points to use. It is difficult to say how many points suffice to get a good approximation before one begins to compute an integral. For example, if f is monotonically increasing and its high-order derivatives are small, Gaussian rules with few points will likely suffice. If instead f is oscillatory, many points will be needed. For many complicated functions, one will not know whether f is easy or difficult to integrate. Unless one knows f to be well-behaved, one must be prepared to deal with difficult integrands.

To deal with poorly behaved integrands or integrands about which we know little, we use adaptive rules. An adaptive rule takes a simple rule and progressively refines it until the estimates don't change "much." Simple examples include the trapezoid and Simpson rules where we first use an interval size h , then $h/2$, then $h/4$, and so forth, proceeding until the approximations appear to be converging. Such rules are robust and recommended if one does not know much about the shape of f . Unfortunately, these rules are conservative and relatively costly to use. The reader who feels he needs such procedures should consult Davis and Rabinowitz (1984). Also QUADPACK, described in Piessens et al. (1983), is an integration package that contains many such routines. However, one should first try to see if Newton-Cotes or Gaussian rules are applicable because the savings can be substantial.

7.5 Multidimensional Quadrature

If we only had to deal with one-dimensional integrals, the choice of method would not be of great importance. However, many interesting problems in economics involve the integration of multidimensional functions. Unfortunately, many economists appear to believe that Monte Carlo integration (a method we examine in the

next chapter) is the only practical way to do multidimensional integration. In this section and in chapter 9, we will consider procedures for multidimensional integration, many of which often dominate Monte Carlo integration for integrals arising in economics.

One approach is to directly extend the one-dimensional methods via product rules. This approach yields formulas that require a rapidly growing number of function evaluations as the dimension increases. Fortunately many of the ideas used in one-dimensional quadrature can be applied to multidimensional integrals without incurring the “curse of dimensionality”; we will describe those methods in the sections below. For high-dimension problems, we often switch to sampling methods, which will be discussed in the next two chapters.

Product Rules

Suppose that we want to compute $\int_{[-1,1]^d} f(x) dx$. We can form product rules based on any one-dimensional quadrature rule. Let $x_i^l, \omega_i^l, i = 1, \dots, m$, be one-dimensional quadrature points and weights in dimension l from a Newton-Cotes rule or the Gauss-Legendre rule. The *product rule* approximates the integral $\int_{[-1,1]^d} f(x) dx$ with the sum

$$\sum_{i_1=1}^m \cdots \sum_{i_d=1}^m \omega_{i_1}^1 \omega_{i_2}^2 \cdots \omega_{i_d}^d f(x_{i_1}^1, x_{i_2}^2, \dots, x_{i_d}^d). \quad (7.5.1)$$

We can also use this idea for any Gaussian quadrature formula. If $w^l(x)$ is a scalar weighting function in dimension l , let

$$W(x) \equiv W(x_1, \dots, x_d) = \prod_{l=1}^d w^l(x_l) \quad (7.5.2)$$

be a d -dimensional weighting function. Then the integral $\int_{[-1,1]^d} f(x) W(x) dx$ can be computed using (7.5.1) where the $\omega_{i_l}^l$ and $x_{i_l}^l$ choices are the weights and nodes for the one-dimensional formula for the weighting function $w^l(x)$.

A difficulty of this approach is that the number of functional evaluations is m^d for a d -dimensional problem if we take m points in each direction. Notice that as d increases, the amount of work rises exponentially—the “curse of dimensionality.” This exponential growth in cost restricts product rules to low-dimension integrals. Product versions of Newton-Cotes formulas are particularly limited, for more points are required in each dimension with Newton-Cotes than Gaussian quadrature to attain a target level of accuracy.

Monomial Formulas: A Nonproduct Approach

The reason for the exponential growth rate for product formulas is that they are exactly integrating tensor product bases of function. We next use the Gaussian idea of finding a rule that exactly integrates a family of polynomials, but we focus on a smaller family, such as the set of complete polynomials, to determine the rule's nodes and weights. This approach yields *monomial* rules.

Consider the case of degree l monomials over $D \subset \mathbb{R}^d$. A monomial rule uses N points $x^i \in D$ and associated weights, ω_i , so that

$$\sum_{i=1}^N \omega_i p(x^i) = \int_D p(x) dx \quad (7.5.3)$$

for each polynomial $p(x)$ of total degree l ; recall that \mathcal{P}_l was defined in chapter 6 to be the set of such polynomials. Any such formula is said to be *complete for degree l* , or, a *degree l formula* for short. For the case $l = 2$, this implies the following system of monomial equations:

$$\begin{aligned} \sum_{i=1}^N \omega_i &= \int_D 1 \cdot dx, \\ \sum_{i=1}^N \omega_i x_j^i &= \int_D x_j dx, \quad j = 1, \dots, d, \\ \sum_{i=1}^N \omega_i x_j^i x_k^i &= \int_D x_j x_k dx, \quad j, k = 1, \dots, d. \end{aligned} \quad (7.5.4)$$

While the system (7.5.4) is easy to express, the algebra associated with its analysis is more problematic than in the one-dimensional case. Since there are $1 + d + \frac{1}{2}d(d + 1)$ basis elements of \mathcal{P}_2 , we have $1 + d + \frac{1}{2}d(d + 1)$ equations in (7.5.4). The unknowns are the N weights ω_i and the N nodes x^i each with d components, yielding a total of $(d + 1)N$ unknowns. If $(d + 1)N$ exceeds $1 + d + \frac{1}{2}d(d + 1)$, we would expect there to be solutions, but given the nonlinearity in our system, that is not guaranteed. Furthermore there may be many solutions. As we increase l , the problems only increase.

Solutions are known for some simple cases. Recall that $e^j \equiv (0, \dots, 1, \dots, 0)$ where the “1” appears in column j . One rule that uses $2d$ points and exactly integrates all elements of \mathcal{P}_3 over $[-1, 1]^d$ is

$$\int_{[-1,1]^d} f \doteq \omega \sum_{i=1}^d (f(ue^i) + f(-ue^i)), \quad (7.5.5)$$

where

$$u = \left(\frac{d}{3}\right)^{1/2}, \quad \omega = \frac{2^{d-1}}{d}.$$

For \mathcal{P}_5 the following scheme works:

$$\begin{aligned} \int_{[-1,1]^d} f &\doteq \omega_1 f(0) + \omega_2 \sum_{i=1}^d (f(ue^i) + f(-ue^i)) \\ &+ \omega_3 \sum_{\substack{1 \leq i < d, \\ i < j \leq d}} (f(u(e^i \pm e^j)) + f(-u(e^i \pm e^j))), \end{aligned} \quad (7.5.6)$$

where

$$\omega_1 = 2^d(25d^2 - 115d + 162), \quad \omega_2 = 2^d(70 - 25d),$$

$$\omega_3 = \frac{25}{324}2^d, \quad u = \left(\frac{3}{5}\right)^{1/2}.$$

Unfortunately, these weights get large and are of mixed sign for $d \geq 3$, both undesirable features for quadrature formulas. This is typical of simple monomial rules.

Another scheme for \mathcal{P}_5 on $[-1, 1]^2$ is the Radon seven-point formula:

$$\begin{aligned} \int_{[-1,1]^2} f(x, y) dx dy &= \omega_1 f(P_1) \\ &+ \omega_2 [f(P_2) + f(P_3) + f(P_4) + f(P_5)] \\ &+ \omega_3 [f(P_6) + f(P_7)], \end{aligned} \quad (7.5.7)$$

where

$$P_1 = (0, 0),$$

$$P_2, P_3, P_4, P_5 = (\pm s, \pm t), \quad s = \sqrt{\frac{1}{3}}, \quad t = \sqrt{\frac{3}{5}},$$

$$P_6, P_7 = (\pm r, 0), \quad r = \sqrt{\frac{14}{15}},$$

$$\omega_1 = \frac{8}{7},$$

$$\omega_2 = \frac{5}{9},$$

$$\omega_3 = \frac{20}{63}.$$

This rule exactly integrates all elements of \mathcal{P}_5 with positive weights.

Despite the weaknesses of these examples, there is theoretical evidence that the monomial approach has potential. Tchakaloff's theorem and its extension by Mysovskikh tell us that there are good quadrature formulas.

THEOREM 7.5.1 (Tchakaloff) Let B be a closed, bounded set in R^2 with positive area. Then there exists at least one collection of $N \leq \frac{1}{2}(m+1)(m+2)$ points in B , P_1, \dots, P_N , and N positive weights, $\omega_1, \dots, \omega_N$, such that

$$\int_B f(x, y) dx dy = \sum_{i=1}^N \omega_i f(P_i)$$

for all f polynomial in x, y of total degree $\leq m$.

In the following theorem a *multi-index* α is a vector of nonnegative integers $(\alpha_1, \dots, \alpha_d)$; the norm of the multi-index α , $|\alpha|$, equals the sum of the components, $\sum_{i=1}^d \alpha_i$. For $x \in R^d$ the term x^α is shorthand for the monomial $x_1^{\alpha_1} \cdots x_d^{\alpha_d}$.

THEOREM 7.5.2 (Mysovskikh) Let $w(x)$ be a nonnegative weighting function on $D \subset R^d$ such that each moment

$$\int_D w(x) x_1^{i_1} \cdots x_d^{i_d} dx_1 \cdots dx_d$$

exists for $i_1, \dots, i_d \geq 0$, $i_1 + \cdots + i_d \leq m$. Then, for some $N \leq (m+d)!/(m!d!)$, there exists N positive weights, ω_i , and N nodes, x^i , such that for each multi-index $|\alpha| \leq m$,

$$\int_D w(x) x^\alpha dx = \sum_{i=1}^N \omega_i (x^i)^\alpha.$$

The problem with the Tchakaloff and Mysovskikh theorems is that they are purely existential, giving us no efficient procedure to find such points and weights. However, we should not be deterred by their nonconstructive nature. The Tchakaloff and

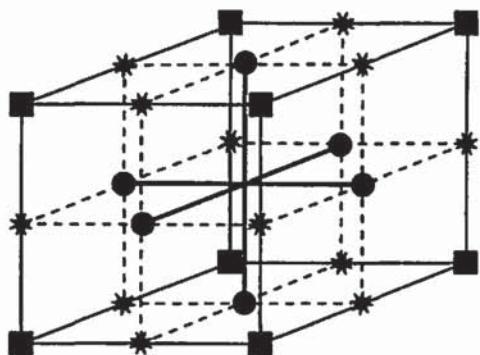


Figure 7.2
Nodes for monomial rules over $[-1, 1]^3$

Mysovskikh theorems essentially reduce the problem of finding good quadrature formulas to solving a set of nonlinear equations, (7.5.3). While that system may have multiple solutions over any particular domain, and some solutions may use negative weights, these theorems tell us that good solutions exist and will (presumably) be found if we search through all the solutions to the monomial equations. If there are better formulas than guaranteed by Tchakaloff and Mysovskikh, we will also find them. For example, Radon's seven-point formula is much better than what is guaranteed by Tchakaloff's theorem.

If one is willing to impose conditions on the points and weights, then one can sometimes derive monomial formulas in a straightforward fashion. Stroud (1971) examines symmetric domains and imposes symmetry restrictions that considerably simplify the monomial equations. There is no assurance that such solutions exist, but this approach often yields good formulas.

The first step in constructing monomial formulas is choosing a set of quadrature nodes. Figure 7.2 illustrates some alternatives for the case of the three-dimensional cube. We often choose a set of points with some symmetry properties. We would usually include the center of the cube. A seven-point formula may use the center plus the six points that are the centers of the cube's faces, the points which are circles in figure 7.2. A nine-point formula might use the center and the eight vertices, the square points in figure 7.2. A twelve-point formula might use the centers of the cube's edges, the stars in figure 7.2. Other formulas might use some combinations of the squares, circles, and stars. Of course, if we use all 27 points (including the center) in figure 7.2, then we would have the collection of points used by the three-dimensional product formula with three points in each dimension. Once we have chosen a collection of points, the weights are chosen so that some important collection of functions is integrated exactly.

We will review a few examples now. They are taken from Stroud (1971) which contains a large collection of such formulas. We first consider integrals of the form

$$\int_{R^d} f(x) e^{-\sum_{i=1}^d x_i^2} dx \quad (7.5.8)$$

which arise naturally when computing expectations of functions of multivariate normal random variables. The following is a degree 3 formula using $2d$ points for the integral (7.5.8):

$$\frac{V}{2d} \sum_1^d f(\pm re^i), \quad r = \sqrt{\frac{d}{2}}, \quad V = \pi^{d/2}. \quad (7.5.9)$$

The following formula is also degree 3 but uses 2^d points:

$$\frac{V}{2^d} \sum f(\pm re^1, \dots, \pm re^d), \quad r = \sqrt{\frac{1}{2}}, \quad V = \pi^{d/2}. \quad (7.5.10)$$

The following is a degree 5 rule using $2d^2 + 1$ points:

$$Af(0, \dots, 0) + B \sum_{i=1}^d (f(re^i) + f(-re^i)) + D \sum_{i=1}^{d-1} \sum_{j=i+1}^d f(\pm se^i \pm se^j), \quad (7.5.11)$$

where

$$r = \sqrt{1 + \frac{1}{2}d}, \quad s = \sqrt{\frac{1}{2} + \frac{1}{4}d}, \quad V = \pi^{d/2},$$

$$A = \frac{2}{d+2} V, \quad B = \frac{4-d}{2(d+2)^2} V, \quad D = \frac{V}{(d+2)^2}.$$

For $\int_{[-1,1]^d} f(x)$ we have the rule

$$Af(0, \dots, 0) + B \sum_{i=1}^d (f(re^i) + f(-re^i)) + D \sum_{x \in C} f(x), \quad (7.5.12)$$

where $C = \{x \mid \forall i (x_i = \pm 1)\}$ and

$$r = \sqrt{\frac{2}{5}}, \quad V = 2^d, \quad A = \frac{8-5d}{9} V, \quad B = \frac{5}{18} V, \quad D = \frac{1}{9}.$$

This is a degree 5 rule using $2^d + 2d + 1$ points.

These formulas are just a few examples of what is available. Stroud (1971) displays many other formulas and discusses methods for generating such formulas.

Multivariate Changes of Variables

All the formulas above assume that integrals are in some standard form. Most integrals that arise in economic contexts are not in these canonical forms. We must transform them into integrals in the canonical forms with a change of variables transformation. The key theorem is the following one:

THEOREM 7.5.3 (Change of variables in R^n) Assume that $\Phi: X \subset R^n \rightarrow R^n$ is C^1 and that there is a C^1 inverse map $\Phi^{-1}: Y \subset R^n \rightarrow X$ where $Y = \Phi(X)$ and Y and X are open. If f is integrable on X , then

$$\int_Y f(y) dy = \int_X f(\Phi(x)) |\det J(\Phi)(x)| dx, \quad (7.5.13)$$

where $J(\Phi)(x)$ is the Jacobian of Φ at x .

Change of variable formulas have many uses. For example, they can be used to convert nonstandard integration domains into the canonical domains used by the methods above. For example, consider the integral

$$\int_a^b \int_c^x f(x, y) dy dx. \quad (7.5.14)$$

The domain of integration in (7.5.14) is a trapezoid if $c < a$ or $c > b$, or a “bowtie” region if $a < c$. The change of variable $y = c + (x - c)(z - c)/(b - c)$ converts (7.5.14) into

$$\int_a^b \int_c^b f\left(x, c + \frac{(x - c)(z - c)}{(b - c)}\right) \frac{x - c}{b - c} dz dx,$$

which has a rectangular domain to which we can apply standard formulas.

A particularly important application of theorem 7.5.3 arises in connection with multivariate normal random variables. To evaluate a multivariate normal expectation, we use the Cholesky decomposition of the variance-covariance matrix together with a linear change of variables. Specifically, suppose that $Y \sim N(\mu, \Sigma)$, and we want to compute $E\{f(Y)\}$. First, we note that

$$E\{f(Y)\} = |\Sigma|^{-1/2} (2\pi)^{-n/2} \int_{R^n} f(y) \exp[-\frac{1}{2}(y - \mu)^\top \Sigma^{-1}(y - \mu)] dy$$

Second, Σ has a Cholesky decomposition since Σ is symmetric positive definite. Let $\Sigma = \Omega\Omega^\top$ be that decomposition. Since $\Sigma^{-1} = (\Omega^{-1})^\top\Omega^{-1}$, the linear change of variables $x = \Omega^{-1}(y - \mu)/\sqrt{2}$ implies that $y = \sqrt{2}\Omega x + \mu$ and that

$$\begin{aligned} & \int_{R^n} f(y) e^{-1/2(y-\mu)^\top \Sigma^{-1}(y-\mu)} dy \\ &= \int_{R^n} f(\sqrt{2}\Omega x + \mu) e^{-\sum_{i=1}^n x_i^2} |\det \Omega| 2^{n/2} dx \end{aligned} \quad (7.5.15)$$

which, is in the form required by (7.5.8).

7.6 Example: Portfolio Problems

In chapter 4 we introduced the general portfolio choice problem. In the examples there we assumed a discrete distribution. We now move to problems with a continuous distribution and use quadrature rules to approximate the integrals that arise when we compute the objective and its derivatives. The portfolio optimization problem with neither transaction costs nor shorting constraints is

$$\begin{aligned} & \max_{\omega_i} E \left\{ u \left(\sum_{i=1}^n \omega_i Z_i \right) \right\} \\ \text{s.t. } & \sum_{i=1}^n p_i (\omega_i - e_i) = 0 \end{aligned} \quad (7.6.1)$$

(see section 4.12). Throughout this example we assume the CRRA utility function $u(c) = c^{1+\gamma}/(1+\gamma)$. This example shows both the power and pitfalls of Gaussian quadrature.

We investigate the individual investor problem. To simplify matters, we assume only two assets. Suppose that each asset costs \$1 per unit. Asset 1 is a safe asset with future value R , and the log of the future value of asset 2 is $Z_2 \sim N(\mu, \sigma^2)$. We assume that the initial endowment is $e_1 = W$ and $e_2 = 0$. If the utility function is $u(c)$, we use the substitution $\omega_1 = W - \omega_2$ and let $\omega = \omega_2$ be the choice variable. Then $c = R(W - \omega) + \omega e^Z$ and the problem reduces to the unconstrained problem

$$\max_{\omega} E\{u(R(W - \omega) + \omega e^Z)\}.$$

This objective has the derivatives $U'(\omega) = E\{u'(c)(e^Z - R)\}$ and $U''(\omega) = E\{u''(c)(e^Z - R)^2\}$, where $U(\omega) \equiv E\{u(c)\}$. The objective and its derivatives are integrals that usually must be numerically approximated.

Table 7.9
Portfolio solutions

γ	Rule				
	GH3	GH4	GH5	GH6	GH7
-3	2.75737	2.57563	2.54240	2.53343	2.53035
-4	2.02361	1.95064	1.94533	1.94503	1.94502
-5	1.59369	1.56166	1.56200	1.56210	1.56209

Note: GH n denotes Gauss-Hermite rule with n nodes.

We first compute the value of various portfolios, $U(\omega)$. Without loss of generality, we let $W = 2$. We examine five cases of the utility function, $\gamma = -0.5, -1.1, -2, -5$, and -10 . We assume that $(\mu, \sigma) = (0.4, 0.3)$. Recall table 7.5. The integrand examined there, (2.12), is the value of the portfolio here with one unit of each asset with $R = 1$ and $\omega = 1$ as computed by various Gauss-Hermite rules. Table 7.5 reports the certainty equivalent of expected utility and the integration errors, since we want to report these quantities in economically meaningful units. The errors in table 7.5 are small even when we use only a few points.

We next compute the optimal portfolios. We use Newton's optimization method to maximize $U(\omega)$ and use Gauss-Hermite integration to evaluate $U(\omega)$. Table 7.9 indicates the optimal choice of ω for various integration rules and various values for γ . One important detail is that we need to use enough points so that we capture the problem correctly. For example, the two-point rule for $R = 1.1$, $\mu = 0.4$, $\sigma = 0.3$, approximates $U(\omega)$ with

$$\frac{0.9[(2.2 + 0.005\omega)^{1+\gamma} + (2.2 + 0.9\omega)^{1+\gamma}]}{1 + \gamma},$$

which is a monotonically increasing function of ω , whereas we know that $U(\omega)$ is concave and eventually decreasing. The three-point rule produces the approximation

$$\frac{0.3(2.2 - 0.2\omega)^{1+\gamma} + 1.2(2.2 + 0.4\omega)^{1+\gamma} + 0.3(2.2 + 1.4\omega)^{1+\gamma}}{1 + \gamma},$$

which does have a maximum. If we are to use quadrature to maximize $U(\omega)$, we clearly need to choose enough integration points so that the objective's approximation has a maximum value.

Table 7.9 displays the maximizing value of ω when we approximate $U(\omega)$ with various Gauss-Hermite formulas. We assume that $\gamma = -3, -4, -5$. The results for $\gamma = -3$ are disturbing, since with $W = 2$ the choice of any $\omega > 2$ implies a positive

Table 7.10
Portfolio Optimization

Iterate	Rule		
	GH3	GH4	GH7
1	0.46001	0.45908	0.45905
2	1.05932	1.05730	1.05737
3	1.51209	1.51197	1.51252
4	1.57065	1.56270	1.56248
5	1.56031	1.56209	1.56209
9	1.56211	1.56209	1.56209

Note: GH n denotes Gauss-Hermite rule with n nodes.

probability that $R(W - \omega) + \omega e^Z$ will be negative, an impermissible case for any CRAA utility function. The reason for the perverse results in table 7.9 is that the Gauss-Hermite rules used did not include enough points to properly treat the extreme cases where e^Z is nearly zero. For $\gamma = -3$ it appears that we would need many points to avoid the problem. This example shows how one must be careful when using quadrature to approximate integrals. These problems are not difficult to avoid; here the problem is that some choices of ω cause singularities to arise, a fact that we should see before attempting to solve this problem. The other solutions are not affected by these considerations because they do not cause any singularities in either the expected utility or expected marginal utility integrals.

We next consider issues associated with the approximation of gradients and Hessians in the portfolio problem. In chapter 4 we discussed the possibility of using low-quality gradients and Hessians to improve performance. We can examine this issue in the portfolio problem, since the objective and any of its derivatives must be computed in an approximate fashion. In table 7.10 we indicate the Newton iterates for the $\gamma = -5$ case using a variety of quadrature rules for the first and second derivatives. Note that the convergence is rapid even when we use the three point formula for U'' (we use GH7 for U'). This one-dimensional example shows that we can save considerably with less precise derivatives. This phenomenon is of even greater value in higher dimensions.

7.7 Numerical Differentiation

Numerical differentiation is used frequently in numerical problems. We use finite-difference approximations of gradients, Hessians, and Jacobians in optimization and

nonlinear equation problems. In fact, as emphasized above, these objects are frequently computed numerically, since analytic computation of derivatives is difficult and time-consuming for the programmer. Numerical derivatives are also important in some differential equation methods. In this section we examine the general approach to developing numerical derivative formulas.

The derivative is defined by

$$f'(x) = \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon}.$$

This suggests the formula

$$f'(x) \doteq \frac{f(x + h) - f(x)}{h} \quad (7.7.1)$$

to approximate $f'(x)$. How big should h be? It would seem that h should be as small as possible. But here the subtraction problem is obviously problematic for small h .

Suppose that f is computed with accuracy ε ; that is, if $\hat{f}(x)$ is the computed value of $f(x)$, then $|f(x) - \hat{f}(x)| \leq \varepsilon$. If we apply (7.7.1) to estimate $f'(x)$ but use the computed function \hat{f} , then $D(h) = (\hat{f}(x + h) - \hat{f}(x))/h$ is the computed approximation to $f'(x)$, and the error is

$$\left| D(h) - \frac{f(x + h) - f(x)}{h} \right| \leq \frac{2\varepsilon}{h}.$$

But by Taylor's theorem,

$$f'(x) = \frac{f(x + h) - f(x)}{h} - \frac{h}{2} f''(\xi)$$

for some $\xi \in [x, x + h]$. Suppose that $M_2 > 0$ is an upper bound on $|f''|$ near x . Then the error in $D(h)$ is bounded above:

$$|f'(x) - D(h)| \leq \frac{2\varepsilon}{h} + \frac{h}{2} M_2. \quad (7.7.2)$$

This upper bound is minimized by choosing $h = h^*$ where

$$h^* = 2 \sqrt{\frac{\varepsilon}{M_2}} \quad (7.7.3)$$

and the upper bound equals $2\sqrt{\varepsilon M_2}$.

Two-Sided Differences

Sometimes the one-sided difference will not be accurate enough for our purposes. Our upper bound formula says that for $M_2 \sim 1$ it gives only $d/2$ significant digits for the derivative when we know the function values to d digits. This may not be good enough for some purposes. Since we generally don't want to compute analytic derivatives, we seek better finite-difference formulas.

A better approximation is the two-sided formula

$$f'(x) \doteq \frac{f(x+h) - f(x-h)}{2h}, \quad (7.7.4)$$

which differs from $f'(x)$ by $(h^2/6)f'''(\xi)$ for some $\xi \in [x-h, x+h]$. The round-off error of the approximation error is ε/h , resulting in an upper bound for the error of $(M_3 h^2)/6 + \varepsilon/h$ if M_3 is an upper bound on $|f'''|$ near x . In this case the optimal h is $h^* = (3\varepsilon/M_3)^{1/3}$ where the upper bound is $2\varepsilon^{2/3} M_3^{1/3} 9^{1/3}$. Using the two-sided formula, we reduce the error from order $\varepsilon^{1/2}$ to order $\varepsilon^{2/3}$. For example, on a twelve-digit machine, the one-sided formula yields (roughly) six-digit accuracy and the two-sided formula yields eight-digit accuracy.

General Three-Point Formulas

We next discuss the general procedure for generating three-point formulas for f' and f'' . Suppose that we know $f(x)$ at x_1, x_2 , and x_3 , and we want to approximate $f'(x_1)$ and $f''(x_1)$ with quadratic accuracy. More specifically, we want to find constants a, b , and c such that

$$af(x_1) + bf(x_2) + cf(x_3) = f'(x_1) + o((x_1 - x_2)^2 + (x_1 - x_3)^2). \quad (7.7.5)$$

By Taylor's theorem, for $i = 2, 3$,

$$f(x_i) = f(x_1) + f'(x_1)(x_i - x_1) + \frac{f''(x_1)(x_i - x_1)^2}{2} + \frac{(x_i - x_1)^3 f'''(\xi_i)}{6}$$

for some ξ_i between x_1 and x_i . If we substitute these expansions into (7.7.5), drop the cubic terms, and then match the coefficients of $f(x_1)$, $f'(x_1)$, and $f''(x_1)$, we find that

$$a + b + c = 0,$$

$$b(x_2 - x_1) + c(x_3 - x_1) = 1, \quad (7.7.6)$$

$$b(x_2 - x_1)^2 + c(x_3 - x_1)^2 = 0,$$

which is a linear system in the unknown a, b , and c . As long as the x_i are distinct, (7.7.6) will have a unique solution for a, b , and c . In the symmetric two-sided case where $x_2 < x_1 < x_3$, and $x_3 - x_1 = x_1 - x_2 = h$, the solution is $a = 0$, $c = -b = 1/(2h)$, the two-point formula. Since we dropped only cubic terms from (7.7.5), this approximation has an error proportional to $(x_1 - x_2)^3 + (x_3 - x_1)^3$. Sometimes, we don't have the symmetric case. Suppose that we instead have x_2 and x_3 positioned to the right of x_1 with $x_2 = x_1 + h$, and $x_3 = x_1 + 2h$. Then $a = -3/(2h)$, $b = 2/h$, and $c = -1/(2h)$ produces a formula with a possibly larger cubic error.

The same approach can be used to compute a three-point formula for $f''(x_1)$ with a quadratic error. Similarly higher-order formulas can be derived for first and second derivatives.

7.8 Software

There is some integration software available in the public domain. Elhay and Kautsky's IQPACK is a Fortran program, available from Netlib, that will compute the Gaussian quadrature and interpolatory quadrature nodes and weights for a large variety of weighting functions. Instead of carrying around a large table of nodes and weights, it is convenient to have a program compute the necessary nodes and weights for a specified interval and weighting function. This generally takes little time and increases the flexibility of a program. QUADPACK, presented in Piessens et al. (1983), is a large package of integration routines for univariate problems, with a focus on adaptive routines.

7.9 Further Reading and Summary

This chapter presented the classical topics in numerical quadrature. Many of the integrals encountered in economics have very well-behaved integrands with many derivatives and simple shape properties such as monotonicity, concavity, and unimodality. This gives economists a variety of formulas to use. The importance of integration in economics will encourage economists to find good integration formulas aimed at integrating the sets of functions they need to integrate.

Davis and Rabinowitz (1984) is the classic in numerical quadrature, but it is a bit dated in its evaluations which are based on technology available at the time. Evans (1993) and Zwillinger (1992) are recent books that cover some recent developments. One new approach is to use the computer to find good integration formulas, substituting computer power for clever algebra and analysis; Cohen and Gismalla (1985, 1986) give some useful examples.

Exercises

1. The least squares orthogonal polynomial approximations discussed in chapter 6 require the evaluation of integrals. Use Gauss-Legendre quadrature to compute the n -term Legendre approximation for x^α on $[a, b]$ for $n = 3, 6, 15$, $\alpha \in \{0.5, 1\}$, $a \in \{0.2, 1\}$, and $b = \{2, 5\}$. In all cases compute the L^∞ and L^2 errors over $[a, b]$.
2. Using the transformation (7.3.7) and appropriate numerical integration rules, compute the n -term Chebyshev least-squares approximation for x^α on $[a, b]$ for $n = 3, 6, 15$, $\alpha \in \{0.5, 1\}$, $a \in \{0.2, 1\}$, and $b \in \{2, 5\}$. In all cases compute the L^∞ and L^2 errors over $[a, b]$.
3. Simpson's rule is based on a local quadratic approximation of the integrand. Derive the corresponding cubic and quartic approximations.
4. Compute three-point approximations for $f'(a)$ and $f''(a)$ using $f(x)$ at $x = a + h, a + 2h$, and $a + 3h$. Next, compute five-point approximations for $f'(a)$ and $f''(a)$ using $f(x)$ at $x = a, a \pm h$, and $a \pm 2h$.
5. Compute $\int_0^\infty e^{-pt} u(1 - e^{-\lambda t}) dt$ for $p \in \{0.04, 0.25\}$, $u(c) = -e^{-ac}$, $a \in \{0.5, 1.0, 2.0, 5.0\}$, $\lambda \in \{0.02, 0.05, 0.10, 0.20\}$, using Gauss-Laguerre quadrature. Use changes of variables to find an equivalent integral over $[0, 1]$, and devise a Newton-Cotes approach to the problem. Compare the Gauss-Laguerre and Newton-Cotes approaches for 3, 5, and 10 node formulas.
6. The Tchakaloff and Mysovskikh theorems tell us that good monomial quadrature formulas exist. Write a program that will search for Tchakaloff nodes and weights for hypercubes of dimension d and valid for monomials with total degree m for $m \in \{4, 5\}$, $d \in \{3, 4, 5\}$.
7. Assume that there n individual investors and m assets with asset j paying a random amount Z^j . Let e_j^l denote agent l 's endowment of asset j . Suppose that investor l 's objective is to maximize expected over utility, $E\{u_l(W)\}$ over final wealth, W . Write a program that reads in data on tastes, returns, and endowments, and computes the equilibrium price of the assets and the expected utility of each investor. First, let $u_l(W) = -e^{-a_l W}$ where $a_l > 0$. Begin with three agents ($n = 3$), one safe asset paying 1 and two risky assets ($m = 3$). Assume that the two risky assets are normally distributed with variance-covariance matrix Σ and mean μ ; try $\mu^T = (1.1, 1.2)^T$ and $\Sigma = \begin{pmatrix} 0.1 & 0.02 \\ 0.02 & 0.2 \end{pmatrix}$. Use Gauss-Hermite quadrature and two-dimensional monomial formulas, and compare the results to the true solution.
8. Compute the discounted utility integral
$$\int_0^\infty e^{-pt} u(c(t), l(t)) dt,$$
where $c(t) = 1 - e^{-\mu_1 t}$, $l(t) = 1 - e^{-\mu_2 t}$, and $u(c, l) = (c^\sigma + l^\sigma)^{(\gamma+1)/\sigma}/(\gamma+1)$. Let $\gamma \in \{-0.5, -1.1, -3, -10\}$, $\sigma \in \{0.5, 2.0\}$, $\mu_1 \in \{0.05, 0.1, 0.2\}$, $\mu_2 \in \{0.05, 0.1, 0.2\}$, and $p = 0.05$. Use both Simpson's rule and Gauss-Laguerre rules. How do the choices of γ, σ, μ_1 , and μ_2 affect accuracy?
9. Suppose that a firm's output at time t is $q = 3 - (1 + t + t^2)e^{-t}$, price is $P(q) = q^{-2}$, cost is $C(q) = q^{3/2}$, and the interest rate is 0.1. What are profits over the interval $[0, 10]$? What are profits over $[0, \infty)$? Use several methods for each integral. Which do best?

8 Monte Carlo and Simulation Methods

In this chapter we discuss various sampling methods that are intuitively based on probability ideas. We will use such sampling methods to solve optimization problems, solve nonlinear equations, and compute integrals. These methods are often called *Monte Carlo* methods, but they also include *stochastic approximation*, *genetic algorithms*, and *simulated annealing*.

Monte Carlo methods have features that distinguish them from most of the methods we have studied in previous chapters. First, they can handle problems of far greater complexity and size than most other methods. The robustness and simplicity of the Monte Carlo approach are its strengths. Second, Monte Carlo methods are intuitively based¹ on laws of large numbers and central limit theorems. Since most readers are already acquainted with these probability results, the marginal cost of learning Monte Carlo methods is low. This combination of general applicability and low marginal cost of learning help explain the popularity of Monte Carlo methods.

The probabilistic nature of Monte Carlo methods has important implications. The result of any Monte Carlo procedure is a random variable. Any numerical method has error, but the probabilistic nature of the Monte Carlo error puts structure on the error that we can exploit. In particular, the accuracy of Monte Carlo methods can be controlled by adjusting the sample size. With Monte Carlo methods we can aim for answers of low but useful accuracy as well as for answers of high accuracy, and we can easily estimate the cost of improved accuracy.

In this chapter we will present basic examples of Monte Carlo approaches to a wide variety of problems. In the next chapter we examine the logical basis of Monte Carlo methods, a discussion that naturally leads to consideration of the more general category of sampling methods. Together these chapters provide us with a collection of tools and ideas that allow us to examine problems too complex and large for more conventional methods.

8.1 Pseudorandom Number Generation

We should first deal with some semantic (some might say metaphysical) but substantive issues. Strictly speaking, Monte Carlo procedures are almost never used since it is difficult to construct a random variable. There have been attempts to construct “true” random variables,² but such methods are costly and awkward. Furthermore it

1. We will later see that the true logical and mathematical foundation for the methods actually used in practice is quite different.
2. One old procedure was to examine the computer’s clock time and use the least significant digits as a random variable; however, this practice could easily degenerate into a cycle if a computer’s programs became too synchronized. Another approach used geiger counter readings. True randomness may be an important feature of future computing, since results of quantum computation would often be random (under the Heisenberg interpretation). However, current technology avoids true randomness.

is not generally practical to replicate calculations that have truly random features, since replication would require one to store all the realizations of the random variables, a task that would often exceed a computer's storage capacity.

Constructing random variables can also drag one into side issues of little interest to numerical analysis. For example, suppose that you generate a random sequence and use it in a problem, but you also record the sequence on paper. A tricky question arises when you later use the recorded sequence in another problem. Is the second result produced deterministically because you deterministically read the recorded sequence, or is it stochastic because the sequence originally came from a stochastic process? If you say stochastic, would your answer change if told that that random process produced a constant string of zeros? And, finally, what mathematical property of the recorded sequence is changed if I next told you that I had lied and that the sequence had been constructed according to a deterministic rule?

Because of the practical difficulties, truly random variables are seldom used in Monte Carlo methods. This fact makes discussions related to randomness and the advantages of random over deterministic approaches of no practical value. In this chapter we will follow the standard practice in economics of using probabilistic language and ideas to describe and analyze Monte Carlo methods; however, the reader should be aware of the fact that this is not a logically valid approach to analyzing methods that don't use true random variables.

Almost all implementations of Monte Carlo methods actually use *pseudorandom sequences*, that is, deterministic sequences, X_k , $k = 1, 2, \dots, N$, that *seem* to be random. "Pseudorandom" essentially means that a sequence displays *some* properties satisfied by random variables. Two basic properties generally demanded of pseudorandom sequences are zero serial correlation and the correct frequency of *runs*. A run is any sequential pattern, such as "five draws below 0.1" or "six draws each greater than the previous draw." The zero serial correlation condition demands that if we take a finite subsequence of length $n < N$, X_k , $k = 1, n$, and we regress X_{k+1} on X_k , the regression coefficient is not "statistically significantly" different from zero. No real statistics is involved because the sequence X_k is deterministic; what we mean is that if we apply statistical techniques to X_k acting as if X_k were random, we would find no statistical evidence for serial correlation. More generally, a candidate pseudorandom sequence should display zero serial correlation at all lags. The serial correlation properties are only linear properties. Analyzing the frequency of runs allows us to examine some nonlinear aspects of X_k . For example, if X_k is i.i.d., then it is unlikely that there is a run of ten draws each greater than its predecessor, but there should be a few in a long sequence. These tests are described in more detail in Rubinstein (1981).

However, no pseudorandom sequence satisfies *all* properties of an i.i.d. random sequence. Any pseudorandom sequence would be found to be deterministic by the methods described in Brock et al. (1986, 1988, 1991). We should keep in mind Lehmer's point that such a pseudorandom sequence is "a vague notion . . . in which each term is unpredictable to the *uninitiated* and whose digits pass a *certain number* of tests traditional with *statisticians*." Keeping these distinctions in mind, we refer to pseudorandom number generators as random number generators.

Uniform Random Number Generation

The simplest pseudorandom number generators use the *linear congruential method* (LCM) and take the form

$$X_{k+1} = aX_k + c \pmod{m}. \quad (8.1.1)$$

One popular example chooses $a = \pm 3 \pmod{8}$ and close to $2^{b/2}$, $c = 0$, X_0 odd, and $m = 2^b$, where b is the number of significant binary bits available on the computer. This will generate a sequence with period 2^{b-2} . If we want a multidimensional uniform pseudorandom number generator, we assign successive realizations to different dimensions. For example, the collection of points $Y_n \equiv (X_{2n+1}, X_{2n+2})$ is a pseudorandom two-dimensional set of points.

The graph of a typical LCM generator is illustrated in figure 8.1. Note the steep slope of the function at each point and the numerous discontinuities. Figure 8.2 gives a typical two-dimensional collection of 1,500 pseudorandom points. Note that the scatter resembles what we expect from a random number generator. The scatter is not uniform. There are some areas where there are too many points and some areas where there are too few, as predicted by the central limit theorem.

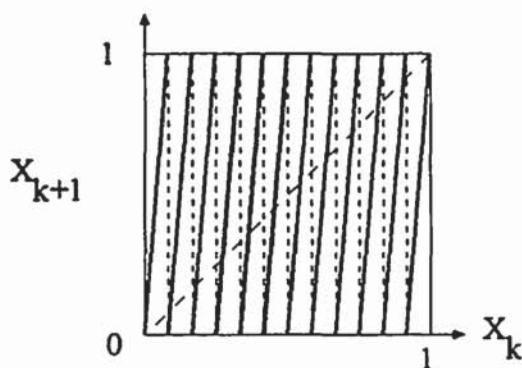


Figure 8.1
Graph of LCM rule

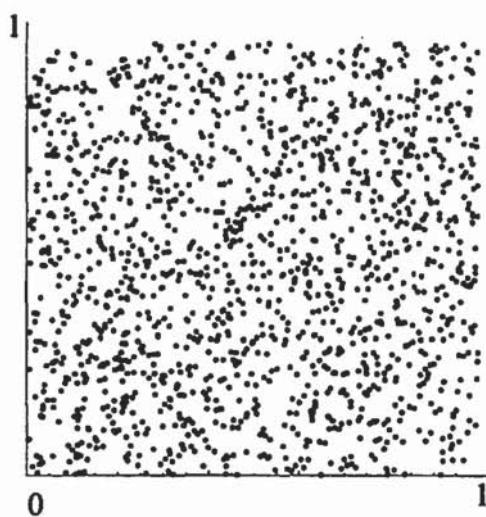


Figure 8.2
Initial 1,500 pseudorandom points

In general, LCM generators have fallen into disfavor. Marsaglia (1968) showed that successive d -tuples of numbers generated by LCM generators live in a few planes in $[0,1]^d$. Specifically, if we have $X_{k+1} = aX_k + c \pmod{m}$, and form the d -tuples $Y_k \equiv (X_k, X_{k+1}, \dots, X_{k+d})$, the Y_k points lie on at most $(d!m)^{1/d}$ hyperplanes. For example, if $m = 2^{31}$ and $d = 6$, the successive d -tuples lie on at most 108 hyperplanes. This implies that moderate-dimensional Monte Carlo methods that rely on LCM generators are really sampling from relatively few hyperplanes, a fact that could lead to poor results. The problem also becomes more severe for higher dimensions.

The response to these problems has been to develop more complex generators. A simple example of this is the multiple prime random number generator (MPRNG) from Haas (1987). Algorithm 8.1 displays Fortran code, taken from Haas, that generates a sequence of NSAMP values for IRAND:

Algorithm 8.1 MPRNG Algorithm

```
M=971;IA=11113;IB=104322;IRAND=481
DO I=1,NSAMP
  M=M+7;IA=IA+1907;IB=IB+73939
  IF(M.GE.9973)M=M-9871
  IF(IA.GE.99991)IA=IA-89989
  IF(IB.GE.224729)IB=IB-96233
  IRAND=MOD[IRAND*M+IA+IB,100000]/10
ENDDO
```

This method generates integers³ between 0 and 9999 for IRAND; to generate numbers with more digits (and more precision) one just concatenates two successive iterates to get one eight digit number. We use IRAND/10,000 as a pseudorandom variable to approximate the uniform random variable on [0, 1]. In this algorithm the multiplier M and the increments IA and IB are constantly changing, in contrast to the linear congruential method where they are constant. This mixing results in a method that avoids many of the problems of other methods and results in a large period of over 85 trillion.

There are many other ways to generate pseudorandom sequences. One can use nonlinear congruential methods that take the form $X_{k+1} = f(X_k) \bmod m$. One old method was the Fibonacci generator $X_k = (X_{k-1} + X_{k-2}) \bmod m$. This sequence has a number of poor properties. In particular, if X_{k-1} and X_{k-2} are small relative to m , so will be X_k . However, an updated version that begins with 55 odd numbers and computes

$$X_k = (X_{k-24} \cdot X_{k-55}) \bmod 2^{32} \quad (8.1.2)$$

is much better, having a period length of approximately 10^{25} and passing many randomness tests.

Nonuniform Random Number Generation

The random number generators constructed above yield approximations to the uniform random variable on [0, 1]. We use them to construct random number generators approximating other random variables.

The most general method is called *inversion*. Suppose that we want to simulate a scalar random variable with distribution $F(x)$ and U is a uniform random variable. Then $F^{-1}(U)$ has distribution function $F(x)$. Therefore, to approximate a random variable with distribution $F(x)$, we need only approximate F^{-1} and use $y = F^{-1}(U)$ where U is generated by a pseudorandom sequence. For example, suppose that we want to generate a random variable that has density proportional to x^2 on [1, 2]. Since $\int_1^2 x^2 dx = 7/3$, the density function is $3x^2/7$, the distribution function is $F(x) = \frac{3}{7} \int_1^x z^2 dz = (x^3 - 1)/7$, and $F^{-1}(u) = (7u + 1)^{1/3}$. Therefore $X \equiv (7U + 1)^{1/3}$ is a random variable over [1, 2] with density $3x^2/7$. Notice that the inversion method allows us to translate all random variables into nonlinear functions of $U[0, 1]$. Sometimes one cannot analytically determine F^{-1} . Approximating F^{-1} is just a problem in approximation, solvable by using the methods of chapter 6. Of

3. The division in the MPRNG algorithm is integer division that drops any noninteger piece of the result.

particular importance here would be preserving the monotonicity of F^{-1} ; hence a shape-preserving method is advised if other methods fail to preserve shape.

Some random variables can be constructed more directly. A popular way to generate $X \sim N(0, 1)$ is the *Box-Muller method*. If U_1 and U_2 are two independent draws from $U[0, 1]$, then

$$\begin{aligned} X_1 &= \cos(2\pi U_1) \sqrt{-2 \ln U_2}, \\ X_2 &= \sin(2\pi U_1) \sqrt{-2 \ln U_2}, \end{aligned} \tag{8.1.3}$$

are two independent draws from $N(0, 1)$.

Gibbs Sampling

Some random variables are difficult to simulate, particularly in multivariate cases. The *Gibbs sampling method* constructs a Markov chain that has a useful ergodic distribution.

Suppose that $f: R^d \rightarrow R$ is the probability density for the random variable X , and let $f(z|y)$ denote the density of $Z \in R^l$ conditional on $y \in R^m$ where $l + m = d$. We construct a sequence, x^k , in a componentwise fashion. We start with an arbitrary x^0 . Then given x^k , we construct x^{k+1} componentwise from a sequence of d draws satisfying the densities in (8.1.4):

$$\begin{aligned} x_1^{k+1} &\sim f(x_1 | x_2^k, \dots, x_d^k), \\ x_2^{k+1} &\sim f(x_2 | x_1^{k+1}, x_3^k, \dots, x_d^k), \\ &\vdots \\ x_d^{k+1} &\sim f(x_d | x_1^{k+1}, \dots, x_{d-1}^{k+1}). \end{aligned} \tag{8.1.4}$$

Each component of x^{k+1} is constructed by a univariate random draw conditional on the most recent draws of the other variates. These univariate draws can be performed by the inversion method or an appropriate direct method.

The key theorem is due to Gelfand and Smith (1990).

THEOREM 8.1.1 Let $g: R^d \rightarrow R$ be measurable, and let the sequence x^k be generated by (8.1.4). Then

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{j=1}^n g(x^j) = \int g(x) f(x) dx = E\{g(X)\}. \tag{8.1.5}$$

The key component to a Gibbs sampler is the collection of conditional random variates. If they are easy to generate, then the Gibbs method is quite practical. The-

orem 8.1.1 says that eventually the x^k sequence becomes sufficiently uncorrelated that it satisfies (8.1.5), which is similar to a law of large numbers. This occurs despite the substantial correlation between x^k and x^{k+1} . If we want a sequence of x^k with nearly zero serial correlation, we take x^1, x^{K+1}, x^{2K+1} , and so on, for a large K . This then provides us with a method for creating nearly independent draws from X .

Gibbs sampling is a powerful tool in many dynamic contexts. We do not develop any of its uses here; see Geweke (1996) for discussion of Gibbs sampling and its applications.

8.2 Monte Carlo Integration

When the dimension is high, simple numerical integration methods, such as product formulas, are likely to be impractical. Each of those methods would do well for smooth integrands if one can evaluate the integrand at enough points. Unfortunately, the minimal necessary number can be very high, and results using fewer points are of no value. Monte Carlo methods for computing integrals can deliver moderately accurate solutions using a moderate number of points. As with the quadrature methods discussed in chapter 7, we select points at which we evaluate the integrand and then add the resulting values. The differentiating feature of Monte Carlo methods is that the points are selected in a “random” fashion instead of according to a strict formula.

The law of large numbers (LLN) motivates Monte Carlo integration. Recall that if X_i is a collection of i.i.d. random variables with density $q(x)$ and support $[0, 1]$, then

$$\lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N X_i = \int_0^1 x q(x) dx, \quad a.s.$$

Furthermore

$$\text{var}\left(N^{-1} \sum_{i=1}^N X_i\right) = \frac{\sigma_x^2}{N},$$

where $\sigma_x^2 = \text{var}(X_1)$. If σ_x^2 is not known a priori, an unbiased estimator is

$$\hat{\sigma}_x^2 \equiv (N-1)^{-1} \sum_{i=1}^N (X_i - \bar{X})^2,$$

where \bar{X} is the sample mean, $\bar{X} \equiv N^{-1} \sum_{i=1}^N X_i$.

The LLN immediately suggests a numerical integration procedure. Suppose that one wants to compute $I_f \equiv \int_0^1 f(x) dx$. If $X \sim U[0, 1]$, then

$$E\{f(X)\} = \int_0^1 f(x) dx;$$

that is, $\int_0^1 f(x) dx$ equals the mean of $Y \equiv f(X)$. Hence one way to approximate $\int_0^1 f(x) dx$ is to estimate $E\{Y\}$. The *crude Monte Carlo* method generates N draws from $U[0, 1]$, $\{x_i\}_{i=1}^N$, and takes

$$\hat{I}_f \equiv \frac{1}{N} \sum_{i=1}^N f(x_i)$$

as an estimate of $\int_0^1 f(x) dx$.

Monte Carlo procedures differs substantially from our earlier methods in that the approximation is itself a random variable, \hat{I}_f , with variance

$$\sigma_{\hat{I}_f}^2 = N^{-1} \int_0^1 (f(x) - I_f)^2 dx = N^{-1} \sigma_f^2.$$

An estimate of the variance of $f(X)$ is

$$\hat{\sigma}_f^2 = (N-1)^{-1} \sum_{i=1}^N (f(x_i) - \hat{I}_f)^2.$$

Since σ_f^2 is unknown, we take $\hat{\sigma}_f^2$ as an estimate of σ_f^2 . The standard error of \hat{I}_f is σ_f/\sqrt{N} .

These variance calculations are central to any Monte Carlo procedure. Since Monte Carlo quadrature procedures yields a random variable, any estimate of \hat{I}_f should be accompanied with an estimate of $\sigma_{\hat{I}_f}^2$. Presenting an estimate of \hat{I}_f without reporting $\hat{\sigma}_{\hat{I}_f}^2$ is like presenting a parameter estimate without also reporting a confidence interval.

The crude Monte Carlo method is seldom used without modification. Although this method is unbiased, its variance is too large. There are a variety of simple techniques that can substantially reduce the variance of the integral estimate but retain the unbiasedness of the estimator.

Stratified Sampling

The first variance reduction technique is *stratified sampling*. The observation is that the variance of f over a subinterval of $[0, 1]$ is often less than over the whole interval. Suppose that we divide $[0, 1]$ into $[0, \alpha]$ and $[\alpha, 1]$. Then, if we have N points sampled over $[0, \alpha]$ and N over $[\alpha, 1]$, we form the estimate

$$\hat{I}_f = \frac{\alpha}{N} \sum_i f(x_{1i}) + \left(\frac{1-\alpha}{N} \right) \sum_i f(x_{2i}),$$

where $x_{1i} \in [0, \alpha]$ and $x_{2i} \in [\alpha, 1]$, $i = 1, \dots, N$. Its variance is

$$\frac{\alpha}{N} \int_0^\alpha f^2 + \frac{(1-\alpha)}{N} \int_\alpha^1 f^2 - \frac{\alpha}{N} \left(\int_0^\alpha f \right)^2 - \frac{(1-\alpha)}{N} \left(\int_\alpha^1 f \right)^2.$$

A good α equates the variance over $[0, \alpha]$ with that over $[\alpha, 1]$.

Note that the basic idea of stratified sampling is to keep the draws from clumping in one region. This feature is typical of acceleration schemes.

Antithetic Variates

A popular acceleration method is that of antithetic variates. The idea is that if f is monotonically increasing, then $f(x)$ and $f(1-x)$ are negatively correlated. One way to apply this idea to reduce the variance of \hat{I}_f for monotone f is to estimate I_f with the antithetic estimate

$$\hat{I}_f^a = \frac{1}{2N} \sum_{i=1}^N (f(x_i) + f(1-x_i)).$$

The antithetic estimate is an unbiased estimate of I_f but will have smaller variance than the crude estimate whenever the antithetic summands are negatively correlated. The antithetic variate method is particularly valuable for monotonic functions f ; in particular, if f is linear (one-dimensional) function, then antithetic variates will integrate $\int_0^1 f(x) dx$ exactly.

Control Variates

The third method is that of *control variates*. Suppose that we know of a function, φ , that is similar to f but easily integrated. Then the identity $\int f = \int \varphi + \int(f - \varphi)$ reduces the problem to a Monte Carlo integration of $\int(f - \varphi)$ plus the known integral $\int \varphi$. The variance of $f - \varphi$ is $\sigma_f^2 + \sigma_\varphi^2 - 2 \text{cov}(f, \varphi)$. So if $\text{cov}(f, \varphi)$ is large, the N -point variance of the crude methods applied to $f - \varphi$ is smaller than $(1/N)\sigma_f^2$, the variance of the crude method. Therefore the control variate method estimates I_f with $\int \varphi$ plus a Monte Carlo estimate of $\int(f - \varphi)$.

Importance Sampling

The crude method to estimate $\int_0^1 f(x) dx$ samples $[0, 1]$ in a manner unrelated to the integrand $f(x)$. This can be wasteful, since the value of f in some parts of $[0, 1]$ is

much more important than in other parts. Importance sampling tries to sample $f(x)$ where its value is most important in determining $\int_0^1 f(x) dx$. If $p(x) > 0$, and $\int_0^1 p(x) dx = 1$, then $p(x)$ is a density and

$$I_f = \int_0^1 f(x) dx = \int_0^1 \frac{f(x)}{p(x)} p(x) dx.$$

The key idea behind importance sampling is that if x_i is drawn with density $p(x)$, then

$$\hat{I}_f^p = \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x_i)}.$$

is an unbiased estimate of I , and its variance is

$$\sigma_{\hat{I}_f^p}^2 = \frac{1}{N} \left(\int_0^1 \frac{f(x)^2}{p(x)} dx - \left(\int_0^1 f(x) dx \right)^2 \right).$$

If $f(x) > 0$ and $p(x) = f(x) / \int_0^1 f(x) dx$, then $f(x) = I_f p(x)$ and $\sigma_{\hat{I}_f^p}^2 = 0$. If $f(x)$ is negative somewhere but bounded below by $B < 0$, we can integrate $f(x) - B$ in this way. Of course, if we could construct this $p(x)$, we would have solved the problem. This example points out that we want to use a $p(x)$ that similar to $f(x)$ in shape, thereby minimizing the variance of $f(x)/p(x)$.

The construction of \hat{I}_f^p is called *importance sampling* because we sample more intensively where f is large, which is where $f(x)$ makes the greatest contribution to the integral $\int f(x) dx$. The cost of this method is that it may take some effort to construct a nonuniform random variable that has a density similar to $f(x)$. We must trade this off against the cost of evaluating $f(x)$ at a larger number of points.

A key problem with importance sampling is clear from the $\sigma_{\hat{I}_f^p}^2$ formula. Note that the key integrand is $f(x)^2/p(x)$; if p goes to zero more rapidly than f^2 as x approaches points where p is zero, then this integrand will go to infinity. This is the case if f has fat tails and p has thin tails. Since a popular choice for p is the thin-tailed normal distribution, this *thin tails problem* arises easily. We must choose p to avoid this problem.

Importance Sampling and Changes of Variables

Importance sampling is usually described in probabilistic terms, giving the impression that importance sampling is probabilistic in nature. The truth is that importance sampling is really an application of the change of variables (COV) theorem discussed

in section 7.1. This observation helps us understand importance sampling and its relation to other integration tools.

To see the connection between importance sampling and changes of variables, let $x(u): [0, 1] \rightarrow D$ be a monotone increasing map, and let $u(x)$ be its inverse. Then $u = u(x(u))$, $1 = u'(x(u)) x'(u)$, and the COV theorem implies that

$$\int_D f(x) dx = \int_0^1 f(x(u)) x'(u) du = \int_0^1 \frac{f(x(u))}{u'(x(u))} du = E\left\{\frac{f(x(U))}{u'(x(U))}\right\},$$

where $U \sim U[0, 1]$.

Consider the maps $u(x)$ and its inverse $x(u)$. Then $1/x'(u(x)) = u'(x)$ and $X = x(U)$ has density $p(x) = u'(x)$. This last equivalence is particularly on point, since it shows that any monotonic map $x: [0, 1] \rightarrow R$ can be used to transform $\int_R f(x) dx$ to an equal integral over $[0, 1]$, which in turn has the probabilistic interpretation as the expectation of a function of U . However, any method can be used to compute the resulting integral over $[0, 1]$, not just Monte Carlo.

The key task in importance sampling, or any COV method, lies in choosing the map $x(u)$. If one ultimately uses Monte Carlo, it is natural to choose a map that leaves the $\text{var}[f(x(U))/u'(x(U))]$ as small as possible, since our error estimates are expressed in terms of this variance. In practice, the objective is to find a $x(u)$ that leaves $f(x(u))/u'(x(u))$ nearly flat. We also want to avoid the thin tails problem that arises if $f(x(u))/u'(x(u))$ is not bounded for $u = 0$ or $u = 1$. This is avoided by choosing $x(u)$ so that the implied density $u'(x(U))$ does not go off to zero too rapidly as U goes to zero or one.

Sometimes the choice of sampling distribution is suggested by the integrand. If the integrand can be expressed in the form $\int_D g(x)f(x) dx$ where $f \geq 0$ on D , then we would like to create a X with density proportional to $f(x)$. Such an X would have density $f(x)/\int_D f(x) dx$, which has distribution function $F(x)$. Then $X = F^{-1}(U)$ has density proportional to $f(x)$ on D , and

$$\int_D g(x)f(x) dx = \left(\int_D f(x) dx \right) E\{g(X)\}.$$

For example, suppose that $D = [0, 1]$ and $f(x) = x^3$. Then $\int_D x^3 dx = \frac{1}{4}$, $F(x) = 4 \int_0^x z^3 dz = x^4$, and $X = (U)^{1/4}$ has density $4x^3$. Hence

$$\int_0^1 g(x)x^3 dx = \frac{1}{4} E\{g(U^{1/4})\},$$

which we approximate with the finite sum $4N^{-1} \sum_{j=1}^N g(u_i^{1/4})$ where the u_i are generated by a uniform pseudorandom number generator.

A few final points need to be made about these acceleration methods. First, despite the use of the term “acceleration,” the *rate* of convergence is still proportional to $N^{-1/2}$ as with crude Monte Carlo. All that is achieved is a reduction in the proportionality constant, not the convergence rate. Second, these methods replace $\int f$ with $\int g$ which has the same value but smaller variance when we apply the crude Monte Carlo method; hence, in some way, we always end up using crude Monte Carlo.

8.3 Optimization by Stochastic Search

Some optimization problems involve objectives that are rough and have many local extrema. Polytope methods can handle nonsmooth objectives, but they are purely local search methods and will stop at the first local extremum found. To solve these ill-behaved problems, one must adopt procedures that rely neither on smoothness nor on local comparisons. In this section we will examine a variety of stochastic search methods commonly used to solve general optimization problems. In practical terms, these are the only general, globally convergent optimization methods we have.

Optimization by Random Search

One simple procedure for optimization is to evaluate the objective at a random collection of numbers and take the best. The law of large numbers assures us that we will almost surely converge to the optimum as we take larger samples. This method is slow. However, we can substantially improve its performance if each randomly generated point is also used to begin a more standard optimization procedure. One such example would be to apply a fixed number of iterations of some standard method to each randomly drawn point. The points generated by the standard optimization procedure are included in the total collection when we choose the best. Combining a random search with standard methods takes advantage of each method’s strengths: The randomness makes sure that we don’t get stuck at a bad point, and if one of these randomly chosen points is close to the optimum, a locally convergent standard method will send us to the optimum.

In problems where Newton’s method and other derivative-based procedures are not effective or not applicable, random search methods are more useful. However, searching the choice set in a serially uncorrelated fashion is seldom efficient. The following procedures learn from past draws to focus their search on regions more likely to contain the solution.

Genetic Algorithms

Genetic algorithms are one class of random search optimization methods that has recently received much attention. The intuition behind genetic methods arises from biological ideas of evolution, fitness, and selection, as will be clear from the terminology. The reader is encouraged to read manuscripts such as Holland (1975) and Goldberg (1989) for the history of these ideas. I will just present a formal description of the method, using basic mathematical ideas to guide our intuition.

Suppose that we want to solve

$$\max_{x \in S} f(x), \quad (8.3.1)$$

where $f: R^n \rightarrow R^n$ and S is a subset of R^n , of the form $S = S_1 \times S_2 \times \cdots \times S_n$, $S_i = [a_i, b_i]$. Since f is to model “fitness,” it is natural that it be nonnegative. If f is not nonnegative everywhere on S , then we replace it with a function of the form $\max\{0, f(x) - f(c)\}$ for some $c \in S$ where $f(c) < 0$. Such a replacement in (8.3.1) will leave the problem essentially unchanged.

A genetic algorithm keeps track of f at a set of points, with that set evolving so as to keep points with high values of f . We start with choosing m points from S to construct $X_1 = \{x^1, \dots, x^m\}$. We construct a sequence of such sets; let X_k denote the set of x 's generated at stage k . The initial points may reflect some information about where the maxima of f are likely to lie, or can be chosen “at random.” The choice of m is important. Large m will make the search more global and reduce the likelihood of premature convergence, but at the cost of more evaluations of f .

The basic idea is to evaluate f at each $x \in X_k$. The points x are called “chromosomes,” and the value of f at x is a measure of the fitness of x . We then “mate” pairs of $x \in X_k$ to produce new chromosomes, such mating involving both “recombination” and “mutation.” Some of these new chromosomes then replace some of the initial chromosomes to create a new population, X_{k+1} . The algorithm produces a sequence of populations in this fashion and ends either when time runs out or when we have a point, x^* , that appears to be as good as possible. The details arise in specifying the mating rules, the rate of mutation, and rules for replacing old points with new points. In the following we will describe the implementation used in Dorsey and Mayer (1992).

Given the values of f at $x \in X_k$, we need to determine the mating process. We want those x with high f values to be “selected.” Therefore we take some nonnegative, increasing $T: R \rightarrow R$ and define

$$p_i = \frac{T(f(x^i))}{\sum_{x \in X_k} T(f(x))} \quad (8.3.2)$$

to be the probability that $x^i \in X_k$ is selected to participate. Dorsey and Mayer (1992) choose $T(y) = y + a$, where a may be altered between iterations to keep the probabilities p_i from being either too diffuse or too concentrated. Using the probabilities in (8.3.2), we make m draws from X_k with replacement, producing a set Y_k of m chromosomes that will participate in mating.

Next draw two points y^1, y^2 from Y_k without replacement. We want to “mate” y^1 and y^2 to produce two new vectors, z^1 and z^2 . To do so, we select $i \in \{1, 2, \dots, n\}$ at random and have y^1 and y^2 “crossover” at position i , that is

$$\begin{aligned} z^1 &= (y_1^1, \dots, y_i^1, y_{i+1}^2, \dots, y_n^2), \\ z^2 &= (y_1^2, \dots, y_i^2, y_{i+1}^1, \dots, y_n^1). \end{aligned} \quad (8.3.3)$$

We keep drawing pairs y^1 and y^2 from Y_k without replacement until we have exhausted Y_k .

This specification of crossover in (8.3.3) is just one possible way to specify the mating process. Another choice would be to construct z_i^1 (z_i^2) by adding the integer part of y_i^1 (y_i^2) to the fractional part of y_i^2 (y_i^1), $i = 1, \dots, n$. There are practically an infinity of ways to specify the mating process, each one revolving around a “schema,” that is, a way to code points in S . See Holland (1975) for the general theory and description of schemas.

Let Z_k consist of the m resulting z 's constructed after reproduction and crossover. We next “mutate” the points. For example, we could, with probability q , replace z_i^j with a random number drawn from $[a_i, b_i]$ (e.g., with the uniform distribution). We do this independently at all $j = 1, \dots, m$, $i = 1, \dots, n$. X_{k+1} , the next generation of points, then consists of the mutations of the Z_k .

The fundamental result, called the Schema theorem, asserts that if we make good choices for the mating and mutation process, the average value of f over X_{k+1} will generally be greater than the average value over X_k . Of course we cannot allow the process to go forever. One kind of rule is to stop when progress has been small for the last j iterations. More precisely, we stop at iteration k if

$$\left| \max_{x \in X_l} f(x) - \max_{x \in X_m} f(x) \right| < \delta$$

and

$$\left\| \arg \max_{x \in X_l} f(x) - \arg \max_{x \in X_m} f(x) \right\| < \varepsilon$$

for $l, m = k-j, \dots, k-1, k$.

There are numerous alternative ways to implement a genetic algorithm. We examine the Dorsey-Mayer implementation above because it is simple. Dorsey and Mayer applied this genetic algorithm to reestimate a variety of econometric analyses where the original estimation used standard optimization methods. Their reestimation using this genetic algorithm sometimes yielded point estimates statistically superior to and economically different from those published. We may initially wonder why the genetic algorithm did better than standard methods. Even when econometricians use standard optimization methods for estimation, they typically restart their computations at several points in order to find the global optimum. Therefore there is no inherent advantage in the genetic algorithm as long as users of standard methods do *enough* restarts. The reason why Dorsey and Mayer's genetic algorithm estimates were better is that their genetic algorithm took a more global view and was more exhaustive in its search than were the computations behind the original estimates. The basic lesson from the Dorsey and Mayer study is that we must be more careful concerning multiple local solutions than is the apparent standard practice, and we should make greater use of procedures that intensively look for global solutions. The genetic algorithm is one possible scheme that learns from experience and searches intelligently.

The most powerful approach may be a *hybrid genetic method*. In such hybrid methods, one begins with a collection of points and mates them to produce a new set, but then allows a standard optimization procedure to proceed for a fixed number of iterations from each of these points. This produces a new collection of parents that will be mated, and so on. The advantage of the hybrid method is that the standard procedures will move toward nearby local optima, and the resulting mating process will give preference to the better local optima. In this marriage of standard and genetic methods, we see that the genetic component is essentially providing a more systematic approach to restarts than random guesses.

Simulated Annealing

Suppose that $f(x)$ is defined over some domain X ; X can be discrete or continuous. Simulated annealing is, similar to the genetic algorithm, a method that uses the objective function $f(x)$ to create a nonhomogeneous Markov chain process that will asymptotically converge to the minimum of $f(x)$. We first create a *neighborhood*

system covering X ; that is, for each $x \in X$ there is a set $V_x \subset X$ such that $x \in V_x$ and for all $x, y \in X, x \in V_y$ iff $y \in V_x$. Essentially V_x is the set of points that are neighbors of x . A further requirement is that any two points, x and y , can be linked by a sequence of points, x_i , such that $x_0 = x, x_{i+1} \in V_{x_i}, i = 0, 1, \dots, n$, and $x_{n+1} = y$. One simple example would be $X = \{l/n : l = 0, 1, \dots, n\}$ for some integer n with $V_x = (x - \varepsilon, x + \varepsilon) \cap X$ for some $\varepsilon > 1/n$. Simulated annealing is commonly used on combinatorial problems, such as the traveling salesman problem, where X is discrete and finite.

We will construct a Markov process on X . The idea is that the process can move from x to any of the neighbors of x ; the linkage condition ensures that the process can ultimately visit any point in X . The Markov process we construct has two stages to each step. If x_n is the n th realization, then we first draw a $y \in V_{x_n}$ which will be our candidate for the next value. However, the final choice of x_{n+1} is governed by the rules

$$f(y) < f(x_n) \Rightarrow x_{n+1} = y,$$

$$f(y) \geq f(x_n) \Rightarrow \begin{cases} \text{Prob}\{x_{n+1} = y\} = e^{-(f(y)-f(x_n))/T_n}, \\ \text{Prob}\{x_{n+1} = x_n\} = 1 - \text{Prob}\{x_{n+1} = y\}. \end{cases}$$

Hence, if $f(y) < f(x_n)$, we surely jump to y , but if $f(y) \geq f(x_n)$, we do so only probabilistically. T_n is called the temperature, and it depends on n , the position in the sequence. If T_n is large, then a jump to y almost always occurs, but as T_n falls, uphill jumps become less likely. Once we have determined the value of x_{n+1} , we then make a random draw $y \in V_{x_{n+1}}$, and proceed to compute x_{n+2} .

This process constructs a sequence x_n that is the realization of a (time-varying) Markov process on X which we hope converges to the minimum of f . The key theorem is that if $T_n > R/\log n$ for sufficiently large R , then x_n will almost surely converge to a global minimum of f . This kind of choice for T_n is called *exponential cooling*, which, assuming X is finite, has the property

$$\Pr\{x_n \notin f_{\min}\} = \mathcal{O}\left(\left(\frac{K}{n}\right)^\alpha\right),$$

where f_{\min} is the set of x which minimize $f(x)$, and K and α are positive constants.

Simulated annealing will be slow on conventional optimization problems, such as continuous functions on R^n . However, it has a good chance of producing good results in many complex situations where conventional methods (which focus on getting the perfect answer) may be impractical or in problems, such as those arising in

integer programming and combinatorial optimization, where conventional methods do not apply.

8.4 Stochastic Approximation

Frequently the objective of an optimization problem includes integration. Specifically, suppose that the objective is $f(x) = E\{g(x, Z)\}$, and we want to solve

$$\min_x E\{g(x, Z)\}, \quad (8.4.1)$$

where Z is a random variable. A conventional way to proceed would use a high-accuracy integration rule to compute $E\{g(x, Z)\}$. That may be quite costly⁴. In this section we discuss procedures that work well with low-quality evaluations of $E\{g(x, Z)\}$ and/or its gradients.

Stochastic approximation is designed to deal with such problems. We begin with an initial guess x^1 . Suppose that we take a draw, z^1 , from Z . The gradient $g_x(x^1, z^1)$ is an unbiased estimate of the gradient $f_x(x^1)$. In the steepest descent method we would let the next guess be $x^1 - \lambda_1 f_x(x^1)$ for some $\lambda_1 > 0$. In the *stochastic gradient method*, we execute the iteration

$$x^{k+1} = x^k - \lambda_k g_x(x^k, z^k), \quad (8.4.2)$$

where $\{z^k\}$ is a sequence of i.i.d. draws from Z and λ_k is a changing step size. Also we are constrained to remain in some set U ; if this step crosses ∂U , then we bounce back into the interior of U in the normal direction.

THEOREM 8.4.1 Suppose that f is C^2 . If $\lambda_k \rightarrow 0$, $\sum_{k=1}^{\infty} \lambda_k = \infty$, and $\sum_{k=1}^{\infty} \lambda_k^2 < \infty$, then the sequence x^k generated by the stochastic gradient method, (8.4.2), confined to U will almost surely have a subsequence that converges to a point either on ∂U or at a (possibly local) minimum of f .

The basic idea behind the stochastic gradient method is clear. First, since λ_k decreases to 0, it is small after an initial period. Even though $g_x(x, z)$ is a poor estimate of the gradient $f_x(x)$, it does have some information, and taking a small step in that direction will not affect x much. The only way x will move appreciably is if a long sequence of z 's pushes it in a particular direction. By the law of large numbers, the most likely direction in which it will be pushed is, to a first approximation, the

4. Econometricians frequently fix a set of z 's, say S , and minimize $\sum_{z \in S} g(x, z)$, an application of the crude Monte Carlo method to $E\{g(x, Z)\}$ that requires a large set S to yield an acceptable result.

Table 8.1
Statistics of (8.4.3) for 25 runs

Iterate	Average x_k	Standard deviation
1	0.375	0.298
10	0.508	0.143
100	0.487	0.029
200	0.499	0.026
500	0.496	0.144
1,000	0.501	0.010

true gradient. The condition that $\sum_{k=1}^{\infty} \lambda_k = \infty$ keeps λ_k from becoming small too quickly and forces the process to be patient before converging. As k increases, λ_k falls, and it takes more iterations for x to make a nontrivial step, strengthening the power of the law of large numbers. The condition that $\sum_{k=1}^{\infty} \lambda_k^2 < \infty$ forces the x^k sequence to settle down eventually, yielding an accumulation point somewhere almost surely. That such an accumulation point be at a local minimum of f is not surprising, for if it wandered close to such a point, the gradients will, on average, push x toward the minimum.

The performance of stochastic approximation is rough. For example, suppose that we want to solve $\min_{x \in [0,1]} E\{(Z - x)^2\}$ where $Z \sim U[0, 1]$. The solution is $x = 0.5$. If we use the sequence $\lambda_k = 1/k$, the scheme (8.4.2) becomes

$$x_{k+1} = x_k + \frac{2}{k}(z_k - x_k), \quad (8.4.3)$$

where the z_k are i.i.d. and distributed $U[0, 1]$. In table 8.1 we display the result of 25 runs of (8.4.3) with random x_1 . At iterate k we display the mean of the x_k and the variance. We see that the means are close to the correct solution. However, the variance is nontrivial even at iterate 1,000.

Equation (8.4.2) is really just solving for the zero of a nonlinear equation. In general, stochastic approximation is a stochastic process of the form

$$x^{k+1} = x^k - \lambda_k h(x^k, z^k). \quad (8.4.4)$$

If (8.4.4) converges to a limit x^* , then x^* is a zero of $E\{h(x, Z)\}$. Therefore we could use (8.4.4) to find zeros of such functions. This is really a stochastic form of successive approximation. In fact we can rewrite (8.4.4) as

$$x^{k+1} = (1 - \lambda_k)x^k + \lambda_k(x^k - h(x^k, z^k)), \quad (8.4.5)$$

which for small λ_k resembles a heavily damped extrapolation version of successive approximation applied to finding a zero of $h(x, z)$. The difference is that z is varying randomly in (8.4.5), and this turns the problem into finding a zero of $E\{h(x, Z)\}$.

Neural Network Learning

Stochastic approximation is often used to compute the coefficients of a neural network. Neural network coefficients are chosen to minimize a loss function. While one could directly minimize the loss function, most users of neural networks determine the coefficients using stochastic approximation methods to model a dynamic process of observation and learning.

Suppose that we have a neural network with weights θ . If we have a set of N data points (generated at random or otherwise), $(y_l, X^l), l = 1, \dots, N$, we want to find θ such that the L^2 error, $\sum_{l=1}^N (y_l - f(X^l, \theta))^2$, is small. In *back-propagation*, a sequence of weights, θ^k , are computed in an adaptive fashion; specifically

$$\theta^{k+1} = \theta^k + \lambda_k (\nabla f(X^k, \theta^k))^T (y_k - f(X^k, \theta^k)), \quad (8.4.6)$$

where λ_k is the learning rate sequence (consistent with theorem 8.4.1) and $\nabla f \equiv \partial f / \partial \theta$. The gradient of f tells us the direction of maximal increase and $y_k - f(X^k, \theta^k)$ is the approximation error for (y_k, X^k) with neural network weights θ^k . This adjustment formula tells us to change θ in the direction of maximal improvement given the values of θ^k and data (y_k, X^k) , and in proportion to the current error. Ideally the process in (8.4.6) continues indefinitely; in practice, it continues for $k = 1, 2, \dots, M$, where $M \gg N$.

While stochastic approximation has some intuitive value in describing dynamic learning processes, experiments indicate that it is not an efficient way to compute the neural network coefficients. To get good results using (8.4.6), one usually runs the data through (8.4.6) many times. Straightforward nonlinear least squares is often superior. However, both methods have problems with multiple local minima.

8.5 Standard Optimization Methods with Simulated Objectives

In this section we continue our discussion of problems of the form

$$\min_{x \in U} E\{g(x, Z)\} = f(x) \quad (8.5.1)$$

for some random variable Z . For many problems of the form in (8.5.1), the objective $f(x)$ and its derivatives can be computed only with nontrivial error. When solving problems of the form (8.5.1), we need to determine how well we need to approximate

the integral. Stochastic approximation was one way to solve (8.5.1). We will now consider standard optimization approaches that use simulation ideas.

The general idea is to take a sample of size N of Z , and replace $E\{g(x, Z)\}$ in (8.5.1) with its sample mean $1/N \sum_{i=1}^N g(x, Z_i)$. For example, suppose that we want to solve

$$\min_{x \in [0,1]} E\{(Z - x)^2\}, \quad (8.5.2)$$

where $Z \sim U[0, 1]$. To solve (8.5.2), we take, say, three draws from $U[0, 1]$; suppose that they are 0.10, 0.73, and 0.49. We then minimize the sample average of $(Z - x)^2$,

$$\min_{x \in [0,1]} \frac{1}{3}((0.10 - x)^2 + (0.73 - x)^2 + (0.49 - x)^2). \quad (8.5.3)$$

The solution to (8.5.3) is 0.43, a rough approximation of the true solution to (8.5.2) of 0.5.

We illustrate this approach to solving problems of the form (8.5.1) with a simple portfolio problem. In fact the use of Monte Carlo integration is quite natural for such problems, since we are essentially simulating the problem. For this example, assume that there are two assets and that the utility function is $u(c) = -e^{-c}$. Assume that the safe asset has total return $R = 1.01$, and the risky asset has return $Z \sim N(\mu, \sigma^2)$ with $\mu = 1.06$ and $\sigma^2 = 0.04$. The portfolio problem reduces to

$$\max_{\omega} -E\{e^{-(1-\omega)R+\omega Z}\}. \quad (8.5.4)$$

With our parameter values, the optimal ω , denoted ω^* , equals 1.25. The Monte Carlo approach to solve (8.5.4) is to use Monte Carlo integration to evaluate the integral objective. Specifically we make N draws of $Z \sim N(\mu, \sigma^2)$, generating a collection $Z_i, i = 1, \dots, N$, and replace (8.5.4) by

$$\max_{\omega} -\frac{1}{N} \sum_{i=1}^N e^{-(1-\omega)R+\omega Z_i}. \quad (8.5.5)$$

The solution to (8.5.5) is denoted $\hat{\omega}^*$ and hopefully approximates the solution to (8.5.4), ω^* .

The quality of this procedure depends on the size of N and how well the integral in (8.5.4) is approximated by the random sample mean in (8.5.5). We perform two experiments to examine the error. First, we set $\omega = 1.25$ and compute $-E\{e^{-(1-\omega)R+\omega Z}\}$ using m samples of N draws. Columns 2 and 3 of table 8.2 display the mean estimate of these m estimates, and the standard deviation of these m

Table 8.2
Portfolio choice via Monte Carlo

N	$N^{-1} \sum_{i=1}^N u(c_i)$		$\hat{\omega}^*$	
	Mean	Standard deviation	Mean	Standard deviation
100	-1.039440	0.021362	1.2496	0.4885
1,000	-1.042647	0.007995	1.2870	0.1714
10,000	-1.041274	0.002582	1.2505	0.0536

estimates, for $m = 100$, and $N = 100, 1,000$, and $10,000$. Notice that the mean estimates are all close to the true certainty equivalent of 1.04125; furthermore the standard errors are roughly 2, 0.8, and 0.2 percent. From these experiments one may conclude that using 1000 points will yield integral approximations with roughly 1 percent error.

We next turn to solving the optimization problem (8.5.4) and determine the number of points needed to solve (8.5.4). To do that, we solve (8.5.5) for $\hat{\omega}^*$ for m draws of 100, 1,000, and 10,000 points. The average $\hat{\omega}^*$ is close to the true value of 1.25, as shown in column 4 of table 8.2. However, the standard deviation of these solutions was 0.49 for $N = 100$, 0.17 for $N = 1,000$, and 0.05 for $N = 10,000$. Even for the largest sample size of 10,000, the standard error in computing the optimal ω was still 4 percent, which may be unacceptable. The key fact is that the error in computing ω^* is much larger, ten to twenty times larger, than the error in computing an expectation.

The conclusions of these exercises show that one must be much more careful in evaluating integrals when they are not of direct interest but computed as part of solving an optimization problem. Since individual optimization problems are important in equilibrium analysis, the same warning applies to the use of Monte Carlo in equilibrium analysis.

8.6 Further Reading and Summary

Monte Carlo methods are popular because of their ability to imitate random processes. Since they rely solely on the law of large numbers and the central limit theorem, they are easy to understand and robust. This simplicity and robustness comes at a cost, for convergence is limited to the rate $N^{-1/2}$.

There are several books on the generation of random variables and stochastic processes, including Ross (1990), Ripley (1987), Rubinstein (1986), Bratley, Fox, and

Schrage (1987), and Devroye (1986). Dwyer (1997) discusses the topic for economists. Ermoliev and Wets (1988), and Beneveniste, Métivier, and Priouret (1990), Kushner and Clark (1979), and Ljung and Soderstrom (1983), and Kushner and Huang (1979, 1980) discuss stochastic approximation. Gibbs sampling is being used to efficiently solve difficult empirical problems. Geweke (1994) discusses many topics in Monte Carlo simulation including Gibbs sampling. Bayesian methods often rely on Monte Carlo methods. John Geweke's web site <http://www.econ.umn.edu/bacc/> provides information about integration methods in Bayesian methods. Bootstrapping is an important computation intensive technique in statistics. The Efron (1994), and Lepage and Billard (1992) books present the basic ideas and applications.

Monte Carlo simulation is particularly useful in understanding economies where agents use particular learning strategies. This is discussed in Cho and Sargent (1996). Marimon et al. (1990), Arifovic (1994), and Marks (1992) are early applications of genetic algorithms to learning processes.

Goffe et al. (1992, 1994) presents applications of simulated annealing methods in econometrics. These papers demonstrate that simulated annealing is more robust than most alternative optimization schemes when there are multiple optima. This is particularly important in maximum likelihood problems. Aarts and Korst (1990) give an introduction to simulated annealing in general.

Exercises

1. Write a program that implements the MPRNG random number generator. Compute the correlation between X_t and X_{t-l} for $l = 1, 2, 3, 4$ using samples of size $10, 10^2, 10^4$, and 10^6 . Repeat this for (8.1.2).
2. Write a program that takes a vector of probabilities, $p_i, i = 1, \dots, n$, and generates a discrete random variable with that distribution.
3. Apply crude Monte Carlo integration to $x, x^2, 1 - x^2$, and e^x over the interval $[0, 1]$, using $N = 10, 10^2, 10^3$, and 10^4 . Next apply antithetic variates using the same sample sizes, and compare errors with crude Monte Carlo.
4. Use random search, simulated annealing, genetic algorithms, and stochastic approximation confined to $x \in [-5, 5]$ to solve

$$\max_{x \in R} \left(\frac{\cos x}{\sqrt{x^2 + a}} \right)$$

for various choices of $a \in [0.1, 10]$. Then do the same for the multivariate version

$$\max_{x \in R^n} \Pi_{i=1}^n \left(\frac{\cos x_i}{\sqrt{x_i^2 + a_i}} \right)$$

for $x \in [-5, 5]^n$, $n = 2, 3, 5, 10$ and random $a_i \in [0.1, 10]$.

5. Suppose that there are three agents, each initially holding \$1 and $\frac{1}{3}$ share of stock. Assume that the stock will be worth $Z \sim N(\mu, \sigma^2)$ tomorrow and a \$1 bond will be worth R surely. If agent i has the utility function $-e^{-a_i c}$ over future consumption, compute the equilibrium price of the equity. Let $a_i \in [0.2, 5]$, $R \in [1.01, 1.5]$, $\mu \in [1.05, 2.0]$ and $\sigma \in [0.001, 0.1]$. Use Monte Carlo integration of various sizes to compute the necessary integrals.
6. Use stochastic approximation to solve the asset demand example of section 8.5.
7. Use stochastic approximation to solve problem 5 above.
8. Use the inversion method to develop a random number generator for exponential distribution, for the Poisson distribution, and for the Normal distribution. Use suitable approximation methods from chapter 6.
9. Suppose X and Y are independent and both $U[0, 1]$, and $Z = (X^{0.2} + Y^{0.5})^{0.3}$. Compute polynomial approximations of the conditional expectation $E\{Y|Z\}$. What size sample do you need to get the average error down to .01.
10. Use Monte Carlo methods to compute the ergodic distributions for $x_{t+1} = \rho x_t + \varepsilon_t$ for $\rho = 0.1, 0.5, 0.9$, and $\varepsilon_t \sim U[-1, 1]$. Next compute the ergodic distribution for $x_{t+1} = \rho \sqrt{x_t} + \varepsilon_t$ for $\rho = 0.1, 0.5, 0.9$, and $\varepsilon_t \sim U[0, 1]$. Repeat these exercises with $\ln \varepsilon_t \sim N(0, 1)$.